Predicate Control for Active Debugging of Distributed Programs

Ashis Tarafdar [*]
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
ashis@cs.utexas.edu

Vijay K. Garg [†]
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Abstract

Existing approaches to debugging distributed systems involve a cycle of passive observation followed by computation replaying. We propose predicate control as an active *approach to debugging such systems. The predicate control approach involves a cycle of observation followed by* controlled *replaying of computations, based on observation.*

We formalize the predicate control problem for both off-line and on-line scenarios. We prove that off-line predicate control for general boolean predicates is NP-hard. However, we provide an efficient solution for off-line predicate control for the class of disjunctive predicates. We further solve on-line predicate control for disjunctive predicates under certain restrictions on the system.

Lastly, we demonstrate how both off-line and on-line predicate control facilitate distributed debugging by allowing the programmer to control computations to maintain global safety properties.

1. Introduction

In distributed systems, a cycle of observation followed by replaying computations is the basis for locating bugs. The debugging process would be greatly facilitated if we add the ability to *actively* control the replayed computations based on the observations. This leads to a cycle of observation and *controlled* replaying which may, if used wisely, greatly accelerate the discovery of bugs. We focus on controlling a computation based on the specification of safety properties on global states. We call this the *predicate control* problem.

There are two important scenarios in which predicate control is useful for debugging distributed programs. Consider the cycle of observation and controlled replaying of computations. We first detect a global bug while observing a certain computation. Next, we try to replay the computation with some added control, to determine if it would be sufficient to eliminate the bug. This control is in the form of added causal dependencies to the existing trace of the computation. Since the computation is known *a priori*, we call this form of predicate control *off-line predicate control*. We may vary the added control and observe the resulting behavior of the computation to help us in localizing the bug. Once we have found a form of control that is effective for one computation, we may apply the same control to a new computation. Here we don't have *a priori* knowledge of the computation. Therefore, we call this control *on-line predicate control*. Section 7 illustrates the active debugging process with an example.

In this paper, we formally define off-line and on-line predicate control in an asynchronous message-passing system. We show that solving predicate control for safety properties based on general boolean predicates is NP-hard. However, for the useful class of disjunctive boolean predicates, we present an efficient algorithm for solving off-line predicate control, and an efficient algorithm for solving on-line predicate control under certain system restrictions. We then illustrate, with an example, the active debugging approach in which both forms of predicate control are applied.

2. Related Work

Our contributions are two-fold: a new active approach to distributed debugging, and a solution of the predicate control problem. We present related work in both areas.

Research in distributed debugging has focussed on two problems: detecting bugs in a distributed computation and replaying a traced distributed computation. Research in the detection of bugs mainly differs in the types of bugs specified. The seminal work in this area is the global snapshot algorithm [3] which is used to detect stable bugs (which re-

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

main true once they become true). Since then, detection algorithms have been designed for many different classes of bugs such as: race conditions [11], predicates on single global states [1], predicates based on sequences of global states [5]. Research in replaying trace computations have focussed on reducing the size of the trace by determining which events are necessary for successful replaying [9]. Our approach focusses on adding a control mechanism to the debugging process to allow computations to be run under safety constraints.

We are aware of two previous studies of controlling distributed systems to maintain classes of global predicates. One study [7] allows global properties within the class of conditional elementary restrictions [7]. Unlike our model of a distributed system, their model uses an off-line specification of pair-wise mutually exclusive states and does not use causality. [13] studies the on-line maintenance of a class of global predicates based on ensuring that a sum expression on local variables does not exceed a threshold. In contrast to these approaches, our focus will be on general global boolean predicates and the class of disjunctive predicates. We also study both the on-line and off-line variants of the control problem.

3. Model and Problem Specification

The distributed system consists of n sequential processes $P_1,\ P_2,\ \ldots,\ P_n$ which can send messages to one another over reliable channels. The system is asynchronous and has no shared memory. No constraints are placed on message ordering.

The *local execution* of P_i consists of a sequence of *states* and *events* starting at a special start state $\bot_i$ and ending at a special final state $\top_i$. An event takes the process from one state to another. An event may be a local event, a message send event, or a message receive event. A state corresponds to an assignment of values to all variables in the process.

For two states s and t in the same process, $s \prec_{im} t$ denotes that s *immediately precedes* t in the sequential execution of the process. $\prec$ (*precedes*) denotes the transitive closure of $\prec_{im}$. We say $s \leadsto t$ (s *remotely precedes* t) if the message sent in the event after s is received in the event before t. Given these relations, the *causally precedes* (happened before) relation [6], $\rightarrow$, is defined as the transitive closure of the union of $\prec_{im}$ and $\leadsto$. Note that $\rightarrow$ is an irreflexive partial-order over states in all processes. So, given any two states s and t, either $s \rightarrow t$ or $t \rightarrow s$ or neither causally precedes the other, denoted by $s \| t$ (s is *concurrent* with t). We use the notation $s \underline{\rightarrow} t$ to denote $s \rightarrow t \ \vee\ s = t$ and $s \ \preceq\ t$ to denote $s \prec t \ \vee\ s = t$.

Let S_i be the set of local states in the local execution of P_i and let $S = \bigcup_i S_i$ then a distributed computation can be modeled as a tuple $(S_1, \ldots, S_n, \leadsto, \prec)$. We call it a *deposet* (decomposed partially-ordered set) provided that $(S, \rightarrow)$ is an irreflexive partial order and it satisfies three reasonable constraints:

D1: No messages are received before the initial state,
D2: No messages are sent after the final state, and
D3: A single event does not both send and receive.

In a distributed computation modeled as a deposet, $(S_1, \ldots, S_n, \leadsto, \prec)$, we define a *global state* to be a subset of S containing exactly one state from each set S_i. Let $\mathcal{G}$ be the set of all global states in the deposet. We define an ordering relation $\leq$ on $\mathcal{G}$ as: For two global states $G, H \in \mathcal{G} : G \leq H$ iff $\forall i : G[i] \preceq H[i]$ where $G[i] \in S_i$ and $H[i] \in S_i$ are the states from P_i in global states G and H respectively. It has been established that $(\mathcal{G}, \leq)$ is a lattice [8].

A global state, G, is said to be *consistent* if $\forall x, y \in G : x \| y$. A consistent global state captures the notion of a global state that could possibly occur in the distributed computation. If $\mathcal{G}^c$ is the set of all consistent global states in the deposet, then $(\mathcal{G}^c, \leq)$ is also a lattice. It is easy to show using **D1** and **D2** that the initial global state $\bot = (\bot_1, \ldots, \bot_n)$ and the final global state $\top = (\top_1, \ldots, \top_n)$ are consistent.

An actual execution of a distributed system would take it from the initial consistent global state $\bot$ to the final consistent global state $\top$ through a sequence of consistent global states (a path in the lattice $(\mathcal{G}^c, \leq)$). We model the global execution as a *global sequence* – a sequence g of consistent global states ordered by $\leq$ such that restricting the sequence to any one process P_i produces the sequence S_i of states ordered by $\prec$ or the sequence S_i with some states consecutively repeated (called a *stutter* of S_i). Note that this does not enforce an interleaving of events since in a global sequence multiple local events can take place simultaneously.

The *control system* is a distinct distributed system whose processes are *controllers*, $C_1, \ldots, C_n$, which communicate using *control messages* on independent channels. A controller C_i monitors and controls process P_i by determining the next state and by occasionally blocking the process. Since the underlying process would not be able to distinguish between its controller's blocking action and a reduction of its execution speed, the control is transparent. The actions (monitoring and blocking) of the controllers are specified by a *distributed control strategy*.

On running a distributed control strategy for a control system, the resultant controlled distributed computation is in no way different from a computation of any other distributed system. We can, therefore, model a controlled distributed computation as a deposet. This deposet would include extra control states and control messages. If we restrict this deposet to the states of the underlying distributed system, we would have the deposet of the underlying system with added control causality between certain states. We use this as our model of a controlled distributed computation as follows.

Given a distributed computation modeled by a deposet $(S_1, \ldots, S_n, \rightsquigarrow, \prec)$ with a causal precedence $(S, \rightarrow)$, we define an *extended deposet* $(S_1, \ldots, S_n, \rightsquigarrow, \prec, \overset{C}{\rightsquigarrow})$ to consist of an extra control relation $\overset{C}{\rightsquigarrow}$ (for $x \overset{C}{\rightsquigarrow} y$, we say x *is forced before* y) between states. Each $\overset{C}{\rightsquigarrow}$ tuple is induced by a control message in the control system and relates the first underlying state before its send and the next underlying state after its receive. We then define an extended causal precedence $(S, \overset{C}{\rightarrow})$ to be the transitive closure of the unions of $\prec_{im}$, $\rightsquigarrow$ and $\overset{C}{\rightsquigarrow}$. The extended deposet would model a valid computation only if $(S, \overset{C}{\rightarrow})$ is an irreflexive, partial-order. However, it is possible to define a $\overset{C}{\rightsquigarrow}$ relation which causes cycles with $\rightarrow$ and results in a $\overset{C}{\rightarrow}$ that is not irreflexive. We say that such a $\overset{C}{\rightsquigarrow}$ relation *interferes* with $\rightarrow$.

Definition. Given a deposet, $(S_1, \ldots, S_n, \rightsquigarrow, \prec)$, with irreflexive partial-order $(S, \rightarrow)$ and a control relation $\overset{C}{\rightsquigarrow}$ which does not interfere with $\rightarrow$, the resultant extended deposet $(S_1, \ldots, S_n, \rightsquigarrow, \prec, \overset{C}{\rightsquigarrow})$ with irreflexive partial-order $(S, \overset{C}{\rightarrow})$ is called the *controlled deposet* of S with $\overset{C}{\rightsquigarrow}$.

It is easy to show that the set of global sequences in the controlled deposet is a subset of the set of global sequences in the original deposet. This is exactly the function expected of a control system.

A *local predicate* for process P_i is a boolean function of the variables associated with P_i. A *global predicate*, B, is an expression using boolean connectives $\neg, \vee, \wedge$ on local predicates. If B can be expressed as $l_1 \vee l_2 \vee \ldots l_n$ where l_i is a local predicate of P_i then B is a *disjunctive predicate*. $B(G)$ denotes the value of predicate B at the global state G. We assume that $B(G)$ can be efficiently computed.

G is said to *satisfy* B if $B(G) = true$. A global sequence g *satisfies* B if every global state in g satisfies B. Similarly, a deposet S *satisfies* B if every global sequence in S satisfies B. Lastly, a distributed control strategy A *satisfies* B if every possible deposet satisfies B. B is *infeasible* for deposet S if no global sequence in S satisfies B.

We can now formally define our problems:

The Off-line Predicate Control Problem
Given a global predicate B and deposet S for the underlying system, construct a distributed control strategy that satisfies B, unless B is infeasible for S.

The On-line Predicate Control Problem
Given a global predicate B and deposet S for the underlying system (provided on-line), construct a distributed control strategy that satisfies B, unless B is infeasible for S.

On-line predicate control is obviously a harder problem than off-line predicate control.

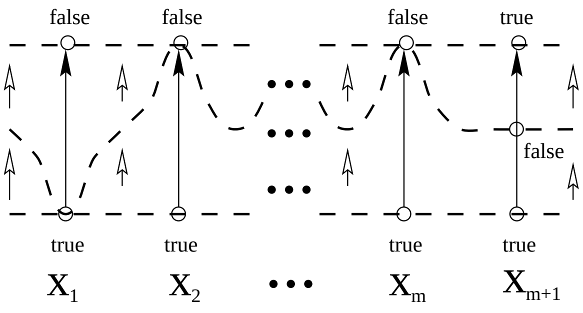

Figure 1. Proof: SGSD is NP-complete

4. Off-line Predicate Control is NP-hard

Consider the problem of detecting if a satisfying execution exists in a computation for a given predicate.

Satisfying Global Sequence Detection (SGSD): Given a deposet $(S_1, \ldots, S_n, \rightsquigarrow, \prec)$ and a global predicate B, determine if B is feasible for S (i.e. if there exists a global sequence in S that satisfies B).

Lemma 1 *SGSD is NP-complete.*

Proof: The problem is in NP because it takes polynomial time to check that a candidate global sequence is valid and that it satisfies B. To show that it is NP-hard, we map SAT to it. If b is the boolean expression in SAT, then for each variable, $x_1, \ldots, x_m$, in b we assign a separate process with two states, one $true$ and one $false$ (Figure 1). We define a process for an extra boolean variable x_{m+1} which starts $true$, goes through a $false$ state, and ends $true$ again. We define $B = b \vee x_{m+1}$ and then apply SGSD to find a satisfying global sequence. If it finds one, then the global state with $x_{m+1} = false$ will have a satisfying assignment for the variables of b. Conversely, if b is satisfiable, then there must be a satisfying global sequence. □

Given a satisfying control strategy, we can determine a satisfying global sequence by simulating a run of the strategy. Conversely, given a satisfying global sequence, we can construct a satisfying control strategy that would only allow that sequence as a possible run. Therefore, the problem of detecting if a satisfying control strategy exists is equivalent to SGSD. A detailed proof of this fact may be found in [12]. This equivalence indicates that:

Theorem 1 Off-line predicate control is NP-hard.

5. Off-line Disjunctive Predicate Control

Since predicate control is NP-hard in general, we restrict our attention to the class of disjunctive predicates. Intuitively, these predicates state that at least one local condition must be met at all times, or, in other words, that a bad combination of local conditions does not occur. Some examples of these predicates are:

(1) Two process mutual exclusion:
$$\neg cs_1 \ \vee \ \neg cs_2$$
(2) At least one server is available:
$$avail_1 \ \vee \ avail_2 \ \vee \ \ldots \ avail_n$$
(3) x must happen before y:
$$\mathit{after}\ x \ \vee \ \mathit{before}\ y$$
(4) At least one philosopher is thinking:
$$think_1 \ \vee \ think_2 \ \vee \ \ldots \ think_n$$

Note how we can even achieve the fine-grained control necessary to cause a specific event to happen before another as in property (3). This was done using local predicates to check if the event has happened yet.

Let the disjunctive predicate under consideration be $B = l_1 \vee \ldots \vee l_n$. Our approach to solving the problem consists of constructing a satisfying controlled deposet of S with $\overset{C}{\rightsquigarrow}$. A control strategy can be implemented from the controlled deposet by forcing the causal order of each tuple in $\overset{C}{\rightsquigarrow}$ by sending and receiving (with blocking) a control message on the concerned processes at the appropriate states.

The sequence determined by the total ordering $\prec$ of local states in S_i can be divided into intervals that are *true* or *false* with respect to local predicate l_i. We define a *false-interval* I as a maximal sequence of consecutive states in S_i for which l_i is false. $I.lo$ and $I.hi$ denote the beginning and ending states on I. To indicate that a false interval is on P_i, we use the notation I_i. We now define:

$$overlap(I_1, \ldots, I_n) \ \equiv \ \forall i,j \ : \ (I_i.lo{\rightarrow}I_j.hi) \ \vee \ (I_i.lo = \bot_i) \ \vee \ (I_j.hi = \top_j)$$

The significance of such an overlapping set of intervals is that in a valid global execution, no process can leave its interval until all the other processes are inside their intervals. So, if we have an overlapping set of false-intervals, then every global sequence must contain a global state in which B is false, and no satisfying control strategy exists. This is stated in the following result from [4]:

Lemma 2 *In a deposet,* $(S_1, \ldots, S_n, \rightsquigarrow, \prec)$*, with causal precedence* $(S, \rightarrow)$*:*
if $\exists I_1, \ldots, I_n \ : \ overlap(I_1, \ldots, I_n)$ *then there is no global sequence in* S *which satisfies* B*.*

Our algorithm makes use of this result by exiting with a *"No Controller Exists"* message when it detects an overlapping set of false intervals. If a set of false-intervals does not overlap then there must be a pair of intervals, I_i and I_j such that I_j can be crossed before I_i is entered. We define:

$$crossable(I_i, I_j) \ \equiv \ (I_i.lo{\not\rightarrow}I_j.hi) \wedge (I_i.lo \neq \bot_i) \wedge (I_j.hi \neq \top_j)$$

Algorithm Description

The algorithm is listed in Figure 2. The current global state g is advanced from $\bot$ forwards. For P_i we will only be interested in local states which are $\bot_i$, $\top_i$, the $I_i.lo$ or $I_i.hi$ for some false-interval I_i. Global state g will only consist of such local states. The predicates defined in the algorithm assume g as an implicit parameter. If $g[i]$ is at a $I_i.lo$ state, we say that P_i is *false*. Otherwise, we say that P_i is *true*. For P_i, $N(i)$ defines the next false-interval with respect to $g[i]$. $ValidPairs()$ is the set of pairs of processes for which the next false-interval of a process may be crossed while maintaining another process true. The local state $next(i)$ consists of the local state of interest after $g[i]$. So, for example, if $g[i]$ is $false$, $g[i]$ must be at an $I_i.lo$ state and $next(i)$ would be $N(i).hi$.

The algorithm takes as input the given trace of the distributed computation. This is implicitly used in evaluating the predicates used in the algorithm. The output is $\mathcal{C}$, a queue of tuples that defines the $\overset{C}{\rightsquigarrow}$ relation.

The central idea of the algorithm is to construct a chain of alternating true intervals and backward-pointing $\overset{C}{\rightsquigarrow}$ arrows. The chain extends from the $\bot_i$ state on some process P_i to the $\top_j$ state on some process P_j. Since any global state must intersect this chain, it must either be inconsistent (if it intersects at a backward pointing $\overset{C}{\rightsquigarrow}$ arrow), or it must satisfy the predicate (if it intersects at a true interval).

At L1, the algorithm enters a loop which exits when g has extended past the last false interval on some process, say P_j, indicating that the output chain has reached $\top_j$ and is complete. In each iteration of the loop, we will cross at least one false interval while some true interval maintains the responsibility of remaining true. Since there are a finite number of false intervals, the algorithm must terminate. We attempt to find such a pair of false and true intervals in L2 - L4. We are guaranteed to find an overlapping set of false intervals if no such pair can be found (for the proof of this fact, refer to [12]). So by Lemma 2, we can safely exit at L3 with "No Controller Exists".

Once we have found such a pair, we update our output chain by executing procedure $AddControl()$ in L14 - L18. Normally, a tuple, $g[k'] \overset{C}{\rightsquigarrow} next(k)$, will be output. Intuitively, this ensures that we do not exit the true interval used in the previous iteration, $(g[k], next(k))$, until the true interval in the current iteration, $(g[k'], next(k'))$, is entered. However, in the first iteration, and in any iteration in which the current true interval starts at $\bot_{k'}$, we may simply reinitialize the chain to $\emptyset$.

In L6 - L9, we cross the chosen false-interval $N(l)$ while advancing g on all the other processes to be causally consistent with this move. In other words, if a crossed state on P_l transitively depends on a state on some other process,

```
Variables:  $\mathcal{C}$      a queue of tuples of local states, initially $\emptyset$
                  and finally outputs the tuples in the $\overset{C}{\leadsto}$ relation.
            $g[1..n] = \bot$        current global state
            $k, k', l, i$           integers
            $t$                     local state

Predicates:
    $false(i) \equiv \exists I_i : g[i] = I_i.lo$
    $true(i) \equiv \neg false(i)$
    $N(i) \equiv \begin{cases} min\ I_i :: g[i] \preceq I_i.lo & \text{if it exists} \\ null & \text{otherwise} \end{cases}$
    $next(i) \equiv \begin{cases} \top_i & \text{if } N(i) = null \\ N(i).lo & \text{if } true(i) \\ N(i).hi & \text{if } false(i) \end{cases}$
    $ValidPairs() \equiv \{\langle i,j \rangle : true(i) \wedge crossable(N(i), N(j))\}$
    $select(Z) \equiv$ randomly selected element of set $Z \neq \emptyset$

L1   while ($\forall i : N(i) \neq null$) {                (* exit on reaching a $\top_i$ *)
L2     if ($ValidPairs() = \emptyset$)                     (* find a true interval which
L3       exit( "No Controller Exists" );                     can be maintained while
L4     $\langle k', l \rangle := select( ValidPairs() );$  a false interval is crossed *)
L5     $AddControl(\mathcal{C}, k', k);$
L6     $t := N(l).hi;$
L7     for ($i \in \{0, \ldots, n\}$)
L8       while ($next(i) \underline{\rightarrow} t$)      (* advance $g$
L9         $g[i] := next(i);$                                consistently with $\rightarrow$ *)
L10    $k := k';$                                          (* record last true interval *)
     }
L11  $k' := select( \{i | N(i) = null\} );$
L12  $AddControl(\mathcal{C}, k', k);$
L13  $exit(\mathcal{C});$

Procedures:
L14  $AddControl(\mathcal{C}, k', k)$ {
L15    if ($g[k'] = \bot_{k'}$)
L16      $\mathcal{C} := \emptyset;$                       (* start chain *)
L17    else if ($k \neq k'$)                              (* optimization *)
L18      $enqueue(\mathcal{C}, \langle g[k'], next(k) \rangle);$   (* add output tuple *)
     }
```

Figure 2. Algorithm: Off-line Control

then that state must also be crossed. Once this is done, we remember this iteration's true interval in the variable k (at L10) and repeat the loop.

Once we exit the loop, we output the last $\overset{C}{\leadsto}$ tuple to finish the chain and exit with the chain as output.

We refer the reader to [12] for the proof of correctness of this algorithm and we merely state:

Theorem 2 *The algorithm in Figure 2 terminates and correctly solves the off-line predicate control problem for disjunctive predicates.*

Evaluation

The time complexity of the algorithm is $O(n^2p)$ where p is the maximum number of false-intervals in a process. The naive implementation of the algorithm would be $O(n^3p)$ because the outer while loop iterates $O(np)$ times and calculating the set $ValidPairs()$ can take $O(n^2)$ time to check every pair of processes. However, an optimized implementation avoids redundant comparisons by computing the set $ValidPairs()$ dynamically. Since, in this approach, each new false-intervals has to be compared with $n-1$ existing false-intervals, the time complexity is $O(n^2p)$.

The size of $\overset{C}{\leadsto}$ is $O(np)$ because one tuple is output in each iteration of the outer while loop. Therefore, the message complexity of control messages used is $O(np)$. To give an idea of this quantity, in the two-process mutual exclusion example, there would be one message for each critical section, in the worst case (which is unlikely).

A good control strategy should also allow as much concurrency as possible. The ideal control strategy would only suppress the non-satisfying global sequences while allowing all satisfying ones. While this metric is hard to define, it is clear, for example, that a control strategy involving one-way, two-process synchronizations allows more concurrency than one involving multiple global synchronizations. Since each control message corresponds to a one-way, two-process synchronization (the receives are blocking), we have $O(np)$ such synchronizations.

6. On-line Disjunctive Predicate Control

Unfortunately, we find that the problem is impossible to solve for non-trivial (i.e. $n \geq 2$) disjunctive predicates. The proof for this [12] constructs a counter-example scenario that forces any control strategy to deadlock.

Theorem 3 *The On-Line Predicate Control Problem for non-trivial ($n \geq 2$) Disjunctive Predicates is impossible to solve.*

Note that even if we generalize on-line predicate control to allow each controller a finite lookahead of the underlying computation, the problem would still be impossible. Since it is impossible to solve the problem as it stands, we make the following assumptions:

A1: $\forall i : P_i$ does not block in states where l_i is *false* .

A2: $\forall i : l_i(\top_i)$

These assumptions allow us to assume that a *false* process will eventually become *true* without blocking.

Our control strategy is listed in Figure 3. At any time during execution, some process is the *scapegoat* and must remain *true* until it is sure that some other process is true and has assumed the role of *scapegoat* from him. The scapegoat simply requests any other process to take on the role. Since A1 and A2 ensure that it would eventually become *true* there will be no deadlocks. The details of the proof of correctness may be found in [12].

Theorem 4 *The distributed control strategy listed in Figure 3 terminates (does not deadlock) and correctly solves the on-line predicate control problem for disjunctive predicates.*

```
Distributed Control Strategy for Controller, $C_i$:

Input:
        $l_i$          a boolean function that takes a state as input
On-line Input:
        $s$            current state of the underlying computation
        $s'$           next state of the underlying computation
Variables:
        $scapegoat = init(i)$   boolean
        $pending = false$       boolean
        $j, k$                  integer
Control Actions:
    • $scapegoat \wedge \neg l_i(s')$:
        send($req(i)$, $select(\mathcal{C})$);
        receive($ack$);
        $scapegoat := false$;
    • $pending \wedge l_i(s)$:
        $pending := false$;
        $scapegoat := true$;
        send($ack$, $C_k$);
    • received($req(j)$):
        if $l_i(s)$ then
            $scapegoat := true$;
            send($ack$, $C_j$);
        else
            $pending := true$;
            $k := j$;
Definitions:
        $init(i)$      *true* for one $i$ and *false* for others
        $\mathcal{C}$  set of all controllers
        $select(Z)$    arbitrary element of non-empty set $Z$
```

Figure 3. Control Strategy: On-line Control

The k-mutual exclusion problem (a recent study may be found in [2]) is a generalization of the traditional mutual exclusion problem where at most k processes can be in the critical section at the same time. For $k = n - 1$, this specifies that at all times, at least one process must not be in the critical section. If we define the false-intervals to be critical sections, our problem becomes equivalent to $(n-1)$-mutual exclusion. Our distributed control strategy, therefore, also solves the $(n-1)$-mutual exclusion problem.

Evaluation

We follow the general guidelines in [10] for evaluating mutual-exclusion algorithms. Since only the critical sections of the scapegoat cause any overhead and the remaining critical section entries do not, we measure the overhead over n critical section entries. Let T be the average message propagation delay and E_{max} be the maximum critical section execution time. *Response time* is the time delay between a request for entering the critical section and the corresponding entry. *Per n critical section entries*, 2 messages are required and response time is bounded between $2T$ and $2T + E_{max}$, depending on when the request arrives. We have the option of reducing the response time at the expense of message overhead. We can devise a scheme where the scapegoat broadcasts a request to all controllers, and so has a better chance of finding at least one of them not in the critical section.

Our control strategy is simpler and more efficient than existing solutions to the k-mutual exclusion problem when specialized to the $k = n - 1$ case. While a complete comparison is beyond the scope of this presentation, the intuitive reason is that the k-mutual exclusion algorithms usually use k tokens while our algorithm uses a single *anti-token* which acts as a liability rather than a privilege. This indicates that for large k, a different class of algorithms may be more appropriate for the k-mutual exclusion problem.

7. Applications

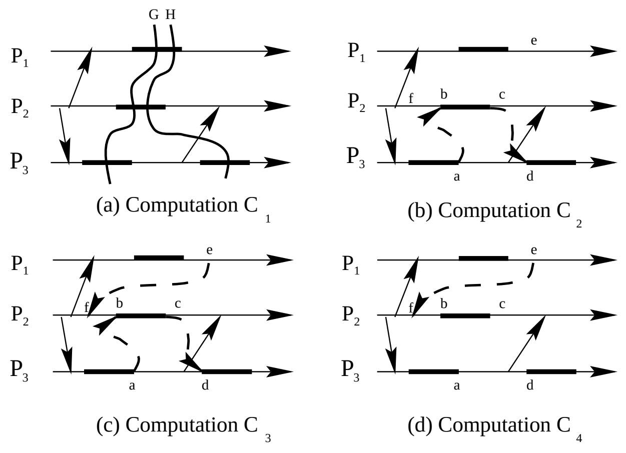

Figure 4. Example: Distributed Debugging

We have provided the ability to control a distributed computation both while being replayed and while being run for the first time. However, the utility of this ability depends on how effectively it can be used in the distributed debugging process. We now discuss applications of both off-line and on-line predicate control while debugging distributed programs.

Our running example will be a replicated server system with three server processes P_1, P_2 and P_3. During debugging, a trace of the distributed computation C_1 was taken as shown in Figure 4(a). The thicker intervals in the process executions indicate intervals when the servers weren't available for service.

The system should have been designed to ensure that one server was available at all times. So we run a predicate detection algorithm on C_1 (such as that in [4]) to detect bug_1: "all the servers are unavailable". We detect two consistent global states G and H, as shown in the diagram, where bug_1 is possible.

Our next step is to control C_1 with the safety predicate:"at least one server must be available at all times". Since this is a disjunctive predicate, we may use our off-line algorithm to control C_1, and the resulting computation C_2 is shown in Figure 4(b). Note how the control messages from a to b and from c to d ensure that global states G and H are no longer consistent and bug_1 doesn't occur.

We now suspect bug_2: "states f and e occur at the same time". We run the predicate detection algorithm in [4] to detect that bug_2 is indeed possible in C_2. We now impose the required safety predicate that "e must happen before f" and control C_2 using our off-line algorithm. The resulting computation C_3 in Figure 4(c) is found to be satisfactory.

However, we suspect that bug_2 may have caused bug_1. We, therefore, return to our first computation C_1 and apply off-line control to it with the safety predicate: "e must happen before f" This leads to computation C_4 in Figure 4(d). Note how the new control message from e to f ensures that G and H are inconsistent. So eliminating bug_2 also eliminates bug_1 and we conclude that bug_2 is the most important bug.

Now that we have discovered a possible bug in the system, we should check all future computations under the constraint that this bug does not occur. We, therefore, apply our on-line algorithm with the predicate: "e must happen before f" while running the system to generate new computations. If no more bugs are detected, our confidence that bug_2 is the problem increases.

In this illustration, we have demonstrated three areas where predicate control may be applied:

- Determining if a bug recurs under added safety constraints. (off-line)

- Determining the most important bug. (off-line)

- Preventing possible bugs in computations being run for the first time. (on-line)

8. Conclusions and Discussion

We have accomplished all of our goals from Section 1. While these relate to distributed debugging, it is important to note that predicate control is a problem of independent interest and that distributed debugging is just one of its main applications. Off-line predicate control would find applications wherever control is required when the computation is known *a priori*, such as in distributed recovery. On-line predicate control is widely applicable, addressing issues of general synchronization, of which important problems such as mutual exclusion form a part.

Given that general predicate control is NP-hard and that we can solve the simple class of disjunctive predicate control efficiently, the next step is to attempt to solve predicate control for more general classes of predicates. Towards this goal, we have recently solved both on-line and off-line predicate control for arbitrary predicates, under the restriction that the false-intervals of local predicates are mutually separated. These *locally independent global predicates* are a generalization of disjunctive predicates and allow us to express properties such as system-wide deadlock avoidance and more general forms of 2-process mutual exclusion. However, there are still many distributed synchronization problems such as general mutual-exclusion which have been solved as independent problems but have not been solved in the framework of predicate control. This is an indication that more classes of predicates should have predicate control solutions.

References

[1] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4. Addison-Wesley, 1993.

[2] S. Bulgannawar and N. H. Vaidya. A distributed k-mutual exclusion algorithm. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 153–160. IEEE, 1995.

[3] 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.

[4] 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.

[5] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 32–42. ACM/ONR, 1993.

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

[7] A. Maggiolo-Schettini, H. Wedde, and J. Winkowski. Modeling a solution for a control problem in distributed systems by restrictions. *Theoretical Computer Science*, 13(1):61–83, January 1981.

[8] 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.

[9] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[10] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18:94–101, 1993.

[11] K. Tai. Race analysis of traces of asynchronous message-passing programs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 261–268. IEEE, 1997.

[12] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. Technical Report ECE-PDS-1998-002, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin, 1998. available at http://maple.ece.utexas.edu as technical report TR-PDS-1998-002.

[13] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed sytems. In *Computer Systems and Education*, pages 257–272. Tata McGraw-Hill Publishing Company Limited, 1994.