

Observation and Control for Debugging Distributed Computations

Vijay K. Garg*

Parallel and Distributed Systems Laboratory,
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712.
<http://maple.ece.utexas.edu>

Abstract

I present a general framework for observing and controlling a distributed computation and its applications to distributed debugging. Algorithms for observation are useful in distributed debugging to stop a distributed program under certain undesirable global conditions. I present the main ideas required for developing efficient algorithms for observation. Algorithms for control are useful in debugging to restrict the behavior of the distributed program to suspicious executions. It is also useful when a programmer wants to test a distributed program under certain conditions. I present different models and their limitations for controlling distributed computations.

1 Introduction

Many problems in distributed systems, especially in distributed debugging, can be viewed as special cases of the problem of observing and controlling a distributed computation. For example, deadlock detection, termination detection, and breakpoint detection are some specific instances of the observation problem while problems of mutual exclusion, resource allocation, global synchronization and ordering of messages are special cases of the control problem. In this paper, I present a general framework for observing and controlling a distributed computation.

There are three fundamental defining characteristics of distributed systems - the lack of shared clock, the lack of shared state, and the presence of multiple processes. The lack of shared clock implies that the order of events in a distributed system can only be partial. The lack of shared state implies that computation of global functions must incur overhead of message passing. Finally, due to presence of multiple processes, there are exponential number of possible global states in a distributed computation and therefore we frequently face the problem of combinatorial explosion in analysis of distributed systems. These three characteristics make the design, analysis and debugging of distributed programs difficult.

What can we do to alleviate these problems? For observation of distributed computation, three ideas have been used in literature to effectively solve the above problems - causality, monotonicity and linearity. Causality is used instead of real-time in defining global states. Monotonicity is used as a restriction on the predicates so that at most one value is required to be communicated per external event of a process. Finally, linearity is used to avoid exploring all possible global states. Based on these ideas, efficient algorithms can be developed for observation.

*supported in part by the NSF Grants ECS-9414780, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

The objective of control in a distributed system is to maintain either an invariant on the global state or to ensure a proper order of events. The problem of control can be formulated under four settings depending upon whether it is online or offline, and whether the controller is allowed to change the order of events or simply introduce delay between events. Since the controller delays or disables events it is important that it does not introduce deadlocks in the system. When the control is online, the future computation is not known and avoiding deadlocks is impossible unless some assumptions are made. For example, it is impossible to maintain even a disjunctive predicate under online control. Similarly, any message ordering based on the online control must include at least all computations that are synchronously ordered. When the control is offline, determining a control strategy is possible but computationally intractable in general. By making restrictions on the control specification, the problem can be solved efficiently for many useful special cases. For example, there exists an efficient algorithm to maintain disjunctive invariants.

This paper is not a complete survey of algorithms on observation or control; rather it is a report on work done in the Parallel and Distributed Systems Laboratory (PDSLAB) at the University of Texas at Austin. I have sketched only the main ideas behind the algorithms. The reader is referred to [Gar96a] for a tutorial on distributed algorithms for observation and to [Gar96b] for a more detailed description of some of these algorithms.

The paper is organized as follows. Section 2 describes our model of distributed computation and global predicates. Section 3 discusses key problems and their solutions for observation of global properties. Section 4 discusses various models of control and reports some of our work for control. Finally, Section 5 describes applications of observation and control to distributed debugging.

2 Our Model

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of N processes denoted by $\{P_1, P_2, \dots, P_N\}$ communicating via asynchronous messages. In this paper, we will be concerned with a single run of a distributed program. We assume that no messages are altered or spuriously introduced. We do not make any assumptions about FIFO nature of the channels.

A *local state* is the value of all program variables and processor registers (including the program counter) for a single process. The execution of a process can be viewed as a sequence of local states. We use a causally precedes relation, ‘ \rightarrow ,’ between states similar to that of Lamport’s causally precedes relation between events [Lam78]. The causally precedes relation between two states s and t can be formally stated as: $s \rightarrow t$ iff s occurs before t in the same process, or the action following s is a send of a message and the action preceding t is a receive of that message, or there exists a state u such that s causally precedes u and u causally precedes t . We define a distributed computation as the partially ordered set (poset) consisting of the set of states together with the causally precedes relation.

Two states s and t are *concurrent* if s does not causally precede t and t does not causally precede s . A *cut* G is a collection of local states such that exactly one state $G[i]$ is included from each process P_i . A cut is called a *consistent cut* or a *global state* if all states are pairwise concurrent. Given two cuts, G and H , we say that $G \leq H$ iff $G[i] \leq H[i]$ for all i . The set of all global states form a lattice under this order [Mat89].

A *local predicate* is defined as any boolean-valued formula on a local state. For any process, P_i , a local predicate is written as l_i . A process can obviously detect a local predicate on its own.

3 Observation of Global Properties

Dear Watson, you see but you do not observe...

Consider a distributed debugging system. Suppose that our task is to implement the most basic command of a debugging system: “stop the program when the predicate q is true.” To stop the program, it is necessary to detect the predicate q ; a non-trivial task if q requires access to the global state. Each of the difficulties mentioned in the introduction needs to be addressed to develop algorithms for detecting q . We discuss each one of them next.

3.1 Lack of shared clock

Our first difficulty is defining the meaning of predicates in a global system. For a simple example, consider the predicate $q = CS_1 \wedge CS_2$ where CS_1 and CS_2 are predicates local on processes P_1 and P_2 respectively. One natural way of defining truthness of this predicate is that q is true in a computation if and only if there exists an instant of time in which both CS_1 and CS_2 are true. While this definition is adequate for sequential systems, it presents many problems in distributed systems. Since clocks are impossible to synchronize perfectly, it is impossible to ascertain whether two events happened simultaneously in a distributed system. This problem is solved by substituting causality for time (and therefore concurrency for simultaneity). Thus, we define q to be true if there exist two states s and t such that CS_1 is true in s , CS_2 is true in t and s and t are concurrent.

An advantage of the definition based on causality is that we can evaluate truthness of the predicate using vector clocks [Fid89, Mat89]. Further, for many applications such as distributed debugging this definition is more useful. For example, even if CS_1 and CS_2 are not true simultaneously, but there exists two concurrent states in which CS_1 and CS_2 are true, then there is an error in the program. This is because there exists a way of running the same program in which CS_1 and CS_2 are true simultaneously.

With the above observation in mind, the truthness of a general boolean predicate q is defined with two modalities. The predicate *possibly*: q is true if in the lattice of global states there is a path from the initial global state to the final global state in which q is true in some intermediate global state. The predicate *definitely*: q is true if q becomes true in all paths from the initial state to the final state in the lattice of global states. *Possibly*: q and *definitely*: q roughly correspond to weak and strong predicates in [GW92]. Possibly true predicates are useful for detecting bad conditions such as violation of mutual exclusion, whereas definitely true predicates are useful to verify occurrence of good predicates such as commit point on transaction systems. In this paper, we will discuss main ideas to develop efficient algorithms for detecting predicates of the form *possibly*: q .

3.2 Lack of shared memory

Our second difficulty stems from the fact that there is no shared memory. Thus, there is an inherent communication complexity for observing a global property of the system. For example, consider a global predicate $q(x_1, x_2)$ where x_1 and x_2 are variables in P_1 and P_2 respectively. If evaluation of this predicate at P_2 requires communication of all values of x_1 from P_1 to P_2 , and evaluation at P_1 requires communication of all values of x_2 from P_2 to P_1 , the function is clearly impractical to evaluate in a distributed system. In particular, if evaluation of a function requires communication of some value for every change in the value of a variable, then that function is impractical. To capture a class of functions which requires at most one value to be communicated per external event, we use the notion of monotonicity.

Informally, a predicate is monotone with respect to a variable x_1 if replacing the variable by a larger value while keeping all other variables the same preserves the truthness of the predicate. Formally, assume that x_1 takes its value from a set totally ordered with respect to a relation $<$. We say that q is monotone with respect to the first argument (x_1) if it satisfies the following equation:

$$\forall a, b, x_2 : (a < b) \Rightarrow (q(a, x_2) \Rightarrow q(b, x_2)).$$

For example, consider the predicate $q = (x_1 > x_2)$ where x_1 and x_2 are integers. Then q is monotone with respect to x_1 , because if $q(x_1, x_2)$ holds for a certain value of x_1 , then it would continue to do so for any larger value of x_1 . The predicate q is also monotone with respect to x_2 because for x_2 we can use the domain of integers with $<$ defined as the greater than relation defined on natural numbers. As another example, consider the conjunctive predicate $x_1 \wedge x_2 \dots \wedge x_n$ where each x_i is a boolean variable. By viewing the boolean domain as a totally ordered set with *false* to be defined as smaller than *true*, it can be easily seen that the conjunctive predicate is also monotone with respect to all variables. An example of a predicate that is not monotone with respect to either x_1 or x_2 is $(x_1 = x_2)$.

Monotonicity of a predicate allows us to restrict our attention to *state intervals* rather than states. A state interval is a sequence of states between two *external* events where an external event is the sending or receiving of a message, the beginning of the process and the termination of the process. For each state

interval it is sufficient to record and communicate the extremal value of the variable rather than all values taken by the variable in that interval. Since the number of state intervals is equal to the number of external events for a process, it is usually a much smaller number than the total number of events in the system. For example, if we are detecting $(x_1 > x_2)$ and x_1 takes values 2, 9, 4 and 7 between two external events, then it is sufficient to communicate the value 9. From now on we will use the term state and state interval interchangeably.

We note here that functions other than boolean can also be evaluated efficiently when they satisfy monotonicity. For example, [TG97b] describes an algorithm to compute $\max(x_1 + x_2)$ in a distributed system.

3.3 Combinatorial Explosion

Now assume that we have overcome limitations due to lack of shared clock and shared memory, for example, by using causality and monotonicity. This implies that we have the poset corresponding to the computation available at one process. So now, the problem is not even of distributed computing because all the required data is at one process. The problem we may face at this point is that of computation complexity. Since there are n processes, the total number of global states possible is m^n where m is the number of state intervals at any process. Consider a boolean predicate q . Even when q is a boolean expression, and processes do not communicate, the problem of detecting *possibly*: q is NP-complete [CG95].

Even though the problem is NP-complete for general boolean expressions, there exist efficient algorithms for several classes of q which occur in practice. One useful class is that of *linear* predicates which is based on the notion of *forbidden* state. Given a computation S , a predicate q , and a cut G in S , a state $G[i]$ is called forbidden if its inclusion in any cut H , where $G \leq H$, implies that q is false for H . That is,

$$\text{forbidden}(G, i) \stackrel{\text{def}}{=} \forall H : G \leq H : (G[i] = H[i]) \Rightarrow \neg q(H)$$

Based on the concept of a forbidden state, we define a predicate q to be linear with respect to poset S if for any cut G in the poset, the fact that q is false in G implies that G contains a forbidden state. Formally, a boolean predicate q is *linear* with respect to a poset S iff:

$$\forall G : \neg q(G) \Rightarrow \exists i : \text{forbidden}(G, i)$$

Observe that the linearity of a boolean predicate also depends on the poset S . We are typically interested in predicates which are linear for all posets consistent with a program. Observe that if q_1 and q_2 are linear, then so is $q_1 \wedge q_2$. Similarly, if q is defined using variables of a single process, then q is linear. It follows that a predicate q of the form $l_1 \wedge l_2 \wedge \dots \wedge l_n$ where each l_i is a local predicate is a linear predicate. If q is false in any cut, then one of the local predicates is false in some state s . The state s is forbidden because q cannot be true in any global state containing s .

There are other interesting examples of linear predicates, for example some *channel predicates*. We define a channel to be a uni-directional connection between two processes — one process performs all send events and the other all receive events. Channels have no memory. Hence, the state of a channel is the difference between the set of messages sent and the set of messages received. A *channel predicate* is any boolean function of the state of a channel. A channel predicate is linear by above definition if given any channel state in which the predicate is false, then either sending more messages is guaranteed to leave the predicate false, or receiving more messages is guaranteed to leave the predicate false. An example of a linear predicate is “channel C_{ij} is empty”. If this predicate is false, that is, the channel is not empty, then sending more messages is guaranteed to leave the predicate false.

As another example, consider the predicate $x + y \geq k$ where x and y are variables on processes P_1 and P_2 , and k is some constant. In general, this predicate is not linear. However, assume that x is known to be decreasing in the computation. In this case, $x + y \geq k$ is linear. Given any cut, if $x + y < k$, then we throw away the state with y variable. This is because that state combined with any future state for x variable can only have a smaller value for $x + y$.

Note that any global predicate, q , defines a (possibly empty) set of global states in which q is true. It is shown in [CG95] that q is linear with respect to a computation iff the set of global states in which q is

Characteristic	Problem	Idea	Bonus
No shared clock	ordering events	causality	avoid data race errors
No shared memory	message/state change	monotonicity	compute extremal functions
multiple processes	combinatorial explosion	linearity	get the first global state

Figure 1. Problems and their solutions for observation

true is an inf-semilattice. An implication of this result is that the *first* cut satisfying q is well defined iff q is linear.

3.4 Algorithms for Observation

In this section we show that above ideas can be used to derive a centralized algorithm to detect predicates of the form *possibly*: q where q is a conjunction of local predicates and linear channel predicates. We call such a predicate a General Conjunctive Predicate (GCP) [GCKM95].

The work of detection of the global predicate is divided among checker and non-checker processes. The non-checker processes are used in the computation and have local predicates and channels with predicates. The checker process is the process that determines if these predicates are true in the same global state.

The non-checker processes monitor local predicates. These processes also maintain information about the send and receive channel history for all channels incident to them. The non-checker processes send a message to the checker process whenever the local predicate becomes true for the first time since the last program message was sent or received. Since we use causality to define the semantics of truthness of q , we use vector clock instead of real-time clock to identify the instant the local predicate becomes true. Further, due to monotonicity of the predicate it is sufficient to send at most one message to the checker process per message sent or received. This message is called a local snapshot and is of the form: $(vector, incsend, increcv)$ where *vector* is the current vector timestamp while *incsend* and *increcv* are the list of messages sent to and received from other non-checker processes since the last message for predicate detection was sent.

The checker process is responsible for searching for a consistent cut that satisfies the GCP. Its pursuit of this cut can be most easily described as considering a sequence of candidate cuts. If the candidate cut either is not a consistent cut, or does not satisfy some term of the GCP (local predicate or a channel predicate), the checker can efficiently eliminate one of the states along the cut. This is due to linearity of a GCP predicate. The eliminated state can never be part of a consistent cut that satisfies the GCP. The checker can then advance the cut by considering the successor to one of the eliminated states on the cut. If the checker finds a cut for which no state can be eliminated, then that cut satisfies the GCP and the detection algorithm halts. This cut is the first cut for which GCP is true.

The above algorithm can be decentralized using techniques described in [GCKM94, GC95, HMRS95]. Other algorithms for predicate detection include stable predicates [CL85], relational predicates [TG97b], atomic sequences [HPR93], linked predicates [MC88], dynamic properties [BR95], general possibly and definitely predicates [CM91], regular patterns [FRGT94], general control flow patterns [GTFR95], conjunction of global predicates [SS95, GM96], event normal form predicates [CK94], recursive poset logic predicates [TG95] and strong conjunctive predicates [GW96].

We note here that the observation problem can either be solved in an on-line or in off-line setting. In an off-line algorithm you assume that the entire computation is given to you whereas the online algorithm is given only the past and must make observations while the computation is unfolding.

3.5 Open problems in observation

In this section, I present some useful problems in observation of distributed programs which are open to the best of my knowledge.

1. *Detecting exactly-k predicate:* Consider the predicate $x_1 + x_2 \dots + x_n = k$ where each x_i is a boolean variable on process P_i . Is there an efficient algorithm to detect this predicate? Observe that efficient

algorithms for $x_1 + x_2 \dots + x_n \geq k$ and $x_1 + x_2 \dots + x_n \leq k$ are known [Gar96b]. Also, if x_i can take any value then the problem is NP-complete since the set partition problem can be reduced to this problem.

2. *Detecting conjunction of 2-local predicates:* Consider the predicate $(P1.x < P2.y) \wedge (P3.x < P4.y) \wedge \dots$. Each of the conjunct in this predicate depends on at most two processes. If each conjunct can refer to any process, then the problem is known to be NP-complete [SS95]. However, if each process appears in the predicate at most once, then the reduction for [SS95] does not work. Is there a polynomial algorithm to detect a predicate of this form?
3. *Detecting 2-SAT predicates:* Consider a boolean predicate q in CNF form. If each clause has up to 3 literals, detecting q is NP-complete. If each clause has exactly one literal, then q can be detected efficiently using [GW94]. What is the complexity of detecting 2-SAT predicates?

4 Control of Distributed Predicates

Who controls the past controls the future, who controls the present controls the past...

George Orwell, Nineteen Eighty-Four.

We now go to the next natural step after observation - control. We propose the notion of a *supervisory process*. A supervisory process not only observes the underlying user process but also controls it by delaying (or disabling) some events or changing the order of messages in the user process. There are many reasons why we need the ability to control a distributed computation via supervisory processes.

Firstly, a supervisory process is essential for fault-tolerance. The current programming methodology views programming task as a simple execution of instructions. This execution may result in a fault which could have been avoided if critical events were verified for their suitability before execution. This is in contrast to human beings, who mix introspection with actual execution of a task. For example, if a human is using a recipe to cook some item, and comes across an instruction asking him to put his hand on fire, his common sense will dictate to him that he must not do so. Thus, a human being rarely follows an instruction blindly. He observes and controls the instruction he executes. In other words, every process has associated with it a *meta-process* which observes and controls the underlying process. Thus, a supervisor can be viewed as the meta-process which deals with events executed by the process itself. It may check integrity of data structures before or during any execution of critical events.

A supervisory process can also be viewed as an auxiliary process that monitors and adapts a program to varying external behavior. Supervisors have long been used for this purpose in control theory. A feedback supervisor can be used for tuning the parameters of the plant that may affect its behavior, or even switch from one policy to the other. For example, assume that procedure A and procedure B achieve the same result, but procedure A performs better than procedure B if the network is highly loaded, and vice-versa if it is lightly loaded. The underlying process may non-deterministically call both procedures, and the supervisor may enable the procedure which is more suitable for conditions at that point in time.

Last but not the least, the notion of supervision is also important in debugging and testing of a distributed program. Debugging or testing a distributed program is in essence search for anomalous behavior and identification of its cause. For this, the programmer needs to observe the program under some controlled environment. Why controlled environment? The programmer may suspect that the bug arises when the execution satisfies certain constraint (for example, when message m_1 is delivered before message m_2). Thus, the programmer is interested only in those executions which satisfy these constraints.

4.1 Different Models for Control

How does a supervisor exercise the control? Four possibilities are outlined in this section.

1. *Offline vs Online control:* We say that a supervisor exercises online control if it does not know about the future of the computation. Not knowing the future, the controller has only limited ability to meet the desired specifications. For example, we later show that it is impossible for an on-line controller to

meet disjunctive specifications without avoiding deadlocks. In the offline control model, we assume that the supervisor knows about the future. At first this model seems unrealistic, but this model has many applications. Consider distributed debugging. Assume that a computation was run in which the final results were unexpected. The programmer may want to run the same computation but now under the supervisor so that the computation goes through some controlled execution.

2. *Delaying events vs Changing order:* Here we address the issue of the power of the supervisor. A milder form of the supervision exists when the supervisor is allowed only to introduce the delay between events. Thus, the only difference between an uncontrolled and controlled computation is that there are fewer executions possible under control. The crucial problem here is introduction of deadlocks. The controller must ensure that no new deadlocks are introduced in the system by delaying of events.

A more powerful controller can decide to change the order of events. For example, the controller may change the order of messages received to meet the desired specification. Observe that both offline and online scenarios are possible. Under offline control, the controller may know that in the last run the message m_1 was received before m_2 . Therefore, in the next run it may deliver m_2 before m_1 possibly by delaying the receive of m_1 . This change of message ordering may be used for generating a different test case or during recovery from a software fault. Under online control, the controller may not know the future; but still be able to exercise some control. The simplest example of such control is imposition of the first in first out ordering on messages. By including sequence numbers, it is easy to control the message ordering so that the FIFO order is maintained.

We next discuss control under all four possibilities.

4.2 Delaying events: Offline control

The problem of offline control by delaying events can be posed as follows. We are given a poset S representing the computation. We are also given a boolean predicate q . The goal of the controller is to determine if there exists a way of delaying events in the poset S such that q is always true in the controlled execution. The task of delaying events can also be viewed as imposing an additional precedence relationship $\overset{C}{\sim}$ between events. Of course, the new relation should not interfere with existing causality relationship. In other words, on adding edges from the $\overset{C}{\sim}$ relation to the graph corresponding to the computation, the graph should remain acyclic. It can be shown that even if the computation is free from communication, and the boolean predicate is simply a boolean expression of local predicates, the problem of determining whether a control strategy exists is NP-complete [TG97a]. Considering that the problem of predicate detection for boolean expression is also NP-complete this is not surprising. The important issue here is whether there exists a useful class of predicates which can be controlled efficiently.

One such class of predicates is disjunctive predicates. A disjunctive predicate can be written as $l_1 \vee l_2 \vee \dots \vee l_n$ where each l_i is a local predicate. This can be viewed as avoiding a bad combination of states. One example is avoidance of deadlock in the classical dining philosophers problem. We can avoid deadlock by ensuring that at least one of the philosophers does not have any fork at all times. As another example consider availability of servers for critical tasks. We may impose a requirement on the system that at least one server is available at all times for critical tasks. One use of control in these applications would be rollback of the system under a fault and then its reexecution under control.

The algorithm for determining the strategy for disjunctive predicate control is based on the idea of overlapping intervals. Let I_1 and I_2 be two sequences of contiguous states such that local predicates l_1 and l_2 are false in I_1 and I_2 respectively. We say that I_1 and I_2 overlap if the lower end point of I_1 causally precedes the higher endpoint of I_2 and vice-versa. An important result is that a control strategy does not exist iff there exist intervals in which local predicates are false such that any pair of these intervals overlap. The proof of this result can be found in [TG97a]. In [TG97a], we also describe an efficient algorithm to add $\overset{C}{\sim}$ relation which does not interfere with the existing causal relationship and guarantees that the given disjunctive predicate is always true in the existing computation.

4.3 Delaying events - Online control

The problem of delaying events to guarantee a predicate is similar to the offline control except that the entire poset is not given to us. That is we assume that nothing about the future is known. For many applications this is the only realistic assumption. In this scenario, under the assumption that the processes are allowed to block for messages at any time it is impossible to control the system to maintain even a disjunctive predicate [TG97a]. The problem is that the deadlock is impossible to avoid while keeping an invariant if the future is not known. Given a choice of which process to delay to avoid falsifying the invariant the controller cannot make the right choice without knowing the future. For example, if the controller chooses to delay P_1 and not P_2 , it may turn out that P_2 waits for P_1 to send a message. At that point the progress cannot be made without violating the invariant. However, if the controller had chosen P_2 , there may be a valid computation which maintains the invariant. Thus, the controller can always be forced to make the wrong choice regarding which processes to delay for maintaining an invariant.

Therefore, we now assume that a process cannot block for a message while its local predicate is false. For example, in a 2-process mutual exclusion this would mean that a process cannot block in its critical section. Under this scenario, maintaining a disjunctive predicate requires that at least one process keeps its local predicate true at all times. If we call the section of the code in which the local predicate is false the *critical* section, then maintaining a disjunctive predicate defined on n processes is equivalent to $n - 1$ critical section problem. That is, at most $n - 1$ processes can be in the critical section at any time. A simple solution based on the concept of token can be used [TG97a]. The process which has the token is *not* allowed to enter its critical section. By our assumption any process that enters the critical section will eventually get out of it and is now a candidate for receiving the token. The details can be found in [TG97a].

In [TG94], we discuss maintaining global assertions which are in the sum of product form. Some examples of such assertions are $x_1 + x_2 + \dots + x_n \leq k$, and $x_1.x_2 + x_3.x_4 \geq k$ where x_i 's are in different processes.

4.4 Controlling Order - Offline Control

Assume that the programmer has run the computation once in which the final results are not correct. The programmer suspects that there is a bug in the program due to race of messages. That is, if a particular process receives and acts on message m_1 before another independent¹ message m_2 , then the results are faulty, otherwise not. In this case, she may want to reexecute the program under control so that m_2 is delivered before m_1 . In distributed debugging, if messages themselves are saved then one process can be fed those messages in any desired order. If message ordering is saved, then the entire computation can be run again but that process is delivered messages under the control of debugger. Kilgore and Chase [KC97] describe a Last-First Reordering algorithm that reexecutes a computation so that greatest number of message pairs are reversed in a single reexecution. Alternatively, the programmer may specify an order expression which tells the possible sequences of events that are allowed. For example, she may use regular expression, its generalizations such as concurrent regular expressions[GR92], path expressions[CH74], or dag expressions[GTFR95] to specify these sequences.

It is important to note here that once ordering of events has been changed, there is no guarantee that the rest of the computation will be the same as the last time.

4.5 Controlling Order - Online Control

We now assume that no information about the future of the computation is available. It is still possible to exercise some control. For example, if process P_1 sends k messages to P_2 , then these messages can be received in $k!$ ways each resulting in a different poset. Putting a FIFO order can be viewed as controlling the system so that some of these posets are not possible.

In this section we will focus only on external events; similar techniques can be applied to control ordering of other events. With each message we associate four events - invocation of the message, send of

¹By independent message we mean that the sending of message m_2 is causally unrelated to the sending of message m_1 .

the message, receive of the message and the delivery of the message. The invocation of the message takes place when user requests a message to be sent. The send of the message takes place when the controller informs the user that the message has been sent. Thus, the supervisor has the ability to delay the send of the message. The receive of the message takes place when the message reaches the destination process. The delivery of the message takes place when the message is actually delivered. Again, the delivery of the message can be delayed by the supervisor. By delaying the sends and the delivery of messages, the supervisor can change the order in which messages are sent and received. Observe that the send and delivery events can be delayed by the supervisor but invocation and receive events are uncontrollable. We will impose one more condition called *liveness* on our protocols. If the only events possible in a computation at certain points are send and delivery, then the protocol must enable at least one of the events. This restriction follows from our assumption of online control. Since the future is not known, the controller must always enable one of the current events for progress.

The following result in [MG97b] shows the limitations of the controllers based on above model. We call a computation *synchronously ordered* if all messages can be drawn vertically in its process-time diagram. We first argue that any computation that is synchronously ordered must be allowed by the protocol. Since the computation is synchronously ordered all the messages can be topologically sorted in the computation. Now we can run the computation in such a manner that exactly one event is enabled at every point before the completion of the run. That is, we invoke the first message. Since invocation is uncontrollable, this event is possible. Now since there is only one event enabled, namely the send event, the protocol must eventually execute the send event. This implies that the receive event is now possible due to uncontrollability condition. Finally, the delivery event would be enabled due to liveness condition. We now repeat this sequence with the second message. Thus, due to liveness condition and the fact that all messages can be sorted, we get that the synchronous computation is possible under the protocol.

One way for the programmer to specify desired message ordering is by using forbidden predicates [MG97b]. A message ordering is acceptable only if it does not satisfy the given forbidden predicate. A forbidden predicate is a conjunction of causality relationships between sends and receives of messages. For example, the following forbidden predicate specifies violation of causal ordering:

$$\exists x, y : (x.s \rightarrow y.s) \wedge (y.r \rightarrow x.r)$$

where $x.s$ denotes send of the message x and $x.r$ denotes receive of the message x . By associating colors and processes with messages, we can define most useful message orderings. One example where color is used is in the local forward flush channel [Ahu93] which requires that for any channel all messages sent before a red message are received before that red message. The forbidden predicate for this specification is

$$(process(x.s) = process(y.s)) \wedge (process(x.r) = process(y.r)) \wedge (color(x) = red) \wedge (x.s \rightarrow y.s) \wedge (y.r \rightarrow x.r)$$

In [MG97b] we show how a controller can take specification as a forbidden predicate decide whether that specification is implementable or not and if it is generate the protocol such that the predicate never becomes true in the computation.

As in the off-line case, another possibility for controlling order of events is based on the concept of an event expression. A primitive event is defined to be as execution reaching a predefined line number or a function. A complex event could be defined as a regular expression of primitive events or its generalizations as discussed before. The task of the controller is to ensure that the order of events generated belongs to the specification.

5 Applications to Distributed Debugging

In this section we describe the concept of observation and control that can be used for distributed debugging. We describe a distributed debugging system that is hypothetical; but all the functionality described can be implemented efficiently by known algorithms. We propose just one additional command to a distributed debugging system. The syntax of the command is quite simple:

do action when condition.

The *action* is taken whenever the specified *condition* becomes true. The detection of the condition, which could be global, corresponds to observation and the action corresponds to the control of the computation discussed in this paper. We first discuss various conditions and their meaning:

1. *boolean predicate q on the global state*: This corresponds to detecting *possibly:q*. Since the detection problem is NP-hard, we may require *q* to be linear which can be efficiently detected. Linearity also guarantees that the first global state satisfying *q* is well defined.
2. *regular expression r*: This corresponds to detecting a pattern in the computation. The regular expression is built out of local predicates. A regular expression (and its generalization - dag expression) can be efficiently detected by algorithms in [FRGT94, GTFR95].

The first type of condition is based on a single global state whereas the second type of condition is based on sequences of local states. There are many ways to combine and extend above conditions; we have kept our proposal simple.

Now we turn our attention to actions and their meaning. We propose the following types of actions.

1. *stop pids*: This command stops processes with given pids. The keyword *all* can be used to signify that all processes need to be stopped.
2. *print expr*: This command prints expression whenever the specified condition becomes true. If *expr* is null, then the intent is just to inform when the specified condition becomes true. Some information such as vector clock or line numbers are also printed to indicate when that condition became true.
3. *maintain bool*: The first two commands exercise trivial control whenever the specified condition becomes true. The *maintain* command is the first non-trivial example of control. The programmer specifies a condition which she wants to be maintained throughout the execution. This control could be exercised either in an on-line fashion or in an off-line fashion. Here we are assuming that there are two commands available to the programmer - **run** and **rerun**. The first command runs a new distributed computation whereas the latter runs the previous computation. It is assumed that the debugger saves ordering of messages (and other sources of non-determinism) so that a run is replayable. During the rerun command, the debugger has access to the future and can therefore exercise offline control strategy.
4. *maintain order-expression* This command controls the ordering of events during the computation. As in the previous command, this control could be online or offline. The order-expression could be specified in multiple ways. For message ordering, a forbidden predicate could be specified. For other events, expression such as a regular expression could be specified. The debugger could then enforce that order on execution of events. For example, consider the case when the programmer wants to enforce the ordering of two function calls *f* and *g* in different processes to be *f* followed by *g*. Then, she could specify:
maintain (exit f).(enter g)
where . corresponds to concatenation of two events. Based on this command the debugger would delay the process entering *g* until the execution of function *f* is finished.

6 Conclusions

Observation and control of a distributed computation is an useful abstraction for many fundamental problems in distributed systems. In this paper, we have presented difficulties and some solutions for observation and control. We have also shown some applications of this framework to distributed debugging. We have assumed a failure-free environment in this paper. An algorithm for observation under a faulty environment is given in [MG97a]. We have also kept our model for control simple. A more complex model may include notion of control variables (variables that are read by the underlying program but written by the controller) or notion of unobservable and uncontrollable events [RW89, KG95].

Acknowledgements

I would like to thank C. M. Chase, E. Fromentin, R. Kilgore, R. Kumar, J. R. Mitchell, V. V. Murty, M. T. Raghunath, M. Raynal, A. Tarafdar, A. I. Tomlinson, and B. Waldecker for collaborating with me on the work reported in this paper.

References

- [Ahu93] M. Ahuja. An implementation of f-channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):658–667, June 1993.
- [BR95] Ö. Babaoğlu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distributed Computing*, 28:173–185, 1995.
- [CG95] C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem. In *Proc. of the Workshop on Distributed Algorithms*, pages 303 – 317, France, September 1995.
- [CH74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science, Springer-Verlag*, 16, 1974.
- [CK94] H. Chiou and W. Korfhage. Efficient global event predicate detection. In *14th Intl. Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.
- [CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.
- [Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.
- [FRGT94] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc. of the 23rd Intl. Conf. on Parallel Processing*, pages 2:73 – 76, St. Charles, IL, August 1994.
- [Gar96a] V. K. Garg. Observation of global properties in distributed systems. In *Eighth IEEE International Conference on Software and Knowledge Engineering*, pages 418–425. IEEE, June 1996. revised version to appear in *IEEE Concurrency*.
- [Gar96b] V. K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publishers, Boston, MA, 1996.
- [GC95] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.
- [GCKM94] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting conjunctive channel predicates in a distributed programming environment. Technical Report TR-PDS-94-02, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1994.
- [GCKM95] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting conjunctive channel predicates in a distributed programming environment. In *Proc. of the International Conference on System Sciences*, volume 2, pages 232–241, Maui, Hawaii, January 1995.
- [GM96] V. K. Garg and J. R. Mitchell. Efficient detection of conjunctions of global predicates in a distributed system. Technical Report TR-PDS-96-005, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1996.
- [GR92] V. K. Garg and M. T. Raghunath. Concurrent regular expressions and their relationship to petri net languages. *Theoretical Computer Science*, 96:285–304, 1992.
- [GTFR95] V. K. Garg, A. I. Tomlinson, E. Fromentin, and M. Raynal. Expressing and detecting general control flow properties of distributed computations. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 432 – 438, San Antonio, TX, October 1995.

- [GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.
- [GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.
- [GW96] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.
- [HMRS95] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunction of local predicates. Technical Report 2731, IRISA, Rennes, France, November 1995.
- [HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).
- [KC97] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *Proc. of the International Conference on System Sciences*, Hawaii, January 1997.
- [KG95] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publisher, Norwell Massachusetts, 1995.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.
- [MC88] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, July 1988. IEEE.
- [MG97a] J. R. Mitchell and V. K. Garg. Detection of global predicates in a faulty environment. Technical Report TR-PDS-97-001, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1997.
- [MG97b] V. V. Murty and V. K. Garg. Characterization of message ordering specifications and protocols. In *Proc. of the International Conference on Distributed Computing Systems*, page to appear. IEEE, May 1997.
- [RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1), January 1989.
- [SS95] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th International Workshop on Distributed Algorithms*, pages 318–332, France, September 1995. Springer-Verlag.
- [TG94] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed systems. In N. Balakrishnan, T. Radhakrishnan, D. Sampath, and S. Sundaram, editors, *Proc. of the Intl. Conf. on Computer Systems and Education*, pages 257–272, New Delhi, June 1994. Tata McGraw-Hill.
- [TG95] A. I. Tomlinson and V. K. Garg. Observation of software for distributed systems with rcl. In *Proc. of 15th Conference on the Foundations of Software Technology & Theoretical Computer Science*. Springer Verlag, December 1995. Lecture Notes in Computer Science.
- [TG97a] Ashis Tarafdar and V. K. Garg. Predicate control in distributed systems. Technical Report TR-PDS-97-006, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1997.
- [TG97b] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal for Parallel and Distributed Computing*, 1997. a preliminary version appeared in Proc. of the ACM Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993, pp.21 – 31.