Debugging Distributed Programs through the Detection of Simultaneous Events *

Madalene Spezialetti
EECS Dept., Packard Lab
Lehigh University
Bethlehem, PA 18015

Rajiv Gupta
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Event based debuggers for distributed systems automatically detect occurrences of user specified events. During debugging it is not always possible to breakpoint a computation in a state that reflects an event occurrence. Thus, to avoid unnecessary breakpoints we must develop techniques that determine, prior to the initiation of a breakpoint, whether an event occurrence will be captured by the breakpoint. In this paper we propose the **simultaneity** *operator for achieving the above goal. This operator asserts that its operand events, which correspond to states of different processes in the distributed computation, are all* **true** *at a point in the program and* **stable** *with respect to this point. The stability of events guarantees that the initiation of a breakpoint at this point will leave the system in a state which reflects the event occurrence. We present static analysis techniques that identify points in a program at which user specified simultaneous events should be evaluated to avoid unnecessary breakpointing. The statically computed information also enables minimal instrumentation of the program for the detection of event occurrences.*

1 Introduction

As distributed computing capabilities become more widely available, the need for methodologies which aid in the analysis and debugging of distributed computations has increased. A common approach to debugging a distributed computation allows the user to specify the activity of interest in the form of predicates, or **event definitions** [2] [12]. A combination of temporal, arithmetic, logical, and relational operations allows users to construct complex event definitions which test the states of one or more processes. During debugging, typically, event definitions represent error conditions. Given the event definitions, a monitoring system automatically collects data pertaining to the activity specified in the definitions, and the events are evaluated utilizing this data [8] [12] [14]. When an event evaluation yields a true result, the occurrence of the event is recognized by the monitoring system and reported to the user. The recognition of events local to a single process is simple. However, when the event is global, that is, the event tests the states of multiple processes, the task of recognition is far more complex. Global events are constructed by combining local events at various processes using temporal operators that check if the occurrence of local events are ordered ($\rightarrow$) or concurrent ($\|$). For example, if a token is being used to implement mutual exclusion between two processes P_a and P_b, then the following global events detect error conditions which represent the violation of mutual exclusion.

CE= $LocalEvent_a \parallel LocalEvent_b$
= P_a acquires token $\parallel$ P_b acquires token
OE= $BeforeLocalEvent_a \rightarrow AfterLocalEvent_b$
= latest operation by P_a acquired the token $\rightarrow$ latest operation by P_b acquired the token

Following the recognition of an event, or an error, we would like to provide the user with the state of the computation to assist in ascertaining the cause of the error. One approach that has achieved some success is the breakpointing of the computation and providing the user with an opportunity to examine the erroneous state of the computation in order to ascertain the cause of the error [4] [10]. In some situations there is a significant delay between event occurrence and its detection which may inhibit breakpointing in error state. Even if the detection of an event occurrence is immediate, delay in breakpointing is caused because all processes must be informed that a breakpoint has been initiated. Since communication of messages takes time, processes will continue execution for some time before the breakpoint is actually achieved. Thus, the state of the computation captured by the breakpoint may not reflect the event which occurred. For example, if processes concurrently acquire the mutual exclusion token they must simultaneously hold the token at some point in time. However, although we may detect an occurrence of the concurrent event *CE* which indicates that processes P_a and P_b at some point in time did concurrently acquire the mutual exclusion token, the processes may not hold the token when breakpoint is actually achieved. Similarly although we may detect the ordered event *OE* which indicates that following process P_a acquiring the token, process P_b also successfully acquires the token, the states of the processes after breakpointing may

*Supported in part by the National Science Foundation through Grant CCR-9212020 to Lehigh University and a Presidential Young Investigator Award CCR-9157371 to the Univ. of Pittsburgh.

indicate that only one process holds the token.

Since breakpointing a distributed computation is a complex task which incurs high run-time cost we should avoid unnecessary initiation of breakpoints. To achieve this goal it is essential to develop techniques which determine whether an event occurrence will be reflected by the state captured through a breakpoint and this determination must be made prior to the initiation of the breakpoint at minimal run-time cost. In this paper, in order to identify useful breakpoints, we exploit the notion of stability of the component events of which the global event is composed. Furthermore, we detect the stability of component events through static analysis [1] and hence there is no run-time cost associated with this task.

Most events are **stable** (or true) over periods of time. For example, the event "Process P holds the token" becomes true as soon as the process acquires the token and continues to be true till the process releases the token. To detect global events we combine the true evaluation of local component events to detect a global event occurrence with the stability of the local component events to detect global events at program points that allow breakpointing of the distributed computation in a state in which the component events continue to hold. We introduce the **simultaneity** (==) operator that achieves this goal. This operator is formally defined as follows:

Definition 1.1: Given a distributed program and local events LE_1, LE_2, ... LE_n such that event LE_i is local to process P_i. A **simultaneous event** $==(LE_1, LE_2, \ldots LE_n)$, alternatively written as $LE_1==LE_2==\ldots==LE_n$, is true at point p in process P iff the following conditions hold:

Occurrence: At point p the process P ascertains that the component events of the simultaneous event, that is, LE_1, LE_2, ... LE_n, are all *true*.

Stability: The events LE_1, LE_2, ... LE_n, are *stable* with respect to p, that is, if P initiates a breakpoint at p, the events LE_1, LE_2, ... LE_n will be true in the state captured through the breakpoint.

The following simultaneous event SE formulates the error in the mutual exclusion example. The detection of this event will also breakpoint the processes P_a and P_b in a state in which they both hold the token.

SE= $LocalEvent_a == LocalEvent_b$
= P_a holds token == P_b holds token

Even though the events CE, OE, and SE are all distinct from each other, they formulate the occurrence of the same error. However, the events CE and OE detect all occurrences of the associated error event while the event SE only detects those occurrences of the error which are stable and therefore the error will also be reflected in the breakpoint state. The user can direct the debugging system to detect events CE (or OE) and SE. If an occurrence of SE is detected a breakpoint can be initiated. On the other hand if CE is detected the occurrence can be reported to the user but a breakpoint need not be initiated. Thus, simultaneous events can be used to enhance the debugging capabilities of an event based debugger.

The recognition of simultaneous events in a distributed environment is a complex task. Three primary factors contribute to its complexity, namely, the lack of a global shared memory, the lack of a global clock, and the need for a means to predict the stability of component events. To handle the lack of a shared memory we employ monitors that maintain information pertaining to the activities of various processes. Given the lack of a global clock, protocols can be utilized to maintain information with which to establish inter-process timing relationships [3] [7] [9] [11] [13]. The introduction of monitors, as well as the use of the above protocols for establishing timing relationships, results in significant communication costs. Finally to predict the stability of events we must must analyze the event semantics in concert with the activities of the processes to determine their affect on the local events.

We have developed a run-time efficient approach for the detection of simultaneous events which exploits information derived through **static analysis** of the distributed program. Techniques for static analysis of distributed programs have been a subject of significant recent interest due to the reason that no run-time overhead is associated with such analysis techniques [6]. The information gathered through static analysis allows us to predict the stability of events and reduce the run-time communication overhead. Existing techniques do not exploit semantic information that can be collected through static analysis. The monitoring overhead is reduced by communicating minimal information among the processes and piggybacking this information onto communication messages that are being generated as part of the distributed computation. In addition, our technique introduces event evaluations only at program points which are identified using static analysis at those points where an event could occur. In this way, the unnecessary saving or evaluation of state changes which could not result in an event recognition are eliminated. Furthermore, the recognition techniques in this approach are designed to permit the evaluation of events which span a number of processes without the utilization of the costly timing protocols. The transmission of event data and the corresponding event evaluations are introduced such that the required timing knowledge will be implicit in the presence of the necessary component data. Although in this paper we demonstrate the approach in the context of detecting simultaneous events, our approach has also been applied to the detection of other types of global events, namely ordered and concurrent events.

In section 2 we describe the program representation on which our analysis algorithms operate. In section 3 we first describe conditions under which potential occurrences of simultaneous events can be detected, static analysis techniques for detecting points in the program at which these conditions are satisfied, and program instrumentation required to detect event occurrences at run-time. Concluding remarks are in section 4.

2 The Program Representation

A distributed program is represented as a **distributed control flow graph** (DCFG) which is an extension of the control flow graphs constructed for sequential programs. There are two types of nodes in a DCFG: nodes representing computational statements, such as an assignment statement, and nodes representing communication. The communication primitives used in the flow graph are the non-blocking send (S) and a blocking receive (R). There are also two kinds of edges in the flow graph. Intraprocess edges, which connect nodes belonging to the same process, represent the flow of control. Interprocess edges, which connect nodes from different processes, represent communication between processes.

Definition 2.1: The **distributed control flow graph** is denoted as DCFG=(V,E^{inter},E^{intra}) where V is the set of nodes in the graph, E^{intra} is the set of edges that connect nodes belonging to the same process, and E^{inter} is the set of edges that connect nodes from different processes.

To construct the DCFG we must match the sends and receives from various processes. The most conservative approach would be to match the send with all receives in the destination process. While this would yield correct results, it would adversely affect the quality of information that would result from static analysis techniques that will be used for detecting simultaneous events. Thus, it would be more appropriate to attempt the elimination of send-receive parings that are infeasible. Only the potentially feasible pairings can then be reflected in the DCFG. Let us consider a pair of processes PS and PR. To determine whether there should be an edge from a send S in PS to a receive R in PR we compute the following by statically analyzing the program: (i) $\#S$ which is the minimum number of sends from PS to PR that must be executed before control in PS reaches S; and (ii) $\#R$ which is the maximum number of receives to PR from PS that could have been executed before control in PR reaches R. If $\#S$ is greater than $\#R$ then we know that no edge is required from S to R since under no program execution can a message sent at S be received at R; otherwise an edge is introduced.

3 Detecting Simultaneous Events

A global simultaneous event consists of a number of local events from different processes composed together using the simultaneity operator. We assume that we are given a program which already contains local event evaluations. The key problem that is being addressed is the introduction of evaluations for simultaneous events. We establish the conditions under which simultaneous events can occur. Next we present static analysis techniques for identifying the program points at which these conditions are satisfied. Finally we describe the introduction of event evaluations and program instrumentation to perform these evaluations.

3.1 Conditions for Simultaneity

In order to identify the occurrence of a simultaneous event we must identify program points at which the occurrence and stability conditions hold. Static analysis is used to identify the points at which the component events can be *potentially true* and *guaranteed to remain stable*. In checking the occurrence condition the static analysis algorithms analyze interprocess communication to determine whether the information regarding component events can be communicated to a single process which can evaluate the simultaneous event. In checking the stability condition the static analysis algorithms analyze the activities at the components' processes and the nature of the interprocess communication in the computation to guarantee that the components' states will not change and therefore will be captured by a breakpoint.

Let us consider the detection of a simultaneous event SE=(LE_1==LE_2==...==LE_n), where each event LE_i is local to process P_i. Based upon its current state, each process P_i continually updates the status of its local event LE_i. The event may be detected at any process belonging to the distributed computation. Let us assume that the event is being detected at point p in process P. Further, without any loss in generality, let us assume that process P is distinct from processes P_1,P_2,...P_n. Fig. 1 shows the scenarios in which process P can detect of the simultaneous events based upon (i) event semantics; (ii) process interaction patterns; and (ii) both event semantics and interaction patterns.

First we consider the scenario in which the semantics of each local event LE_i guarantees that the event is **monotonic** [12], that is, once the event has occurred it always remains true. In Fig. 1a the interaction, referred to as the **pre-recognition interaction**, allows each process P_i to communicate the value of LE_i to process P. Once P recognizes an event occurrence it can initiate a breakpoint.

The second scenario shown in Fig. 1b considers the situation in which each local event in LE_i is **dependent monotonic** [12]. In this situation although the semantics of LE_i does not guarantee its stability, the semantics of the interactions among the processes guarantees event stability. In this scenario process P interacts with each process P_i twice. The pre-recognition interaction enables the communication of relevant information to process P. The second interaction, referred to as the **post-recognition interaction**, provides a means to ensure LE_i's stability. Process P evaluates the simultaneous event at point p which follows pre-recognition interactions with the processes and precedes the post-recognition interactions with the processes. As shown in Fig. 1b, following the pre-recognition interaction through S_i, process P_i is blocked at the post-recognition interaction R_i till process P executes R_i^P. Thus the state associated with LE_i remains unchanged if P initiates the breakpoint prior to the execution of R_i^P. Such interaction patterns can arise due to the usage of such features as remote procedure calls and rendezvous.

The third scenario shown in Fig. 1c is that of **generalized dependent monotonicity** that is achieved by exploiting event semantics. Here, although process P_i does not block immediately following the pre-recognition interaction at S_i, it must be guaranteed

that P_i will block at R_i unless S_i^P has been executed. Furthermore, the event semantics should guarantee that along all paths from S_i to R_i, the local state relevant to the evaluation of LE_i remains unchanged.

To determine the points, such as point p in process P, static analysis of the distributed program is carried out. It should be noted that the static analysis required for exploiting generalized dependent monotonicity is the superset of analysis required for the other two situations. The following conditions must be satisfied for a point p in process P to be a valid candidate for the evaluation of a simultaneous event.

1. Potential Pre-Recognition Interactions: This condition is required for all three kinds of events discussed above. There is a possibility of interaction between P and each of the processes' P_1, P_2,... and P_n as shown below. In these interactions, $\rightarrow$ represents an interprocess communication edge and $\rightsquigarrow$ represents an intraprocess path. These interactions are *potential* interactions, that is, there is a possibility that the execution of S_i will eventually lead to the execution of R_i^P in accordance with the specified interactions. No process is visited more than once during any pre-recognition interaction.

$$S_1 \rightarrow R_{11} \rightsquigarrow S_{12} \cdots \rightarrow R_1^P$$
$$S_2 \rightarrow R_{21} \rightsquigarrow S_{22} \cdots \rightarrow R_2^P$$
$$\cdots$$
$$S_n \rightarrow R_{n1} \rightsquigarrow S_{n2} \cdots \rightarrow R_n^P$$

2. Guaranteed Post-Recognition Interactions: This condition is required for generalized dependent monotonic and dependent monotonic events. There is a possibility of interaction between P and each of the processes' P_1, P_2, ... and P_n as shown below. These interactions are *guaranteed* interactions, that is, the execution of S_i^P will definitely lead to the execution of R_i in accordance with the specified interactions. No process is visited more than once during any post-recognition interaction.

$$S_1^P \rightarrow R_{11} \rightsquigarrow S_{12} \cdots \rightarrow R_1$$
$$S_2^P \rightarrow R_{21} \rightsquigarrow S_{22} \cdots \rightarrow R_2$$
$$\cdots$$
$$S_n^P \rightarrow R_{n1} \rightsquigarrow S_{n2} \cdots \rightarrow R_n$$

3. Stability of Local Events: For monotonic events and dependent monotonic events the event stability is guaranteed. For generalized dependent monotonic events, for each process P_i it must be guaranteed that there is no change to LE_i along all paths from S_i to R_i.

4. Valid Point of Recognition: There is a point p in process P such that: (i) there exists a path from the start of process P to point p which passes through R_1^P, R_2^P, ... and R_n^P; and (ii) after reaching p process P is guaranteed to execute S_1^P, S_2^P, ... and S_n^P. The second condition is only required for generalized dependent monotonic and dependent monotonic events.

The above conditions guarantee that if a valid point of recognition is found then the simultaneous event can be evaluated at this point by communicating the relevant information to process P through the pre-recognition interactions. Furthermore, once process P reaches the point of recognition, it is guaranteed to initiate the post-recognition interactions and the local events remain stable until the post-recognition interactions. Thus, the event state will be captured by the breakpoint.

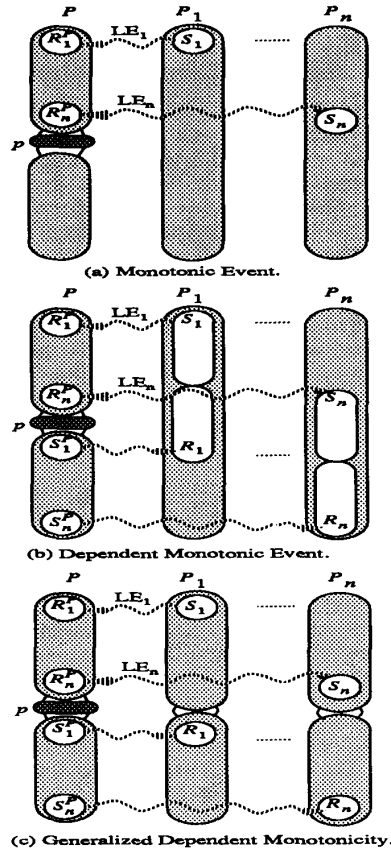


Figure 1: Simultaneous Event Detection.

3.2 Static Communication Analysis

In order to identify the points at which event evaluations should be introduced, we must collect information regarding communication relationships among the processes as well as control flow relationships local to each of the processes. This information will enable us to identify the program points which satisfy the conditions described in the preceding section. We assume that for each statement node n we are provided with sets $Succ(n)$ containing all immediate intraprocess successors of n, $Pred(n)$ containing all immediate intraprocess predecessors of n, $Reach(n)$ containing all nodes reachable from n, $Dom(n)$ containing all nodes that lie on all paths to n from the start of the program, and $PDom(n)$ containing all nodes that lie along all

paths from n to the end of the program.

3.2.1 Potential Pre-Recognition Interactions

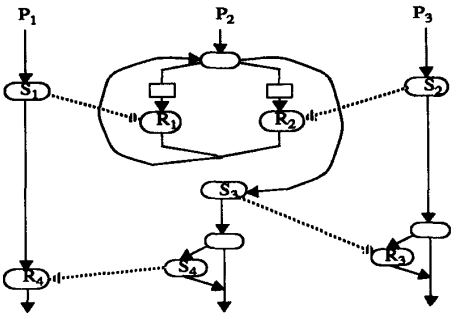
We compute all potential pre-recognition interactions among the processes by identifying, for each receive statement in a process, all those send statements in other processes such that there is a potential pre-recognition interaction between each of the send statements and the receive statement. There is a potential pre-recognition interaction between a send and a receive in another process if a chain of communication messages from the send to the receive can be found and this chain of messages visits each process at most once. The existence of such chains can be inferred from the information captured by the PRE set associated with each send and receive statement. The precise definition of the PRE set is as follows:

Definition 3.1: $\forall$ S and R
PRE(S)={S}$\cup${S':$\exists$ a potential interaction
$S' \rightarrow R_1 \rightsquigarrow S_2 \cdots \rightarrow R_n \rightsquigarrow S$}
PRE(R)={S':$\exists$ a potential interaction
$S' \rightarrow R_1 \rightsquigarrow S_2 \cdots \rightsquigarrow S_n \rightarrow R$}.

The system of equations that enables the computation of these sets for all relevant statements is given below. Since static analysis techniques can only approximate the run-time behavior of the program, one can at best obtain an overestimate or an underestimate of the true PRE sets. The approximation computed by the equations given below is an overestimate. An overestimation was chosen in this case to ensure that no potential pre-recognition interactions were excluded from consideration. For a receive R the set PRE(R) is computed by the first equation from the PRE sets of all the sends from which there is an interprocess edge to R. Since we are interested in interprocess interactions, the sends belonging to the process to which R belongs are excluded from PRE(R). The set PRE(S) of a send statement S is computed by unioning together the PRE sets of all receives from which there is an intraprocess path to S. The presence of an intraprocess path is determined using the reachability sets (*Reach*). Since the computation of *Reach* sets does not take into account interprocess communication, these sets are a conservative overestimate and hence the PRE sets computed also represent overestimates of the true PRE sets. The problem of computing reachable sets, taking into account interprocess communication, has been shown to be recursively undecidable.

Computation: $\forall$ R and S
PRE(R)=$\bigcup_{S \ni \exists S \rightarrow R}$ PRE(S)-{S':P(S')=P(R)}
PRE(S)=$\bigcup_{S \in Reach(R)} PRE(R) \cup \{S\}$

In accordance with the above equations, the PRE sets are computed by propagating the sends along intraprocess and interprocess edges in the DCFG. Interprocess propagation is carried out from a send in one process to the corresponding receive in another process. Intraprocess propagation is carried out from a receive to all sends reachable from that receive through intraprocess control flow edges. The algorithm for solving the above equations is as follows. A set LIST is maintained which contains all communication edges $S \rightarrow R$'s, such that there has been a change in set PRE(S). Edges are removed from the list one at a time and processed. First the set for the receive, that is set PRE(R), is updated. The change in PRE(R) is propagated to sends in the same process that are reachable from R. The edges leaving from these sends are added to the LIST for reexamination. Eventually when the PRE sets stablize no new edges are added to the LIST and the algorithm terminates. The above approach of computing the PRE sets can be viewed as consisting of two phases: intraprocess analysis and interprocess analysis. The computation of *Reach* sets is based upon intraprocess analysis. The computation of PRE sets utilizes the *Reach* sets to perform interprocess analysis. This approach allows efficient implementation of the interprocess analysis. The PRE sets for a sample DCFG are shown in Fig. 2.



N	PRE(N)
S_1	{ S_1 }
R_1	{ S_1 }
S_2	{ S_2 }
R_2	{ S_2 }
S_3	{ S_1, S_2, S_3 }
R_3	{ S_1, S_2, S_3 }
S_4	{ S_1, S_2, S_4 }
R_4	{ S_2, S_4 }

Figure 2: PRE sets for a sample DCFG.

3.2.2 Guaranteed Post-Recognition Interactions

A guaranteed interaction between a send statement and a receive statement is one in which the execution of the send guarantees the execution of the receive or the execution of the receive statement guarantees the execution of the send statement. This information will be captured by the POST sets defined below. For a send statement S, its POST set contains some send statements (POST1) and some receive statements (POST2). A send S' belongs to POST1(S) if the execution of S' will lead to the execution of S. A receive R' belongs to POST2(S) if the execution of

R' is guaranteed to be preceded by the execution of S. The POST set for a receive is defined similarly.

Definition 3.2: ∀ S and R
POST(S) = POST1(S) ∪ POST2(S)
$=\{S\} \cup \{S'$: executing S' $\Rightarrow S' \rightarrow R_1 \rightsquigarrow S_2 \cdots \rightsquigarrow S\}$
$\cup \{R'$: executing R' $\Rightarrow S \rightarrow R_1 \rightsquigarrow S_2 \cdots \rightarrow R'\}$
POST(R) = POST1(R) ∪ POST2(R)
$=\{R\} \cup \{S'$: executing S' $\Rightarrow S' \rightarrow R_1 \rightsquigarrow S_2 \cdots \rightarrow R\}$
$\cup \{R'$: executing R' $\Rightarrow R \rightsquigarrow S_1 \rightarrow R_2 \cdots \rightarrow R'\}$

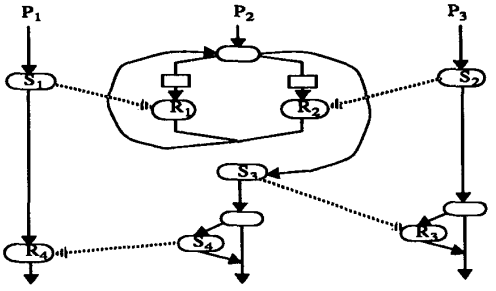
Next we present data flow equations for computing an approximation for the above sets. Since we are interested in guaranteed interactions, our approximation will be an underestimate of the true set of guaranteed interactions. This is essential to ensure that if a guaranteed interaction is used to predict the stability of an event then under no circumstance will we falsely reach this conclusion. The set POST1 is computed by forward propagation of send statements. For a receive statement R, the POST1 set includes elements of POST1(S) if the only outgoing interprocess edge from S has R as its destination. A send statement S contains the elements of POST1(R) if S and R belong to the same process and S postdominates R, that is, following the execution of S we are guaranteed that R will be executed. By propagating the information using the postdominator relationships and only along interprocess edges that belong to send statements that have a unique out-going edge ($S \rightarrow uniqR$), we are able to identify guaranteed post-recognition interactions. The set POST2 is computed in a similar fashion; however, it requires backward propagation of receive statements. The information is propagated using the dominator relationships and only along interprocess edges that belong to receive statements which have a unique in-coming edge ($uniqS \rightarrow R$). The algorithms for solving the following equations are similar to the algorithm described in the preceding section.

Computation:
Forward approximation: ∀ R and S
$POST1(R) = \bigcup_{S \ni S \rightarrow uniqR}$ POST1(S)-{S':P(S')=P(R)}
$POST1(S) = \bigcup_{S \in Pdom(R)}$ POST1(R) ⋃ {S}
Backward approximation: ∀ S and R
$POST2(S) = \bigcup_{R \ni uniqS \rightarrow R}$ POST2(R)-{R':P(R')=P(S)}
$POST2(R) = \bigcup_{R \in Dom(S)}$ POST2(S) ⋃ {R}

The POST sets of a sample DCFG are shown in Fig. 3. This example illustrates that the computation of POST1 and POST2 does not detect the same guaranteed interactions. Since S_1 is in the set POST1(R_3) we have a guaranteed interaction $S_1 \rightarrow R_1 \rightsquigarrow S_3 \rightarrow R_3$. However, since R_3 does belong to POST2(S_1) the same interaction is not identified during backward analysis. Similarly situations can be constructed where the POST2 identifies interactions that are not identified through the computation of POST1.

3.2.3 Stability of Local Events

In order to determine the stability of local events we will need to determine whether the state affecting a



N	POST1 ∪ POST2 = POST (N)		
S_1	{ S_1 }	∪	{ R_1 }
R_1	{ S_1 }	∪	{ R_1 }
S_2	{ S_2 }	∪	{ R_2 }
R_2	{ S_2 }	∪	{ R_2 }
S_3	{ S_1, S_2 }	∪	{ R_3 }
R_3	{ S_1, S_2, S_3 }	∪	{ R_3 }
S_4	{ S_4 }	∪	{ R_4 }
R_4	{ S_4 }	∪	{ R_4 }

Figure 3: POST sets of a sample DCFG.

local event changes following the execution of a *source* statement and preceding the execution of a *destination* statement. In order to achieve this goal we must identify all program statements that appear along some path starting at *source* and ending at *destination*. The algorithm *StatsAlongPaths* computes the set of statements that lie along a path starting at statement *source* and terminating at statement *dest*. Once all the statements have been identified we can examine these statements to determine whether the statements affect a given local event. If the statements do not alter the state tested by the local event, the event is stable from source to destination.

3.2.4 Valid Recognition Points

In order to determine whether a point p in process P is a valid recognition point we must find the associated pre-recognition and post-recognition interactions (see Fig. 1). Let R_1^P, R_2^P, ... and R_n^P be the receive statements in P corresponding to the pre-recognition interactions and S_1^P, S_2^P, ... and S_n^P be the send statements in P corresponding to the post-recognition interactions. As mentioned in section 4.1, two conditions must be satisfied: (i) there should be a path in process P that passes through R_1^P, R_2^P, ... and R_n^P before reaching p; and (ii) after reaching p process P is guaranteed to execute S_1^P, S_2^P, ... and S_n^P. This can be determined by checking the following conditions. The function *PathExists(p,list)* returns true if a path terminating at p that visits all nodes in list can be found.

(i) required for all three types of events
- PathExists(p,$\{R_1^P, R_2^P, \ldots, R_n^P\}$)

- $\forall\ R_i^P, S_i \in \text{PRE}(R_i^P)$

(ii) not required for monotonic events

- $\forall\ S_i^P, S_i^P \in \text{Reach(p)}$
- PathExists(stop,$\{S_1^P, S_2^P, \ldots S_n^P\}$)
- $\forall\ S_i^P$, $(S_i^P \in \text{POST}(R_i) \wedge R_i$ postdominates $S_i)$ $\vee (R_i \in \text{POST}(S_i^P) \wedge S_i^P$ postdominates p)

The algorithm *PathExists*, that determines whether there is a path in a process that visits $n_1, n_2, \ldots, n_m$ before reaching p, is implemented as follows. The list PATH contains all the statements for which a path has been found, that is, a path that visits PATH(1),PATH(2),...,PATH(|PATH|) has been found. We initialize PATH to contain p and one by one introduce $n_1, n_2, \ldots, n_m$ in the PATH as shown in the algorithm below. If we are unable to insert a statement n_i in PATH, the algorithm terminates with failure, otherwise we continue to insert statements till all have been successfully processed. To determine whether a statement n can be inserted between PATH(l) and PATH(l+1) we use of the algorithm *StatsAlongPaths*.

3.2.5 Worst Case Time Complexity

We present complexity analysis for generalized dependent monotonic events since they are the most complex. Let S, R, and E denote the number of sends, the number of receives, and the number of interprocess edges connecting sends to receives, respectively. Furthermore, let P denote the number of processes and N denote the maximum number of statement nodes in any given process.

1. Computation of Potential Pre-Recognition Interactions: The propagation of a single send to all possible PRE sets will in the worst case require the examination of each interprocess edge. Since there are S send nodes and E interprocess edges we will at most require O(S×E) time.

2. Computation of Guaranteed Post-Recognition Interactions: Using the same reasoning as was used above we can also conclude that the computation of sets POST1 and POST2 each requires O(S×E) time.

3. Determining the Stability of Local Events: The algorithm *StatsAlongPaths* used in this task will perform the most work if we were to determine all statements along the paths from the start of a process to the end of the process since this would require examining the entire program. Furthermore, this task takes time which is at most $O(N^2)$. Once the statements have been found, checking for stability requires O(N) time. Furthermore, the above process may have to be carried out for each process. Thus, the overall worst case complexity of this step is $O(P \times N^2)$.

4. Checking the Validity of the Point of Recognition: The algorithm *PathExists* performs at most $O(PATH^2)$ executions of *StatsAlongPaths*. However, PATH is at most equal the number of processes in the computation. Thus, the worst case time complexity of algorithm *PathExists* is $O(P^2 \times N^2)$.

From the analysis described above we can see that each static analysis step is quite efficient. Furthermore, we must note that since this overhead is incurred during static analysis, no runtime cost is incurred.

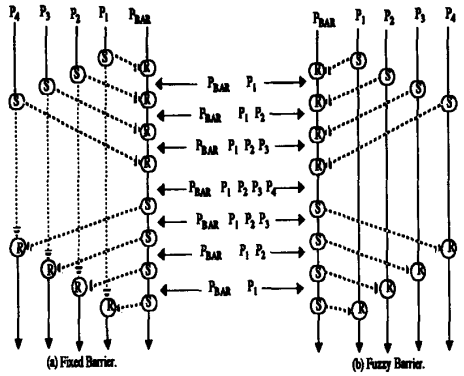


Figure 4: Simultaneous Event Detection by a Barrier Process.

3.2.6 An Example

Consider the example depicted in Fig. 4, in which process P_{BAR} implements barrier synchronization between processes P_1, P_2, P_3 and P_4. In Fig. 4a P_{BAR} implements a traditional fixed barrier in which each process after reaching the barrier at S waits at R till all processes arrive at the barrier. Our algorithm will detect that, as processes check into the barrier, the potential for detecting *dependent monotonic* simultaneous events among the processes exists. The set of processes among which dependent monotonic simultaneous events can be computed at the barrier process includes the processes which have checked in but have not checked out. Thus, dependent monotonic simultaneous events can be detected among increasingly larger sets of processes until, at the point at which the full synchronization is achieved, all processes are included in that set. In Fig. 4b P_{BAR} implements a fuzzy barrier in which each process can perform some useful computation while waiting at the barrier [5]. This computation is executed between S and R. In this situation our algorithm will detect that, as processes check into the barrier, the potential for detecting *generalized dependent monotonic* simultaneous events among the processes exists. To guarantee the stability of events local to the processes we must ensure that the computation executed between S and R does not alter the state relevant to the local event.

3.3 Program Instrumentation

There are three main steps in the instrumentation of the program to detect a simultaneous event: identification of program points at which event evaluations are to be introduced; construction of event evaluations to test appropriate state information and the instrumentation of communication edges to carry this information piggybacked to computation messages; and elimination of redundant evaluations of local events. In Fig. 5 we present the instrumented version of the

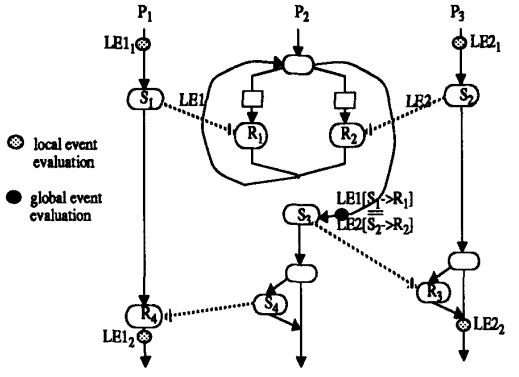


Figure 5: Program Instrumentation.

DCFG used in preceding section. We will illustrate the instrumentation process through this example.

Step 1: Consider any process P. First we identify whether there is a possibility that a simultaneous event may be evaluated by P. This is achieved by finding receives in the process which may be involved in pre-recognition interactions and then identifying sends that may participate in post-recognition interactions. If candidate events are found then we determine whether any of these points are valid recognition points using the techniques previously described.

Step 2: During the evaluation of an event $LE1{==}LE2$ in process P we must identify the appropriate values of $LE1$ and $LE2$ that are to be tested during the event evaluation. If the value of an event local to process P is being tested then the current value of that event is used. If the event value being used was computed at another process P', then we identify the send in P' and receive in P along which the value is communicated. For example, the event evaluation in Fig. 5 must use that value of $LE1$ which was sent from S_1 and received at R_1. In addition, all interprocess edges along which the value of $LE1$ or $LE2$ arrives at P should be instrumented to communicate the appropriate values.

if P uses LE received from P' in evaluating SE **then**
- associate with LE the pair $[S \rightarrow R] \ni$ the value of LE is sent out by P at S and received by P' at R.
- value of LE will be carried along $S' \rightarrow R'$ if $S' \in PRE(R) \wedge S \in PRE(S')$

Step 3: If an evaluation of a component event used in forming a simultaneous event does not reach any check for the simultaneous event, then this particular evaluation of the component event should be eliminated. For example in Fig. 5 the evaluations $LE1_2$ and $LE2_2$ are redundant since their results are not used in any evaluation of the simultaneous event $LE1 == LE2$.

4 Concluding Remarks

In this paper we introduced the simultaneity operator which is useful in formulating events that extend across multiple processes in a distributed computation. The utility of this operator in the debugging of a distributed computation was discussed. We presented static analysis techniques for detecting points at which event evaluations should be introduced. This approach is efficient because it does not cause the evaluation of a global event every time there is a change in a component of the event. Instead evaluations only occur if there is a potential that the evaluation would result in the detection of an event occurrence. We have also successfully applied the same approach for the efficient detection of concurrent and ordered events.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 1986.

[2] P.C. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *Proc. ACM Workshop on Par. and Dist. Debugging*, pp.11-22, 1988.

[3] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. Workshop on Par. and Dist. Debugging*, pp.140-150, 1991.

[4] J. Fowler and W. Zwaenepoal, "Causal Dist. Breakpoints," *Proc. 10th ICDCS*, pp.134-141, 1990.

[5] R. Gupta and M. Epstein, "High Speed Synchronization of Processors Using Fuzzy Barriers," *IJPP*, 19(1):53-73, 1990.

[6] R. Gupta and M. Spezialetti, "Towards a Non-Intrusive Approach for Monitoring Dist. Computations through Perturbation Analysis," *Proc. 6th Work. on Lang. and Compilers for Par. Comp.*, LNCS 768 Springer Verlag, pp.586-601, 1993.

[7] D. Haban and W. Weigel, "Global Events and Global Breakpoints in Dist. Systems," *Proc. 21st HICCS*, 1988.

[8] J. Joyce, G. Lomow, K. Slind, and B. Ungar, "Monitoring Dist. Systems," *ACM TOCS*, 5(2):121-150, 1987.

[9] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems," *CACM*, 21(7):558-565, 1978.

[10] B. Miller and J.D. Choi, "Breakpoints and Halting in Distributed Systems," *Proc. 8th ICDCS*, 1988.

[11] M. Spezialetti and J.P. Kearns, "Efficient Global Snapshots," *Proc. 6th ICDCS*, pp.382-388, 1986.

[12] M. Spezialetti and J.P. Kearns, "A General Approach to Recognizing Event Occurrences in Distributed Computations," *Proc. 8th ICDCS*, pp.300-307, 1988.

[13] M. Spezialetti and J.P. Kearns, "Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Systems," *Proc. 9th ICDCS*, pp.61-68, 1989.

[14] M. Spezialetti and J.P. Kearns, "A General Methodology for the System State Characterization of Event Recognitions," *Proc. 9th Symp. Reliable Dist. Systems*, pp.175-184, 1990.