Predicate Control for Active Debugging of Distributed Programs

Ashis Tarafdar and Vijay K. Garg

TR-PDS-1998-002 January 1998

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Predicate Control for Active Debugging of Distributed Programs

Ashis Tarafdar *
ashis@cs.utexas.edu
Dept. of Computer Sciences

Vijay K. Garg †
garg@ece.utexas.edu
Dept. of Computer and Electrical Engg.

University of Texas at Austin, Austin, TX 78712

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

Debugging programs on distributed systems [1] is more difficult than on sequential sequential systems for two reasons. First, unlike in sequential systems where the bug is based on an observable local state, a bug in a distributed system is based on a global state that is not easy to observe. The second problem is that even after we have detected a bug, we may not be able to reproduce it. This is a result of non-determinism in the system, brought about by varying processor and channel speeds. Thus, unobservability of global states and irreproducibility of distributed computations are issues that need to be addressed while building a distributed debugging system. This has led to research in the detection of bugs [3, 1, 9, 4] and in the replaying of distributed computations [11, 16].

In such 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

*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

[1] Although, we focus on distributed systems, the discussions apply equally well to parallel systems based on asynchronous message-passing.

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 repeat this control mechanism to localize the bug further. We may also determine whether bugs depend on each other so that eliminating one bug would eliminate the other. After we have found a probable cause of a bug and determined a safety predicate which eliminates the bug in a computation, we must further check other computations. It is likely that we would redo the same work for every other computation. However, in order to prevent the known bug from occurring, we would have to control the system without *a priori* knowledge of the computation. This control is called *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.

In the remainder of this section, we informally describe the key ideas presented in the paper.

Model of a Control System. Our model of a distributed system is that of asynchronous processes communicating with messages. Control is achieved by superimposing a distinct distributed *control system* on the underlying distributed system as follows. A *controller* is assigned to each process. Controllers communicate with one another using *control messages*. The controller can monitor and control the execution of its local process by predicting its next state and by temporarily blocking it. The control mechanism is transparent because the underlying process would perceive the blocking action as a decrease in processor speed. Thus, as expected of a debugger, no extra code has to be added by the programmer to debug his programs. Since the underlying system has message passing and blocking of its own, the controller's blocking action may cause the system to deadlock. It is the controller's responsibility to ensure that this does not happen.

The Predicate Control Problem. In asynchronous distributed systems, we distinguish between a *computation* and an *execution*. This distinction does not exist for sequential systems. A computation specifies a partial-ordering of local states, which can be determined by the sequence of local states on each process, and the ordering of message receives. An execution is a sequence of global states of the system, determined by processor speeds and message delays. Running a distributed system results in many possible computations, each of which corresponds to multiple possible executions.

Our problem is to control the system to suppress bad executions which do not satisfy the required global predicate [2] . In general, the global predicate is any boolean expression on local atomic predicates that may be efficiently evaluated. We define the distributed control problem to

[2]Since only a global observer in real-time could distinguish between two executions of the same computation, it may be argued that we should not concern ourselves with executions at all. However, for systems which interact with the external world, there is an implicit global observer viewing these interactions of the system in real-time. This is the basis for much work in distributed debugging such as predicate detection, and also other areas such as distributed mutual exclusion.

be either off-line or on-line depending on whether the future computation of the underlying system is known *a priori* or not. In general, an on-line problem is harder to solve than the corresponding off-line problem.

Solving General Predicate Control. We first study the possibility of solving the predicate control problem in its full generality. We show that the decision problem corresponding to off-line predicate control is equivalent to finding whether a global execution satisfying the predicate exists in a distributed computation. We prove that this problem is NP-complete by transforming SAT to it. Therefore, the off-line predicate control problem is NP-hard in its full generality.

Disjunctive Predicates. Since the general problem is NP-hard, we restrict our study to a simple, yet useful, class of problems specified by disjunctive predicates: disjunctions of local predicates. Intuitively, these predicates state that at least one property must be satisfied at all times or, in other words, that a bad combination of events 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:	$after\ x \vee before\ y$
(4) At least one philosopher is thinking:	$think_1 \vee think_2 \vee \ldots think_n$
(5) $(n-1)$-mutual exclusion:	$\neg cs_1 \vee \neg cs_2 \vee \ldots \neg cs_n$

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.

Off-line Disjunctive Predicate Control. We solve off-line disjunctive predicate control by constructing a centralized algorithm which takes the predetermined underlying computation and produces a control strategy. The control stategy specifies causal dependencies to be added to the given computation producing a stricter computation that maintains the given disjunctive predicate. The algorithm also detects if no control strategy is possible. The algorithm is efficient in terms of time and message complexity and also places low concurrency constraints. A full analysis is presented in Section 5.

On-line Disjunctive Predicate Control. We show that it is impossible to solve the problem of on-line predicate control for non-trivial disjunctive predicates (and hence, for general predicates). This is a result of the controller's lack of clairvoyance. When a controller makes a decision, an adversary can always ensure that for the decision it makes the system would deadlock, whereas for the other choice there would be a valid computation. However, under certain system restrictions we provide a solution. We show that the on-line predicate control problem for disjunctive predicates is equivalent to the $(n-1)$-mutual exclusion problem. This is a special case of the general k-mutual exclusion problem which has been studied in [2, 8, 13, 17, 20]. We show that in the special case of $k = n-1$, it is possible to do better than applying the general k-mutual exclusion algorithms.

2 Related Work

Our main contributions are two-fold: we propose a new active approach to distributed debugging, and we define and solve the predicate control problem. We, therefore, present related work in both areas.

Distributed Debugging. 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 remain true once they become true). Since then, detection algorithms have been designed for many different classes of bugs such as: race conditions [4, 21], predicates on single global states [1, 6, 7], predicates based on sequences of global states [15, 9]. Research in replaying trace computations have focused on reducing the size of the trace by determining which events are necessary for successful replaying [11, 16]. Our approach focusses on adding a control mechanism to the debugging process to allow computations to be run under safety constraints.

Predicate Control. We are aware of two previous studies of controlling distributed systems to maintain classes of global predicates. One study [12] allows global properties within the class of conditional elementary restrictions [12]. 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. [18] and [22] study the on-line maintenance of a class of global predicates based on ensuring that a sum or sum-of-product 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

3.1 Model of a Distributed System. The distributed system consists of n sequential processes P_1, P_2, ..., 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.

3.2 Model of a Distributed Computation. The *local execution* of P_i consists of a sequence of *states* and *events* starting at a special start state $\perp_i$ 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 $s \rightsquigarrow t$ (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 [10], $\rightarrow$, is defined as the transitive closure of the union of $\prec_{im}$ and $\rightsquigarrow$. 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 sequence of local states in P_i and let $S = \bigcup_i S_i$ then a distributed computation can be modeled as a tuple $(S_1, \ldots, S_n, \rightsquigarrow)$. We call it a *deposet* (decomposed partially-ordered set) [5] 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 a message.

3.3 Global states and Consistency. In a distributed computation modeled as a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, we define a *global state* to be a subset of S containing exactly one state from each sequence 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 [14].

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 $\perp = (\perp_1, \ldots, \perp_n)$ and the final global state $\top = (\top_1, \ldots, \top_n)$ are consistent.

3.4 Global Execution. An actual execution of a distributed system would take it from the initial consistent global state $\perp$ 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 in P_i 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.

3.5 Model of a Control System. 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. In particular, a controller is capable of determining the next state of its process and is capable of temporarily blocking the process. A controller and its underlying process are run on the same physical processor and may be together viewed as a single process with a combined controlled local execution formed by scheduling the controller at a higher priority. A controller is not capable of altering the local computation in any other way. The underlying process would not be able to distinguish between its controller's blocking action and a reduction of its execution speed. Therefore, the control system is entirely transparent to the underlying system. The actions (monitoring and blocking) of the controllers are specified by a *distributed control strategy*.

3.6 Model of a Controlled Distributed Computation. 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 the deposet to states of the underlying distributed system, then we have a valid deposet of the underlying distributed system except for extra causality between its states induced by the extra control messages. If we remove this extra causality, we would have the deposet that would have occurred if the control system had not existed (since the control system is transparent to the underlying system). Instead of modeling the controlled computation as a deposet including control states and messages, it is convenient to think of it as an extension to a deposet of an underlying computation with added control causality between its states.

Given a distributed computation modeled by a deposet $(S_1, \ldots, S_n, \rightsquigarrow)$ with a causal precedence $(S, \rightarrow)$, we define an *extended deposet* $(S_1, \ldots, S_n, \rightsquigarrow, \overset{C}{\rightsquigarrow})$ 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 $\rightarrow$.

Since, an extended deposet is formed by restricting an actual deposet of the controlled distributed system, and since, given an extended deposet, it is easy to construct an actual deposet (by adding necessary control states and control messages), from here on we do not distinguish between

an extended deposet and an actual deposet.

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

3.7 Global Predicates. Given a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, let X_i be the set of variables associated with P_i so that each state $s \in S_i$ defines a value for each variable $x \in X_i$. A *local predicate* for P_i is a boolean function of the variables in X_i. A *global predicate*, B, is an expression using boolean connectives $\neg, \vee, \wedge$ on local predicates. We use $B(G)$ to denote the value of predicate B in the global state G. We assume that $B(G)$ can be efficiently computed. If $B(G) = true$, we say that G *satisfies* B. Further, if for a global sequence, g, every global state G in g satisfies global predicate B, then we say that g *satisfies* B. If for a deposet S, every possible global sequence satisfies global predicate B, then we say that S *satisfies* B. If for a deposet S, at least one global sequence satisfies global predicate B, then we say that B is *feasible* for S. If B is not feasible for S, we say it is *infeasible* for S. If for a distributed control strategy, A, every possible deposet satisfies global predicate B, then A *satisfies* B. 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 (a boolean-valued function of the variables in X_i) then B is a *disjunctive predicate*.

3.8 Problem Specifications.

The Off-line Predicate Control Problem
Given a global predicate B and a 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 a 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.

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 $(S_1, \ldots, S_n, \rightsquigarrow)$ 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 Satisfiability to it. If b is the boolean expression in Satisfiability, 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.

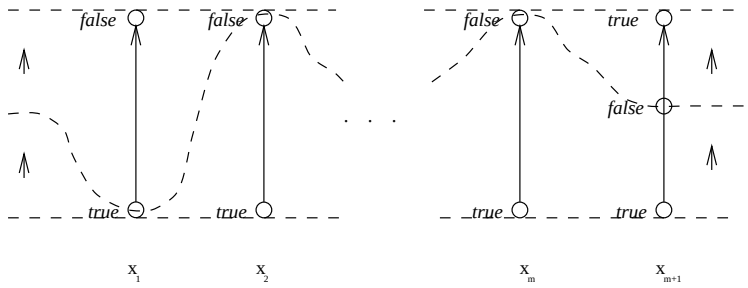

Figure 1: Proof: SGSD is NP-complete

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. □

The problem of detecting if a satisfying control strategy exists turns out to be equivalent to SGSD. The proof of this fact is straightforward and is presented in the appendix.

Satisfying Control Strategy Detection (SCSD): Given a deposet $(S_1, \ldots, S_n, \rightsquigarrow)$ and a global predicate B, determine if there exists a distributed control strategy that satisfies B for a control system whose underlying computation is the deposet S.

Lemma 2 *SCSD and SGSD are equivalent.*

Off-line predicate control requires finding a satisfying control strategy, if one exists. This is a search problem and SCSD is the corresponding decision problem. So from Lemma 1 and Lemma 2, we have:

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

5 Off-line Predicate Control for Disjunctive Predicates

The Off-line Predicate Control Problem for Disjunctive Predicates
Given a global predicate, $B = l_1 \vee \ldots \vee l_n$, and a deposet $(S_1, \ldots, S_n, \rightsquigarrow)$ for the underlying system, construct a distributed control strategy that satisfies B, unless B is infeasible for S.

Our approach to constructing the satisfying control strategy will be to construct 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 of local states S_i of P_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

$$overlap(I_1, \ldots, I_n) \equiv \forall i, j \in \{1, \ldots, n\} : (I_i.lo \rightarrow I_j.hi) \vee (I_i.lo = \bot_i) \vee (I_j.hi = \top_j)$$

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. This is shown in Figure 2 where dashed arrows indicate causal precedence ($\rightarrow$). So, if we have an overlapping set

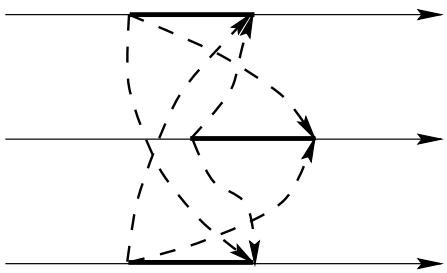

Figure 2: Overlapping intervals

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 [7]:

Lemma 3 *In a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, 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

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

The algorithm is listed in Figure 3. 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 global 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 a $I_i.lo$ state and $next(i)$ would be $N(i).hi$.

Algorithm Description. 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. This is illustrated in Figure 6 for the proof of Lemma 8 in the Appendix. 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 (this is because of an invariant INV 2 - refer to the proof of Lemma 4 for details). So using Lemma 3, 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

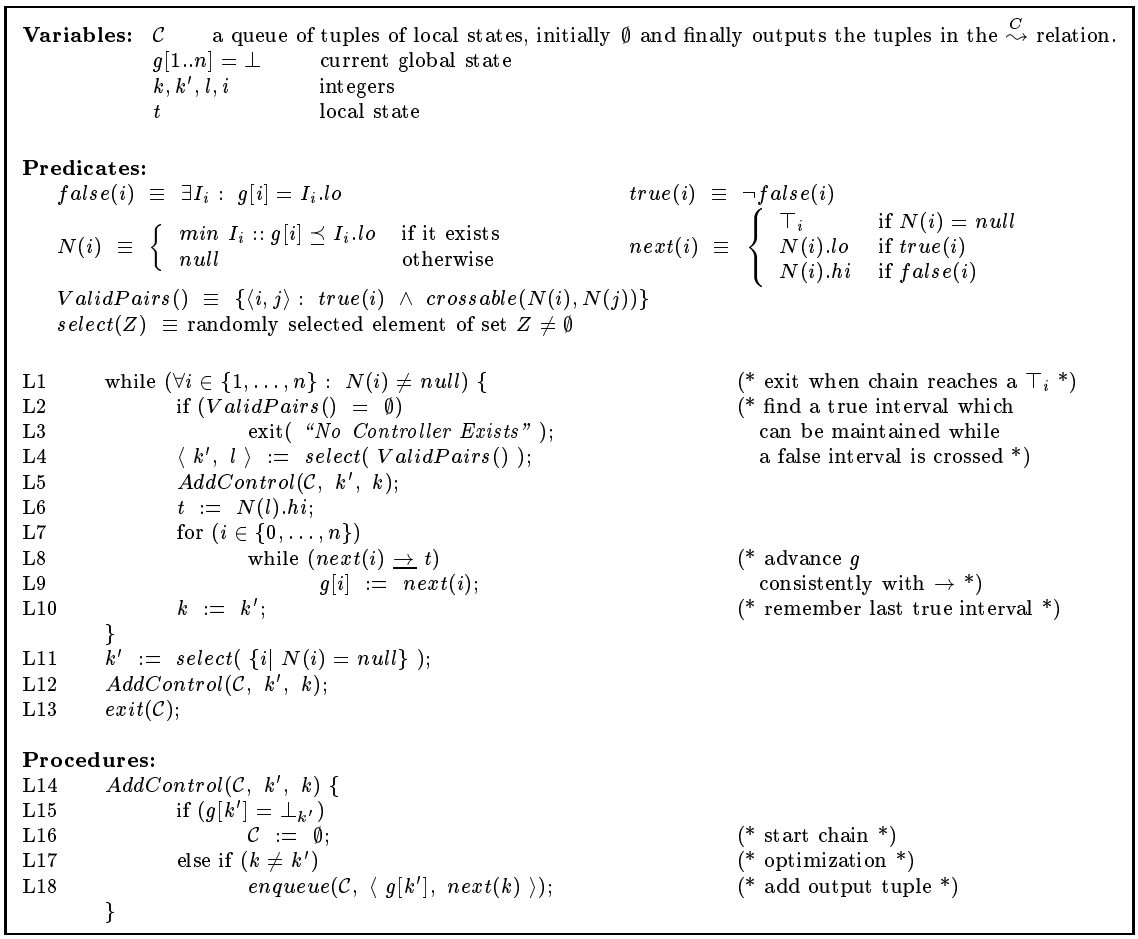

Figure 3: Algorithm for Off-line Predicate Control of Disjunctive Predicates

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, 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 $\overset{C}{\rightsquigarrow}$ tuple to finish the chain and exit with the chain as output.

Proof of Correctness. We list the steps in proving the correctness of the algorithm. The proofs may be found in the appendix.

The central property of our algorithm is that it maintains a global state g to satisfy two important properties specified in the following lemma:

Lemma 4 *The algorithm in Figure 3 maintains the following invariants:*

INV 1: $\forall i, j : next(i) \not\rightarrow g[j]$

INV 2: $\forall i : g[i] = \bot_i \vee \exists j : (g[i] \underline{\rightarrow} g[j] \wedge true(j))$

except while g is being updated (in loop L7 - L9*).*

Lemma 5 *The algorithm in Figure 3 terminates.*

Lemma 6 *If the algorithm in Figure 3 exits in L3 with "No Controller Exists", then B is infeasible for S.*

Lemma 7 *If the algorithm in Figure 3 exits in L13 with $\mathcal{C}$, then $\mathcal{C}$ defines a $\overset{C}{\rightsquigarrow}$ relation that does not interfere with $\rightarrow$.*

Lemma 8 *If the algorithm in Figure 3 exits in L13 with $\mathcal{C}$, then $\mathcal{C}$ defines a $\overset{C}{\rightsquigarrow}$ relation such that the controlled deposet of S with $\overset{C}{\rightsquigarrow}$ satisfies B.*

Lemmas 5, 6 and 8 together demonstrate the correctness of our algorithm:

Theorem 2 *The algorithm in Figure 3 terminates and correctly solves the off-line predicate control problem for disjunctive predicates.*

Complexity Analysis and 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 $\overset{C}{\rightsquigarrow}$ 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 Predicate Control for Disjunctive Predicates

The On-line Predicate Control Problem for Disjunctive Predicates
Given a global predicate, $B = l_1 \vee \ldots \vee l_n$, and a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, with causal precedence $(S, \rightarrow)$, for the underlying system (provided on-line), construct a distributed control strategy that satisfies B, unless B is infeasible for S.

Unfortunately, we find that this problem is impossible to solve for non-trivial (i.e. $n \geq 2$) disjunctive predicates. The proof for this hinges on the unpredictability of possible future causal dependencies making it impossible to prevent deadlocks in the system. The complete proof may be found in the appendix.

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

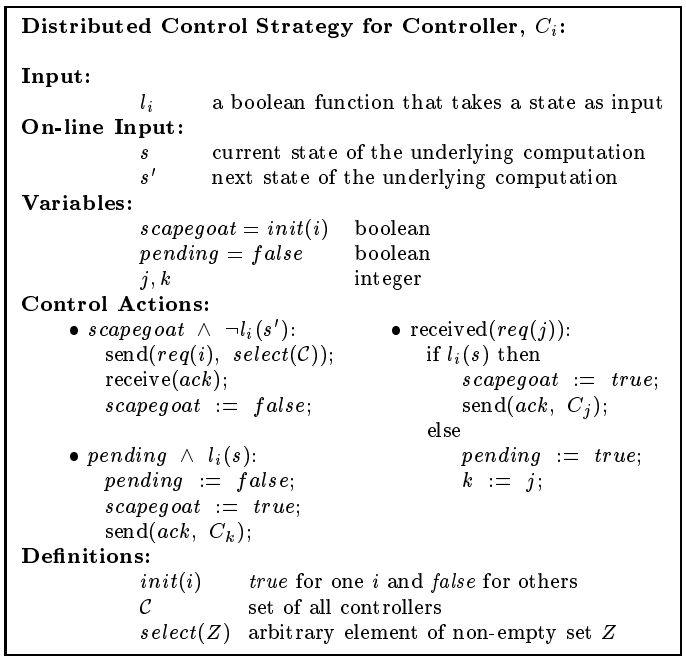

Figure 4: Distributed Control Strategy for On-line Predicate Control with Disjunctive Predicates

Note that even if we generalize on-line predicate control to allow each controller a finite lookahead of the underlying computation, we could design a counterexample to demonstrate impossibility. 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 4. 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 correctness of our strategy is established by the following theorem:

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

Proof: We first prove that deadlocks cannot occur, as follows. The only wait involved is when the scapegoat is waiting for an *ack*. This is guaranteed to arrive because every process in a *false* state will eventually reach a *true* state (by A1 and A2).

We next prove that the control strategy satisfies B. It is easy to prove by structural induction that in every possible global state the number of scapegoats in the system is strictly greater than the number of acknowledgment messages in the system. This indicates that there is at least one scapegoat in every possible global state. Since the algorithm ensures that a scapegoat is *true* , every possible global state satisfies B. □

The k-mutual exclusion problem [2, 8, 13, 17, 20] 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 and Comparison to Existing Solutions We follow the general guidelines in [19] 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 seen to be simpler and more efficient than existing solutions to the k-mutual exclusion problem [2, 8, 13, 17, 20] when specialized to the $k = n - 1$ case. Although a complete comparison is beyond the scope of this presentation, the intuitive reason for this is that the k-mutual exclusion algorithms usually use k tokens or wait for k replies and thus work well when k is small. 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

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 5(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 [7]) 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 5(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 [7] 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

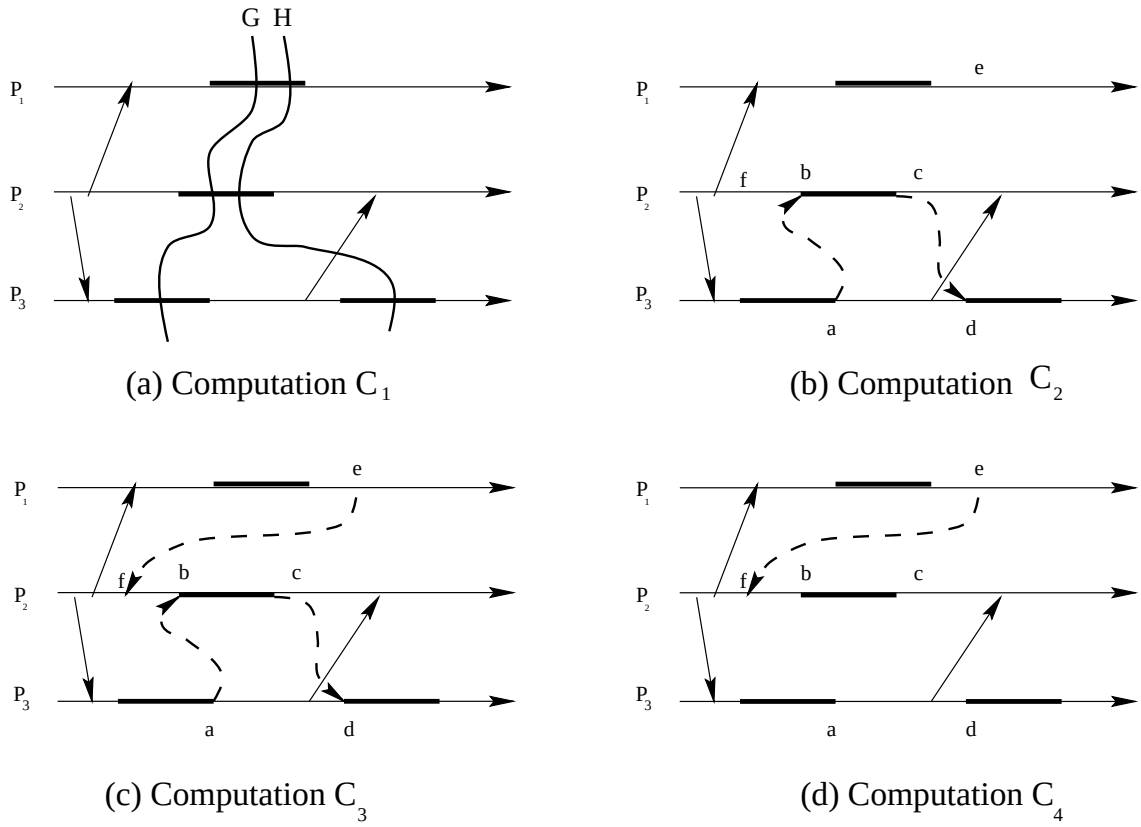

Figure 5: Example: Debugging a Distributed Program

computation C_3 in Figure 5(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 5(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 propose a new active approach to debugging distributed systems based on off-line and on-line predicate control. We demonstrate that the general off-line problem is NP-hard, that off-line disjunctive predicate control can be efficiently solved, and that on-line disjunctive predicate control can be efficiently solved under some restrictions. We also show that on-line disjunctive predicate

control is equivalent to $(n-1)$-mutual exclusion, and that our solution outperforms existing solutions to k-mutual exclusion applied to $k = n - 1$. We have also demonstrated how to apply off-line and on-line predicate control to facilitate the distributed debugging process.

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] R. Cypher and E. Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 534 – 541. IEEE, 1995.

[5] V. K. Garg. *Principles of Distributed Systems.* Kluwer Academic Publishers, 1996.

[6] 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.

[7] 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.

[8] S.-T. Huang, J.-R. Jiang, and Y.-C. Kuo. k-coteries for fault-tolerant k entries to a critical section. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 74 – 81. IEEE, 1993.

[9] 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.

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

[11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471 – 482, April 1987.

[12] 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.

[13] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park. A token based distributed k mutual exclusion algorithm. In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 408 – 411. IEEE, December 1992.

[14] 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.

[15] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316 – 323. IEEE, 1988.

[16] 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.

[17] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30:189 – 193, February 1989.

[18] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley and Sons Ltd., 1988.

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

[20] P. K. Srimani and R. L. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41:51 – 57, January 1992.

[21] 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.

[22] 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.

A Proofs

A.1 Equivalence of SCSD and SGSD

Satisfying Control Strategy Detection (SCSD): Given a deposet $(S_1, \ldots, S_n, \rightsquigarrow)$ and a global predicate B, determine if there exists a distributed control strategy that satisfies B for a control system whose underlying computation is the deposet S.

Lemma 2 *SCSD and SGSD are equivalent.*

Proof: If there is a satisfying control strategy, then we merely have to simulate a run on it to find a satisfying global sequence. If there is a satisfying global sequence g, then we construct an overly restrictive control strategy that allows only that global sequence and no other, as follows.

First, we construct a controlled deposet, S_c, for which the only global sequence is g. Let g be the sequence $G_1, \ldots, G_m$. For every distinct pair of global states G_k and G_l such that $k < l$, consider every pair of local states $G_k[i]$ and $G_l[j]$ of distinct processes P_i and P_j. If $G_k[i] \neq G_l[i]$ and $G_k[j] \neq G_l[j]$ and $G_k[i] \not\rightarrow G_l[j]$ then we impose $G_k[i] \overset{C}{\rightsquigarrow} G_l[j]$. It is clear that $\overset{C}{\rightsquigarrow}$ doesn't interfere with $\rightarrow$ and that every global state in g is consistent for $\overset{C}{\rightarrow}$. We can show that every global state H not in g is inconsistent for $\overset{C}{\rightarrow}$ as follows. H must consist of at least two local states which belong to distinct global states of g. By our construction, these local states must be connected by $\overset{C}{\rightsquigarrow}$ unless they are already connected by $\rightarrow$. Thus, the only global sequence allowed by S_c is g.

We can now construct a control strategy enforcing S_c as the only computation, as follows. For every $s \overset{C}{\rightsquigarrow} t$, where s is a state of P_i and t is a state of P_j, ensure that controller C_i sends a control message containing information (s, t) to P_j immediately after s. Immediately before t, controller C_j blocks P_j until the message (s, t) is received. It is easy to see that this ensures that S_c is the only possible computation. Therefore, the only valid global sequence obtained from this control strategy is g. □

A.2 Correctness of the Algorithm for Off-line Disjunctive Predicate Control

The central property of our algorithm is that it maintains a global state g to satisfy two important properties specified in the following lemma:

Lemma 4 *The algorithm in Figure 3 maintains the following invariants:*
INV 1: $\forall i, j : \; next(i) \not\rightarrow g[j]$
INV 2: $\forall i : \; g[i] = \bot_i \; \vee \; \exists j : \; (g[i] \underline{\rightarrow} g[j] \; \wedge \; true(j))$
except while g is being updated (in loop L7 - L9*).*

Proof: (induction on the number of occurrences of loop L7 - L9)

Initially: INV 1 holds because $g = \bot$ and D1 (in the definition of a deposet) tells us that no state can causally precede an initial state. INV 2 also holds because $g = \bot$.

After an update of g (in loop L7 - L9):

We first show by contradiction that INV 1 holds. Assume that $\exists i, j : \; next(i) \rightarrow g[j]$. We have two possibilities: either $g[j]$ has or has not been changed by this update of g. If $g[j]$ has not been changed, then by the inductive hypothesis of INV 1 and since *next* never decreases, we arrive at a contradiction. In the other case, if $g[j]$ has been changed, then loop L7 - L9 ensures that $g[j] \underline{\rightarrow} t$ while $next(i) \underline{\not\rightarrow} t$ which together contradict $next(i) \rightarrow g[j]$.

We now show that INV 2 holds as follows. Consider any $g[i]$. We have two possibile cases for P_i: either $g[i]$ has or has not been changed by the update of g. If $g[i]$ has been changed, then from loop L7 - L9 we have $g[i] \underline{\rightarrow} g[l]$ and from L6 we have $true(l)$. In the other case, if $g[i]$ has not changed, then by the inductive hypothesis of INV 2, either $g[i] = \perp_i$ or $\exists j : (g[i] \underline{\rightarrow} g'[j] \wedge true'(j))$. (where primed variables indicate the values before the update.) If $g[i] = \perp_i$, we are done. Consider the case of $\exists j : g[i] \underline{\rightarrow} g'[j] \wedge true'(j)$. Again for P_j, there are two possibilities: either $g[j]$ has or has not been changed by the update of g. If $g[j]$ has not been changed ($g[j] = g'[j]$), then we are done. Otherwise, if $g[j]$ has been changed, then from loop L7 - L9 we have $g[j] \underline{\rightarrow} g[l]$ and from L6 we have $true(l)$. Since $g[i] \underline{\rightarrow} g'[j]$ and $g'[j] \preceq g[j]$, we have $g[i] \underline{\rightarrow} g[l] \wedge true(l)$. □

Lemma 5 *The algorithm in Figure 3 terminates.*

Proof: We can show that the loop L8 - L9 terminates as follows. In every iteration, since $g[i]$ increases with respect to $\preceq$, so does $next(i)$ upto a maximum value of $\top_i$. By D2 (in the definition of a deposet), $\top_i$ does not causally precede any state. Finally, by the definition of crossable, we know that $t \neq \top_l$. So $\top_i \not\underline{\rightarrow} t$. So in a finite number of iterations we have $next(i) \not\underline{\rightarrow} t$ and the loop terminates.

We show that the loop L1 - L10 terminates as follows. In each iteration, for at least one process P_l, $g[l]$ advances (by L6 - L9 and the definition of N). So g advances with respect to $\leq$ in each iteration. Since the deposet has a finite number of states, we are guaranteed that in a finite number of iterations, some $N(i)$ would become *null* and the loop would terminate. □

Lemma 6 *If the algorithm in Figure 3 exits in L3 with "No Controller Exists", then B is infeasible for S.*

Proof: If the algorithm exits in L3, then when it exits, $ValidPairs() = \emptyset$. Consider two cases:

There is no pair $N(i)$, $N(j)$ such that $crossable(N(i), N(j))$: Then $overlap(N(1), \ldots, N(n))$ and by Lemma 3, B is infeasible for S.

There is a pair $N(i)$, $N(j)$ such that $crossable(N(i), N(j))$: We show by contradiction that this case cannot arise. Since $ValidPairs() = \emptyset$, we must have $false(i)$. By the definition of $false(i)$, we know that $g[i] = N(i).lo$. So by the definition of *crossable*, we have $g[i] \neq \perp_i$. So using INV 2 in Lemma 4, we have $\exists k : (g[i] \rightarrow g[k] \wedge true(k))$. By the definition of $true(k)$, we have $g[k] \prec N(k).lo$. Since $N(i).lo = g[i]$ and $g[i] \rightarrow g[k]$ and $g[k] \prec N(k).lo$, we have $N(i).lo \rightarrow N(k).lo$. So, by D1 we have $N(k).lo \neq \perp_k$. So, since $crossable(N(i), N(j))$ we have $crossable(N(k), N(j))$. Since $true(k)$, we find that $ValidPairs() \neq \emptyset$ which is a contradiction. □

Lemma 7 *If the algorithm in Figure 3 exits in L13 with $\mathcal{C}$, then $\mathcal{C}$ defines a $\overset{C}{\rightsquigarrow}$ relation that does not interfere with $\rightarrow$.*

Proof: (by contradiction)

Assume that $\mathcal{C}$ defines a $\overset{C}{\rightsquigarrow}$ relation that interferes with $\rightarrow$. So there must be a cycle in $\overset{C}{\rightarrow}$. Let $\langle g_1[k'_1], next_1(k_1) \rangle$ be the last $\overset{C}{\rightsquigarrow}$ tuple that was enqueued in $\mathcal{C}$ that belongs to such a cycle (it must exist because $\rightarrow$ has no cycles). So $g_1[k'_1] \overset{C}{\rightsquigarrow} next_1(k_1)$. Consider the cycle in $\overset{C}{\rightarrow}$ including this tuple. Let $g_2[k'_2] \overset{C}{\rightsquigarrow} next_2(k_2)$ be the next $\overset{C}{\rightsquigarrow}$ tuple in the cycle. (Note: this may be the same as the first tuple $g_1[k'_1] \overset{C}{\rightsquigarrow} next_1(k_1)$ if there is just one $\overset{C}{\rightsquigarrow}$ tuple in the cycle.) So $next_1(k_1) \rightarrow g_2[k'_2]$. Since $g_1[k'_1] \overset{C}{\rightsquigarrow} next_1(k_1)$ is the last $\overset{C}{\rightsquigarrow}$ tuple that belongs to a cycle, the algorithm's variable g

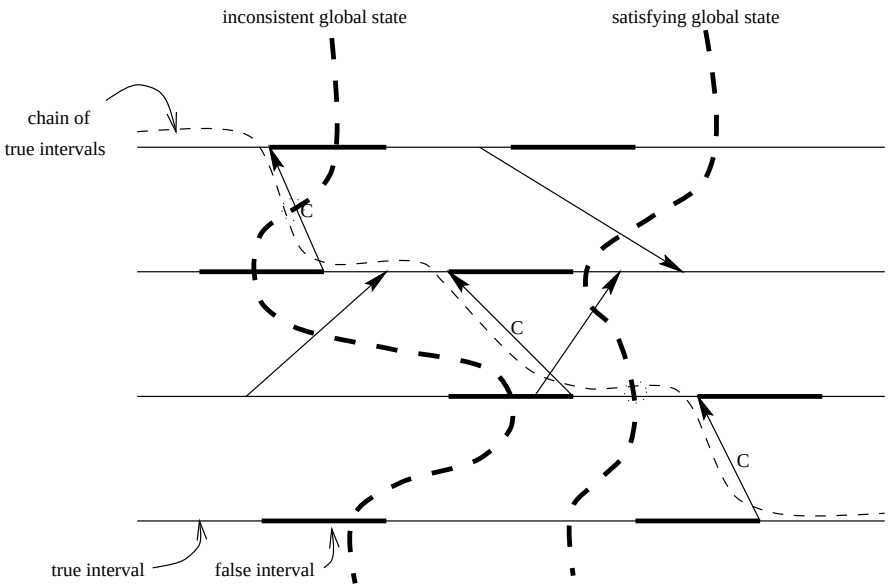

Figure 6: Proof: Lemma 8

takes on the value g_2 before it takes on the value g_1. So $g_2[k_2'] \preceq g_1[k_2']$. So $next_1(k_1) \rightarrow g_1[k_2']$. This contradicts INV 1 in Lemma 4. □

Lemma 8 *If the algorithm in Figure 3 exits in L13 with $\mathcal{C}$, then $\mathcal{C}$ defines a $\overset{C}{\rightsquigarrow}$ relation such that the controlled deposet of S with $\overset{C}{\rightsquigarrow}$ satisfies B.*

Proof: Since by Lemma 7, $\overset{C}{\rightsquigarrow}$ does not interfere with $\rightarrow$, the controlled deposet S_C of S with $\overset{C}{\rightsquigarrow}$ is well-defined.

Consider the point in the algorithm when a tuple $\langle g[k'], next(k)\rangle$ is added to $\mathcal{C}$ at L18. At this time, from L4 we have $true(k')$. Since in the previous iteration, the current k was k', and since $g[k']$ is not updated in the loop L8 - L9 (by the definition of *crossable*), we have $true(k)$. Since we have both $true(k)$ and $true(k')$, both intervals defined by $(g[k], next(k))$ [3] and $(g[k'], next(k'))$ are true-intervals. So the tuple $g[k'] \overset{C}{\rightsquigarrow} next(k)$ connects the base of a true-interval to the top of another true interval. It is easy to verify from the algorithm that the sequence of such tuples form a chain of true intervals from the start state on some process to the end state of some other process with backward pointing $\overset{C}{\rightsquigarrow}$ arrows (Figure 6). So any global state must intersect this chain. If the intersection occurs in a backward pointing $\overset{C}{\rightsquigarrow}$ arrow, then the global state is inconsistent with respect to $\overset{C}{\rightarrow}$. If the intersection occurs in a true-interval, then the global state satisfies B. So all global states that are consistent with respect to $\overset{C}{\rightarrow}$ satisfy B. So S_C satisfies B. □

Lemmas 5, 6 and 8 together demonstrate the correctness of our algorithm:

Theorem 2 *The algorithm in Figure 3 terminates and correctly solves the off-line predicate control problem for disjunctive predicates.*

[3] Note: We use the notation (a, b) to mean an interval which does not include its end-states a and b.

A.3 Impossibility of On-line Disjunctive Predicate Control

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

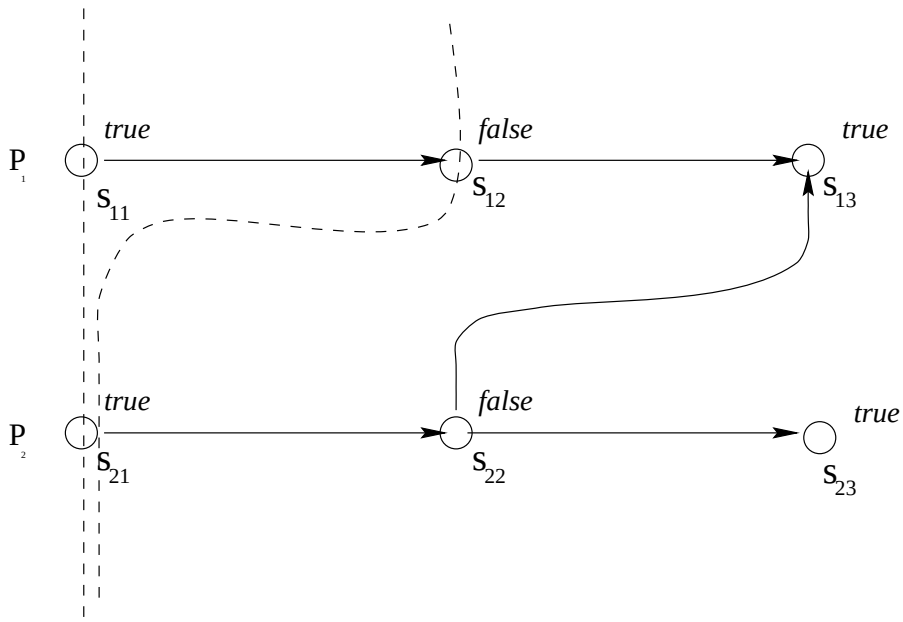

Figure 7: Impossibility of On-line Predicate Control

Proof: (by contradiction) For simplicity, we prove for the case $n = 2$. The proof is easily extensible to general $n > 2$. Consider a deposet S consisting of $S_1 = s_{11}^t, s_{12}^f, s_{13}^t$ and $S_2 = s_{21}^t, s_{22}^f, s_{23}^t$. Let $s_{22}^f \rightsquigarrow s_{13}^t$. Let $B = l_1 \vee l_2$ be the disjunctive predicate. The superscripts of the states indicate whether they are *true* or *false* with respect to the corresponding local predicate.

The control strategy starts at the global state (s_{11}^t, s_{21}^t) and is only aware of the next states but is ignorant of what comes after the second states. Since the situation is perfectly symmetrical, it has a choice of either of two possible next global states in the global sequence corresponding to which process advances. So it may choose to let P_1 advance so that the next global state is (s_{12}^f, s_{21}^t). (Note that (s_{12}^f, s_{22}^f) would cause B to be violated.) However, since $s_{22}^f \rightsquigarrow s_{13}^t$, P_1 has to wait at s_{12}^f for a message to be received from P_1 after state s_{22}^f. The system cannot advance to (s_{12}^f, s_{22}^f) because this would violate B. Hence, the system is deadlocked.

We now show that B was indeed feasible for S . This is demonstrated by the global sequence, $(s_{11}^t, s_{21}^t), (s_{11}^t, s_{22}^f), (s_{12}^f, s_{23}^t), (s_{13}^t, s_{23}^t)$, which satisfies B. $\square$