Design of Run Time Monitors for Concurrent Programs

David Helmbold

Doug Bryan

Technical Report No. CSL-TR-89-395
Program Analysis & Verification Group Report 43

October 1989

This work was supported by the Air Force Office of Scientific Research, under Grant AFOSR 87-0150C.

Design of Run Time Monitors for Concurrent Programs

David Helmbold
Doug Bryan

Computer Systems Laboratory
Stanford University
Stanford, California 95030

October 1989

Computer Systems Laboratory CSL-TR-89-395
Program Analysis & Verification Group Report 43

Abstract

We address the problem of correctly monitoring the run time behavior of a concurrent program. We view a program as having three (potentially different) sets of behavior: computations of the original program when monitoring is not performed, computations after the monitor is added to the program, and "observations" produced by the monitor. Using these sets of behaviors, we define four properties of monitor systems: non-interference, safety, accuracy and correctness. We define both a minimal level and a total level for each of these properties. The non-interference and safety properties address the degree to which the presence of the monitor alters a computation (the differences between the first two sets of computations). Accuracy is a relationship between a monitored computation and the observation of the computation produced by the monitor. Correctness is a relationship between observations and the unmonitored computations.

A run time monitor for TSL-1 and Ada has been implemented. This monitor system uses two techniques for constructing the observation. We show that any monitoring system using these two techniques is at least minimally correct, from which the (minimal) correctness of the TSL-1 monitor follows.

Keywords: TSL; runtime monitor; run time; concurrent software; partial orders; observations.

Design of Run Time Monitors for Concurrent Programs

David Helmbold
Computer Information Sciences
University of California at Santa Cruz

Doug Bryan*
Computer Systems Laboratory
Stanford University

September 28, 1989

1 Introduction

We address the problem of correctly monitoring the run time behavior of a concurrent program. This paper presents the general principles involved in the design of such a monitor. Ada [1] will be used as an example programming language.

We view a program as having three (potentially different) sets of behavior. The first set of behaviors are the possible computations of the original program when monitoring is not performed. The second set contains the possible computations after the monitor is added to the program. The third set is the possible "observations" produced by the monitor. Using these sets of behaviors, we define four properties of monitor systems: non-interference, safety, accuracy and correctness. Although absolute safety, correctness, etc. is desirable, they may not be achievable without substantial performance penalties. Therefore we define both a minimal level and a total level for each of these properties. The non-interference and safety properties address-the degree to which the presence of the monitor alters a computation (the differences between the first two sets of computations). Accuracy is a relationship between a monitored computation and the observation of the computation produced by the monitor. Correctness is a relationship between observations and the unmonitored computations.

A run time monitor for TSL-1 [2,3] has been implemented. This monitor system uses two techniques for constructing the observation. We show that any monitoring system using these two techniques is at least minimally correct, from which the (minimal) correctness of the TSL-1 monitor follows.

2 Basic Capabilities of a Monitor

The functional capabilities of a monitor may be categorized as follows:

*This work was supported by the U.S. Air Force Office of Scientific Research under contract 87-0150A.

1. obtain information on the events performed by the program,
2. construct an observation of the underlying computation,
3. compile or interpret program specifications,
4. check for consistency between the observation and the specifications, and
5. interact with the user and report specification violations.

The monitor must be made aware of observable events performed by the program. This reporting can take place either at run time [4,5,6], where the monitor executes concurrently with the program, or during post-execution analysis [7,8]. We feel that run time monitoring is preferable because it enables the monitor to be used in conjunction with other run time testing tools and provides a monitoring capability for programs which (may) never terminate. The events are commonly reported through either synchronous or asynchronous message passing mechanisms. The method in which events are reported will affect both the real time behavior of the underlying program when monitoring is performed, as well as influencing the difficulty of preserving (or reconstructing) an observation of the computation. Our current monitor uses primarily synchronous message passing.

Our monitor contains an interactive user interface. This interface allows the user to issue ad hoc queries to the monitor during analysis. Through this query capability, the user should be able to determine the cause of specification violations.

The third and fourth capabilities listed above are the core of our monitor. Specifications may be handled in a number of ways:

- they may be compiled into the monitor,
- they may be passed to the monitor at run time as a part of the observation itself, or
- they may be passed to the monitor through the user interface.

The last two methods require the monitor to interpret specifications at run time. The second method is used in the implementation of the monitor for TSL-1. High-level specifications are preprocessed (at compile time) into an intermediate graph representation. This representation is passed to the monitor at run time. The monitor constructs a *token graph* for each specification and *event matching* causes tokens to propagate through these graphs [9,10]. Specification violation is assessed whenever a token reaches the "finish" node of a graph.

There are also many ways in which monitors can check specifications. Specification analysis techniques, in general, can be grouped into two categories: proof and run time checking. Either method, or a combination of them, can be used in monitoring concurrent programs. In the area of sequential program specification, for example, techniques which involve a combination of run time checking and theorem proving have been implemented [11,12].

3 A Paradigm for Describing Program Behavior

In order to monitor a program, aspects of its behavior must be made observable. The general term *action* is given to these observable aspects of a program. Actions represent abstractions of parts of a program's computations. Therefore, each program will have (i.e., declare) its own fixed and finite set of relevant actions. In general, an action represents a small fragment of a program; the set of actions declared cover only small sub-computations of complete computations.

The execution of an action generates an *event.* An event is an instance of an action and is dynamic, i.e., defined during a computation. In general, events can be thought of as tuples having two or more components; the two required components being an indication of the action performed and the task (independent thread of control) executing the action.

The action $calls$ is an example of an action used to describe Ada computations. It is executed whenever a task calls an entry of some task. Events generated by the execution of this action will contain as components the calling task, the action $calls$, the task called, and the entry called.

A program and its input define a set of computations, C. We view each computation as a (possibly infinite) set of events, together with a partial ordering on the events. Although the same action may be performed many times by a single task, we assume that each event is uniquely identified. If two events have equal components [same action, same task, etc.) they are tagged in some manner to make them distinguishable. For example, in the specifications described in [13], like events are tagged with a count indicating the number of times they have occurred earlier in the computation. For an alternative view where events are allowed to be indistinguishable, see [14].

When a monitor is added to a program, a new program is created. This new program typically has an augmented set of actions and a different set of computations, denoted by $\mathcal{M}$. Each monitored computation, $M \in M$, produces an *observation* of the program, O_M. We define the set of observations, $\mathcal{O}$, as $\{ O_M : M \in M\}$. Observations, like computations, are a partial ordering of events. It is important that $\mathcal{O}$, the set of possible observations, agree with C, the set of possible (unmonitored) computations. This will be discussed at length in Section 4.

In a computation, all events generated by a given task are linearly ordered by the times that the task executed the actions. We use < to denote this ordering between events generated by the same task. Although all events generated by the same task are ordered, it is often not the case that events generated by different tasks are ordered. Therefore one can not assume that computations are totally ordered. One may argue that a total ordering may be *defined* for all events, however a partial ordering of the events is more accurate to *observe* [14,15,16,17]. For example, it may be said that the events are ordered according to the "time" at which they are performed, relative to some global clock. However, constructing such a global clock for distributed systems would be inefficient and would not accurately model the concurrent aspects of a system.

The < operation together with a "causal" relationship between events of different tasks defines a partial order over the set of events of a computation. The causal relation ($\leadsto$) can only be defined in terms of specific programming features; the semantics of this relation must be guaranteed by the semantics of the programming language or programming environment being used. Informally, $e \leadsto f$ within a computation if f cannot be generated unless e has been generated. CSP-based [18] tasking models, such as that of Ada, have features which guarantee the causal relationship between particular events. Given a thorough set of actions corresponding to the tasking constructs of Ada, the following causal relations, among others, are defined by the language:

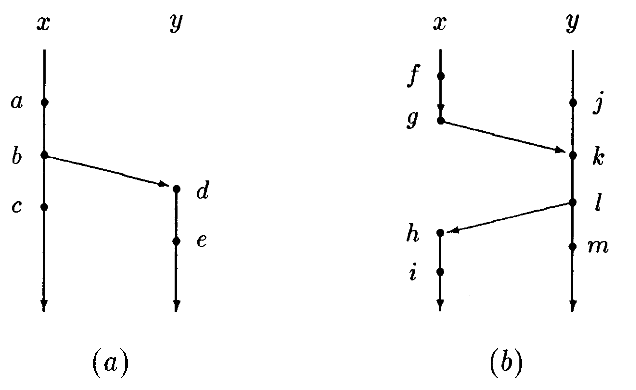

Figure 1: Partial Orders for Activation and Rendezvous.

(a) the first event of task y "is caused by" task x activating y,
(b) task y accepting a call from task x "is caused by" x calling y, and
(c) an event of x immediately following a rendezvous with y "is caused by" y ending the rendezvous with x.

Figure 1 illustrates these relationships. Vertical edges represent orderings defined by $<$ and diagonal edges represent orderings defined by $\leadsto$. In Figure l(a), b represents the event used to observe that task x activates task y, thus $b \leadsto d$. In Figure l(b), g represents x calling y, k represents y accepting the call and beginning a rendezvous, and l represents y ending the rendezvous with x. Thus g $\leadsto k$ and $l \leadsto h$. *(See* [19] for a complete specification of $\leadsto$ as defined by Ada.)

Let $\prec$ denote the partial order over the set of events of a computation. $\prec$ is defined as the transitive closure of the disjunction of $<$ and $\leadsto$.

4 Properties of Monitors

The issues to be addressed in this section include relationships between the set of possible (unmonitored) program executions, C, monitored executions, M, and observations produced by the monitor, $\mathcal{O}$. Of particular importance are how the presence of the monitor alters the program's functional behavior (i.e., monitor interference) and the accuracy of the monitor's observations.

It is desirable that a monitor not significantly alter the semantics of a program and that a monitor's observation of a computation accurately reflects the computation itself. To more formally state these concerns, notation is introduced to describe the partially ordered sets (POset) representing computations and observations.

A POset, (V,E), consists of two sets; V is a set of vertices and E is an asymmetric transitively closed relation on the vertices. For our purposes we consider partial orders where V is a set of events performed and E represents temporal relationships between the events. Note that the restrictions on E are met by $\prec$, the partial order on the events of a computation.

The POset $C = (V_C, E_C)$ denotes a particular computation in $\mathcal{C}$. When a monitor is added to the program, "monitoring events" may be performed and the resulting set of possible computations $\mathcal{M}$ may differ from $\mathcal{C}$. We use $M = (V_M, E_M)$ for a particular monitored computation. Each monitored computation produces a single observation. O denotes a particular observation, and O_M denotes the observation produced by the monitored computation M. Note that unmonitored computations, monitored computations, and observations are all POsets.

The events in a monitored computation fall into two categories. There are the events performable in unmonitored computations, as well as (possibly) some additional events performed as part of the monitoring process. Given a particular monitored computation M, $V_{M,\mathcal{M}}$ denotes the events of V_M which is part of the monitoring process and $V_{M,\mathcal{C}}$ denotes the events performable in an unmonitored computation. Thus $V_{M,\mathcal{M}}$ and $V_{M,\mathcal{C}}$ are disjoint and their union is V_M. $E_{M,\mathcal{C}}$ denotes the set of edges $\{(v, v') : (v, v') \in E_M \wedge v, v' \in V_{M,\mathcal{C}}\}$ where (v, v') denotes a directed edge from v to v'.

The four desirable properties for monitoring schemes *we* define are: *non-interference, safety, accuracy,* and *correctness*. We first give an intuitive English description of these terms and then present a formal definition using the notation developed above.

- A non-interfering monitor allows the monitored program to do the same things as the unmonitored version.
- A safe monitor does not allow the monitored program to produce behaviors that the unmonitored version is not capable of.
- An observation produced by an accurate monitor reflects the activity of the original program executing within the monitored computation producing the observation.
- The observations produced by a correct monitor reflect a possible computation of the original program, and a correct monitor is capable of producing an observation reflecting each of the possible unmonitored computations.

There are several different degrees of non-interference, differing in how "same" is defined. Similarly, there are different degrees of accuracy and correctness depending on how "reflect" is interpreted. These issues are made precise with the following definitions.

Two degrees of non-interference are

$$\textit{total} \qquad \forall C \in \mathcal{C} : \exists M \in \mathcal{M} \text{ s.t. } C = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$$

$$\textit{minimal} \ \forall C \in \mathcal{C} : \exists M \in \mathcal{M} \text{ s.t. } V_C = V_{M,\mathcal{C}} \wedge E_C \subset E_{M,\mathcal{C}}$$

and two degrees of safety are:

$$\textit{total} \qquad \forall M \in \mathcal{M} : \exists C \in \mathcal{C} \text{ s.t. } C = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$$

$$\textit{minimal} \ \forall M \in \mathcal{M} : \exists C \in \mathcal{C} \text{ s.t. } V_C = V_{M,\mathcal{C}} \wedge E_C \subset E_{M,\mathcal{C}}$$

Non-interference and safety address the relationships between C and M. In a sense, they are duals of each other. This can be seen in the symmetry of their formal definitions. Note that in all four definitions $V_C = V_{M,\mathcal{C}}$. That is, even the minimal properties require the monitor to preserve the entire set of events of an unmonitored computation. The difference between minimally and totally satisfying one of these two properties has only to do with the introduction of new orderings. Specifically, monitored computations must preserve all orderings of an unmonitored computation.

Two degrees of accuracy are:

$$\textit{total} \qquad \forall M \in \mathcal{M} : O_M = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$$

$$\textit{minimal} \quad \forall M \in \mathcal{M} : V_{O_M} = V_{M,\mathcal{C}} \wedge E_{M,\mathcal{C}} \subset E_{O_M}$$

Accuracy addresses the differences between the sets $\mathcal{M}$ and $\mathcal{O}$, not the differences between C and $\mathcal{O}$; the accuracy of an observation is concerned with the (monitored) computation producing the observation, regardless of any unmonitored computations that may exist.

Lastly, correctness is concerned with the relationships between C and $\mathcal{O}$. Two degrees of correctness are:

$$\textit{total} \quad \begin{array}{l}(\forall M \in \mathcal{M} : \exists C \in \mathcal{C} \text{ s.t. } C = O_M) \wedge \\ (\forall C \in \mathcal{C} : \exists M \in \mathcal{M} \text{ s.t. } O_M = C)\end{array}$$

$$\textit{minimal} \quad \begin{array}{l}(\forall M \in \mathcal{M} : \exists C \in \mathcal{C} \text{ s.t. } V_C = V_{O_M} \wedge E_C \subset E_{O_M}) \wedge \\ (\forall C \in \mathcal{C} : \exists M \in \mathcal{M} \text{ s.t. } V_{O_M} = V_C \wedge E_C \subset E_{O_M})\end{array}$$

$\mathcal{O}$ is implicit in these definitions since $\mathcal{O} = \{O_M : \exists M \in \mathcal{M}\}$. Again, all events of an unmonitored computation are preserved in both the total and minimal definitions. Total correctness means that the observation is equivalent to an unmonitored computation. A minimally correct monitor must preserve the set of events and set of orderings, but may add additional orderings.

It should be clear that total correctness (non-interference, safety, or accuracy) implies minimal correctness (non-interference, safety, or accuracy).

Lemma 1 *Any totally non-interfering, totally safe, and totally accurate monitoring system is also totally correct.*

Proof: Assume that we are given a totally non-interfering, totally safe, and totally accurate monitoring system.

Let $M \in \mathcal{M}$ be any execution of a monitored program. Since the monitor system is totally accurate, $O_M = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$. Since the system is totally safe, there is an execution of the original program, C, such that $C = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$. Therefore $C = O_M$, and the system meets the first condition for total correctness.

Now let $C \in \mathcal{C}$ be any execution of a program. Since the system is totally non-interfering, there is a $M \in \mathcal{M}$ such that $C = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$. By the total accuracy of the system, $O_M = (V_{M,\mathcal{C}}, E_{M,\mathcal{C}})$. Therefore $C = O_M$, and the system meets the second condition for total correctness. □

Lemma 2 *Any minimally non-interfering, minimally safe, and minimally accurate monitoring system is also minimally correct.*

Proof: Assume that we are given a minimally non-interfering, minimally safe, and minimally accurate monitoring system.

Let $M \in \mathcal{M}$ be any execution of a monitored program. Since the monitor system is minimally accurate, $V_{O_M} = V_{M,\mathcal{C}}$ and $E_{M,\mathcal{C}} \subset E_{O_M}$. Since the system is minimally safe, there is an execution of the original program, C, such that $V_C = V_{M,\mathcal{C}}$ and $E_C \subset E_{M,\mathcal{C}}$. Therefore $V_C = V_{O_M}$, $E_C \subset E_{O_M}$, and the system meets the first condition for minimal correctness.

Now let $C \in \mathcal{C}$ be any execution of a program. Since the system is minimally non-interfering, there is a $M \in \mathcal{M}$ such that $V_C = V_{M,\mathcal{C}}$ and $E_C \subset E_{M,\mathcal{C}}$. By the minimal accuracy of the system, $V_{M,\mathcal{C}} = V_{O_M}$ and $E_{M,\mathcal{C}} \subset E_{O_M}$. Therefore $V_C = V_{O_M}$, $E_C \subset E_{O_M}$, and the system meets the second condition for minimal correctness. □

5 Observation Construction Disciplines

The standard method for monitoring a computation (at the source level) works as follows. A transformation process modifies the original program so that whenever an action *a* is performed, one or more new actions are also performed. These new actions notify the monitor that action *a* has been (or will be) performed, enabling it to create an observation.

The TSL-1 monitor is implemented as a single, pure server task, simply receiving notifications and processing them. Therefore its addition does not cause any deadlocks. Any deadlocks that do occur result from dependencies between the original tasks. The additional monitoring events do not perturb the state of tasks in the original program, resulting in the following:

Consequence 1 *The state of a task is not changed by performing the additional actions inserted by the transformation process.*

The above consequence implies that all else being equal, a transformed task is capable of performing the same sequence of events as its untransformed version. Unfortunately, a task's computation depends not only on the values it receives, but also on the order and relative time in which those values are received. For example, the order of arrival of two entry calls could have a large effect on the computation of a task. Similarly, whether or not an entry call arrives before the delay expires in a selective wait statement [1, §9.7.1] could change the task's computation.

These are examples of how an Ada program can have many possible computations, even when running on the same input. This results in part from timing issues like the above, and in part from non-deterministic features in the language. Although a single Ada environment (or scheduler) may be unable to produce all of the possible computations of a program, we take a system-independent view and treat any behavior consistent with the Ada semantics as a potential computation. In particular, Ada makes no guarantee as to how long any particular operation takes to execute.

Even the guarantees provided by the Ada priority mechanism are very weak. The intent is that a high priority task will never be waiting for a processor if low priority tasks are executing. If there are enough processors so that all executable tasks are running, priorities are irrelevant. Furthermore, the different processors may not run at the same speed; a processor's relative speed may be an arbitrarily complicated function depending on many variables. This gives us the following:

Consequence 2 Any task (or tasks) may perform an arbitrary amount *of* work (not involving communication *with* other tasks) without *the* remainder *of* the program performing any significant computation.

It may be useful to think of the above consequence in terms of scheduling the tasks (without priorities) on a single processor. The above consequence corresponds to a random scheduler, one which picks the next task to execute at random.

Consequence 2 is admittedly a very strong assumption. However, it is unlikely that any monitoring system can be proven non-interfering without strong assumptions about the underlying method of scheduling the program.

Lemma 3 Any monitor system for which Consequences *1* and 2 hold is (at least) minimally *non-interfering*.

Proof: Let C be any execution of the original program. We construct a monitored execution *M* such that $V_C = V_{M,C}$ and that if e $\prec$ e' in C then e $\prec e'$ in *M*. Topologically number the events in C, $\{e_1, e_2, e_3, \ldots, e_n\}$ so that whenever $e_i \prec e_j$ then $i <$ j. Let t_i be the task performing e;. Ignore timing issues for the moment and concentrate on the states of the various tasks as they perform the events in C.

Each e_i is performed by t_i whose external input at that point is some subset of e_1 through e_{i-1}. Consequence 1 implies that if t_i gets the same input then t_i is able to perform e_i in the monitored computation exactly as it did in C (possibly subject to non-deterministic choices made by t_i). Let *M* be the monitored computation constructed as follows:

> The monitor and the task performing e_1 both start out executing on "fast" processors, while the other tasks run on "slow" processors. Since the input to t_1 is the same in both cases (the empty set), t_1 is capable of performing e_1. Assume that it does.
>
> Now let t_1's processor slow down and t_2's processor speed up. t_2 is then capable of performing e_2. Repeat this for all successive e,.

The monitored computation M clearly contains all of the events of C. Furthermore, the only other events in M are added by the transformation, so $V_C = V_{M,\mathcal{C}}$. If $e_i \prec e_j$ in the unmonitored computation then the events are performed by the same task or e_i causally effects e_j. Since they are the same events in M, they will again be performed by the same tasks or e_i must still effect e_j in the monitored computation. Therefore, whenever $e_i \prec e_j$ in C, $e_i \prec e_j$ in M.

Now consider the timing issues. First, assure that the inputs to the tasks arrive in the same order in M as in C. This is easily seen since the causal relationships between events in C are preserved. Secondly, assure that races between input events and delays are resolved in the same way. By controlling the relative speeds of the processors one can insure that all races have the same resolution in the monitored computation as in the unmonitored computation. $\square$

Lemma 4 Any monitor system *for which* Consequences *1* and 2 hold is ***(at*** least) minimally *safe.*

Proof: Reverse the construction of Lemma 3. $\square$

5.1 Synchronous Reporting

Now define a simple transformation discipline for inserting the new actions which guarantees that Consequences 1 and 2 will hold. Call this discipline *synchronous* ***reporting.***

In the synchronous reporting discipline, the monitor consists of a single task. The notifications received by the monitor are synchronous, so that the monitor receives one notification at a time and the task sending the notification is suspended until the notification has been received and processed. The observation constructed by the monitor is the (totally ordered) list of event notifications.

Distinguish three fundamentally different kinds of events generated by unmonitored computations:

- *Sequential events* are those events which are not related to any other event by the $\leadsto$ relation.
- *Blocked events* are those events which appear on the right-hand side of a $\leadsto$ relationship. A blocked event can not occur until the appropriate event(s) which precede it have occurred.
- *Enabling events* are those events which appear on the left-hand side of the $\leadsto$. Enabling events must be performed before the corresponding blocked events.

Any event e which is both a blocked event and an enabling event can conceptually be split into a "start e" subevent and an "end e" subevent. The ***start*** e subevent will be a blocked event (but not an enabling event) and the *end* e subevent will be an enabling event (but not a blocked event).

The key to the synchronous reporting discipline is the order of the new notification events with respect to the events being signaled to the monitor—the order of notification and $V_{M,\mathcal{C}}$ events within $\mathcal{M}$. The event which notifies the monitor of a sequential event can be generated either before or after the sequential event occurs. The event which notifies the monitor of a blocked event must be generated *after* the blocked event occurs. The event which notifies the monitor of an enabling event must be generated *before* the enabling event. In all three cases, no other events may be generated by the original task between the generation of the notifying event and the event being reported. That is, if n is the event notifying the monitor of t generating e, then there must not exist any event e' performed by t such that, $n \prec e' \prec e$, or $e \prec e' \prec n$, or e' and n are unordered.

Lemma 5 *If* synchronous reporting is *used for every* event, then *the* monitor *system is* (at *least)* minimally accurate.

Proof: Let M be any execution of a monitor system using synchronous reporting. One notification is sent to the monitor for every event in $V_{M,\mathcal{C}}$, and every notification sent to the monitor is for an event in $V_{M,\mathcal{C}}$. Therefore each event (and only those events) in $V_{M,\mathcal{C}}$ appear in the observation, so $V_{M,\mathcal{C}} = V_{O_M}$.

Now show that if $e \prec e'$ in $V_{M,\mathcal{C}}$, then e appears before e' in the observation. Since the $\prec$ relationship is the transitive closure of the $\rightsquigarrow$ and $<$ relations, it suffices to show that whenever $e \rightsquigarrow e'$ or $e < e'$ then e precedes e' in the observation. Consider these two cases separately.

First, consider the case when $e < e'$. This means that e and e' are performed by the same task, say task t. The transformation process ensures that task t performs the events notifying the monitor of e before t performs the events notifying the monitor of e'. Since the notifying task is blocked while the monitor processes the notification, e will appear in the observation before e'.

Now consider the case when $e \rightsquigarrow e'$. Since e is an enabling event, its notification will be processed before e is performed. Similarly, since e' is a blocked event, its notification will be processed after e' is performed. Because $e \rightsquigarrow e'$, event e must be performed before event e' in M. Therefore, the notification of e is processed before e is performed before e' is performed before the notification of e' is processed. Thus e appears in the observation before e'. $\square$

Theorem 1 *If* synchronous reporting is *used for every* event, then *the* monitor *system* is (at least) minimally *correct.*

Proof: This follows from Lemmas 2, 3, 4, and 5. $\square$

5.2 Derived Reporting

Synchronous reporting can only be used when additional statements can be inserted around the statements performing an action. Unfortunately, notifications for some Ada events (namely task termination, scope termination, and abortion) can not be generated in this way. Notifications for these events must be *derived* by the monitor from the other notifications that it receives. The abort statement and terminate alternative of the selective wait statement greatly complicate the derived notifications generated by our monitor. A detailed discussion of these peculiarities of Ada is beyond the scope of this paper. (See [20] for a TSL-1 specification of the Ada semantics of abortion and termination. See [19] for a specification of the causal relationships ($\leadsto$) defined by these features.)

An Ada task may terminate simply by reaching the end of its block. This is a special case of *block termination*. The event *block-completion* occurs after the last statement in the block has been executed. The monitor is notified of block-completion using the synchronous reporting discipline.

A block, b, terminates when it has completed *and* all tasks dependent on b have terminated. If block b has no dependents (or the monitor has already derived termination events for all dependent tasks), then the monitor *derives* a block termination event while processing the *block-completion* event of b. Otherwise there will be a last notification of a task termination event enabling the block termination event. While processing this last notification, the monitor *derives* and processes a termination notification for block b, causing the event "b terminates" to appear in the observation.

Consequence 3 *No events* are *performed within* a *block between its completion* and its termination.

Consequence 4 Notifications for the terminations of all dependent tasks (and nested blocks) of block b, and the notification for the termination of b *itself*, are processed before block b actually terminates.

Task termination in Ada can be both a blocked event (awaiting the termination of all its dependent tasks) and an enabling event (allowing its parent task or block to terminate). In addition, a block termination can only enable and be blocked by other block terminations, resulting in:

Consequence 5 If $e \leadsto e'$ and either e or e' is a block termination event, then both e and e' are *block* termination events.

Since Consequence 1 holds for both synchronous reporting and derived reporting (as well as their combination) it is known that the monitor is still at least minimally non-interfering and minimally safe (Lemmas 3 and 4). It remains to show that the monitor using derived reporting is still minimally accurate, and thus minimally correct.

Let M be a monitored computation generating the observation O_M (using a combination of derived and synchronous reportings). The observation contains one event for each notification generated by the synchronous reporting discipline, one block termination event for each block of the original program that terminates, and no other events. Therefore, the observation contains exactly those events arising from an unmonitored computation. Now prove that the ordering of the observation is consistent with the event ordering from the monitored computation, and thus the monitor is minimally accurate.

Lemma 6 The ordering *of events in the observation is consistent with the* partial ordering *induced by the computation.*

Proof: Since the events in the computation are ordered by the transitive closure of $<$ and $\leadsto$, it suffices to show that whenever e $<$ e' or e $\leadsto$ e' in M then e $\prec$ e' in O_M.

First, consider the case when e $<$ e' in M. This means that e and e' are performed by the same task. If synchronous reporting is used to notify the monitor of both e and e', then the notification for e will be processed first (Lemma 5) and the notification of e' will be processed second.

If derived reporting is used for e', e must be a block completion event. The derived reporting discipline ensures that block completion notifications are processed before the (derived) block termination notification. Therefore, if e $<$ e' in M, then e $\prec$ e' in the observation.

If e is a block termination event, then the monitor produces a derived notification of e before the block terminates (Consequence 4). Synchronous reporting is used to notify the monitor of event e'. Therefore the notification of e' will only be sent to the monitor after the block has terminated, and thus e' follows e in the observation.

Now consider the case when e $\leadsto$ e' in M. Either both e and e' are block termination events or synchronous reporting is used for both e and e' (Consequence 5). When synchronous reporting is used for both e and e', e appears in the observation before e' by the argument in Lemma 5.

Now consider the case where derived reporting is used for both e and e'. In this case both e and e' are block termination events. Since e $\leadsto$ e', e must be the termination of a dependent of e'. The derived reporting procedure guarantee that all dependents terminations are entered into the observation before the enclosing block's termination (Consequence 4). Therefore, e precedes e' in the observation. $\square$

Theorem 2 *The TSL-I* monitoring *system is* minimally correct.

Proof: The TSL-1 system uses only synchronous and derived reporting. $\square$

6 Alternative Reporting Techniques

Though it has been shown how synchronous and derived reporting can be used to form a minimally correct monitor, these disciplines have a number of disadvantages. One is that synchronous reporting will significantly affect the real time behavior of computations since tasks of C are suspended while observations are processed. Another is that these disciplines result in totally ordered observations, even when a partial ordering of the events may be more accurate. This section briefly outlines other methods of constructing observations which address these disadvantages.

The most apparent method of minimizing the real time effects of event reporting is to report using asynchronous communication. Figure 2 illustrates one method of asynchronous event reporting. In this approach the tasks of C place event reports in local queues. The tasks are suspended only during this enqueue operation. A new module, *merger layer,* is then needed to construct an

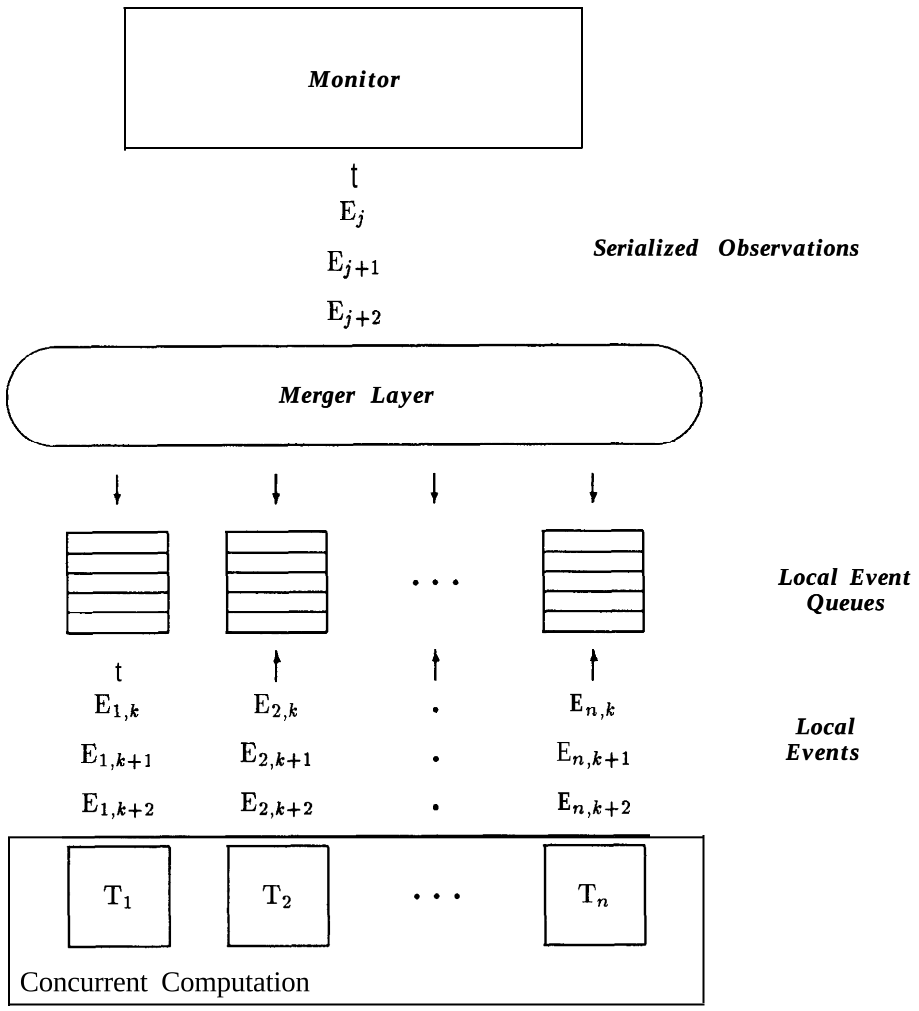

Figure 2: Asynchronous Event Reporting.

observation. There are a number of algorithms which could be used by the merger layer. One would be to simply implement the causal relation $\leadsto$ presented earlier. Since this type of monitor still serializes events, it could never be made totally correct.

In general, a causal relation of a programming language or environment may be handled in the same ways that user specifications are handled: the definition of the relation may be compiled into the monitor, the semantics of the relation may be passed to the monitor at run time, or the relation may be defined through the monitor's user interface. The last two methods allow one to construct language independent monitors.

A disadvantage of asynchronous event reporting is that it would be possible for the computation to "get ahead" of the monitor. That is, events could be reported while previous, ordered events have yet to be processed. This would restrict the capability of the monitor to be used in conjunction with other run time tools. Also, assuming that the computation may "get ahead" of the monitor, the size (memory requirements) of the local event queues becomes problematic. If a queue ever becomes full, the computation must effectively resort back to synchronous event reporting. Lastly, note that some synchronization is required for the tasks of C and the monitor to share queues. The time that the computation is blocked during event reporting would, however, still be significantly less than if the computation was blocked for the duration of event notification processing.

Another method of minimizing the effects of monitoring is to simply make the monitor execute faster. This could be done by distributing the monitor itself. Methods of distributing the monitor are currently being explored.

An alternative approach to constructing an observation is to transform the program so that each event is "stamped" in such a manner that the partial order is apparent. This was explored in [15]. Though the algorithms described in [15] produce partially ordered observations, they do not produce the partial orders described in this paper. Specifically, [15] introduces new orderings not present in unmonitored computations and thus could not be used to construct a totally correct monitor. More recent algorithms for stamping events [17,21,22] use a vector of stamps, rather than a single scalar stamp. This approach appears capable of producing totally correct monitors. However, maintaining and communicating the stamp vectors increases the cost of monitoring, especially for systems with large numbers of tasks.

The paradigms of Figure 2 and [17] are examples of two degrees of correctness; a monitor like that shown in Figure 2 is less correct since it serializes observations; a monitor based on the algorithms of [17] is the more correct since it preserves the partial ordering of events in unmonitored computations; monitors using the scalar time stamps of [15] fall somewhere in between.

Note that in the definitions of accuracy and correctness given in Section 4, accuracy deals only with the relationships between the monitored computation and the resulting observation. If a monitor were to use synchronous reporting together with a time stamp vectors algorithm, the monitor would not be totally accurate since synchronous reporting will always introduce orderings into M that are not present in some C. That is, for such a monitor E_{O_M} would be a strict subset of $E_{M,\mathcal{C}}$. This monitor would be neither minimally nor totally accurate by our definitions. This monitor could though, be totally correct.

7 Conclusions

Four independent properties of monitors have been defined. The major two design issues effecting a monitors ability to satisfy these properties are the reporting of events and the construction of observations. Event reporting (e.g., synchronous, derived) will impact the non-interference and safety of a monitor. The way in which observations are constructed (e.g., totally ordered according to notification, scalar time stamps, vector time stamps) will influence the accuracy and correctness of a monitor. These design issues are independent in that one can construct a monitoring system which is not totally non-interfering, yet is totally correct. Further, a monitor which is not totally accurate may still construct totally correct observations.

Totally satisfying any one of the properties offers the designer a number of technical challenges. Yet it is felt that building a totally correct run time monitor is within the state-of-the-art.

Asynchronous event reporting does reduce the real time effect of monitoring, the main disadvantage being that synchronization between the computation and testing is lost. Thus, for finite computations, this method of monitoring has few advantages over post-execution analysis.

A monitor for Ada and TSL-1 which satisfies the four minimal properties given in Section 4 has been implemented. Use of the monitor indicates that it is a useful and practical tool for testing and debugging concurrent programs.

8 Acknowledgements

We thank Geoff Mendal, Sriram Sankar and David Luckham for their comments on earlier drafts of the paper, as well as the entire TSL group at Stanford University for several interesting discussions.

References

[1] *Reference Manual for the Ada Programming Language.* U. S. Department of Defense, U. S. Government Printing Office, ANSI/MIL-STD-1815A edition, January 1983.

[2] D. Helmbold and D. Luckham. TSL: task sequencing language. In *Proceedings of the Ada International Conference* 1985, pages 255-274, Cambridge University Press, Paris, France, 14-16 May 1985.

[3] D.C. Luckham, D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. Task sequencing language for specifying distributed Ada systems: TSL-1. In Habermann and Montanari, editors, *System Development and Ada, proceedings of the CRAI workshop on Software Factories and Ada. Lecture Notes in Computer Science. Number 275,* pages 249-305, Springer-Verlag, May 1986.

[4] David J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers,* C-31(7):681–685, July 1982.

[5] P.C. Bates and J.C. Wileden. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems and Software,* (3):255–264, 1983. Also ACM Softw. Eng. Notes 8(4), ACM SIGPLAN Notices 18(3).

[6] D.L. Bryan. Run-time monitoring of tasking behavior using a specification language. In *Second Workshop on Large Grain Parallelism,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 11-14 October 1987. An extended abstract.

[7] C.H. Ledoux and D.S. Parker. Saving traces for Ada debugging. In *Proceedings of the Ada International Conference 1985,* pages 97-108, Cambridge University Press, Paris, France, 14-16 May 1985.

[8] A.A. Hough and J.E. Cuny. Belvedere: prototype of a pattern-oriented debugger for highly parallel computation. In *Proc. International Conf. on Parallel Processing,* pages 735-738, 1987.

[9] D.L. Bryan. The implementation of a run time monitor for concurrent programs. Forthcoming technical report.

[10] D.P. Helmbold. *The Meaning of TSL: An Abstract Implementation of TSL-1.* Technical Report CSL-TR-88-353, Computer Systems Laboratory, Stanford University, March 1988. Also published by Computer Information Sciences Board, UC Santa Cruz as UCSC–CRL–87–29.

[11] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, forthcoming.

[12] S. Sankar. Incremental theorem proving techniques for runtime checking against algebraic specifications. Unpublished draft.

[13] C.J. Fidge. Partial orders for parallel debugging. In *Workshop on Parallel and Distributed Debugging,* pages 183-194, ACM SIGPLAN/SIGOPS, Madison, Wisconsin, May 5-6 1988.

[14] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming,* 15(1):33–71, 1986.

[15] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM,* 21(7):558–565, 1978.

[16] F. Mattern. *Determining the Partial Order of Distributed Events.* Technical Report SFB124–28/87, University of Kaiserslautern, Federal Republic of Germany, 1987.

[17] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications,* 10(1):56–66, February 1988.

[18] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM,* 21(8):666-677, 1978.

[19] D. Bryan. An algebraic specification of the partial orders generated by concurrent ada computations. In *Proc. Tri-Ada* '89, ACM Press, Oct. 1989. Forthcoming.

[20] D.S. Rosenblum. *Design and Verification of Distributed Tasking Supervisors for Concurrent Programming Languages.* Technical Report CSL-TR-88-357, Computer Systems Laboratory, Stanford University, 1988. (Ph.D. thesis).

[21] F. Mattern. *Virtual Time and Global States of Distributed Systems.* Technical Report SFB124–38/88, University of Kaiserslautern, Federal Republic of Germany, October 1988. To appear in, M. Cosnard (ed.) "Proceedings of Parallel and Distributed Algorithms", Elsevier Science Publishers (North-Holland).

[22] P. Leu and B. Bhargava. Multidimensional timestamp protocols for concurrency control. *IEEE Transactions on Software Engineering,* SE-13(12):1238–1253, 1987.

Contents