Contact Author: Ashis Tarafdar (ashis@cs.utexas.edu)
Tracks: Long Presentation (if not selected in the Long Presentation Track, please consider the abstract for the Brief Announcement Track)
Student Paper: Yes. Ashis Tarafdar is a full-time student

Predicate Control in Distributed Systems

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

Vijay K. Garg [†]
garg@ece.utexas.edu
Dept. of Computer and Electrical Engineering

University of Texas at Austin, Austin, TX 78712

Abstract

A number of important problems in asynchronous distributed systems can be formulated as special cases of the notion of controlling a distributed system to maintain global properties. We formalize this notion by defining the *predicate control problem* in terms of boolean global predicates and a model of distributed control. The problem arises in both off-line and on-line scenarios. We prove that general off-line predicate control is NP-Hard. However, we present an efficient solution for the class of disjunctive predicates. We show that on-line predicate control, on the other hand, is impossible to achieve even for disjunctive predicates. However, by placing restrictions on the underlying system, we are able to present an effective on-line control strategy.

1 Introduction

An intrinsic problem in asynchronous distributed systems is that while no one process can have a global view, we still require the system as a whole to maintain global properties. This conflict has led to the design of distributed algorithms which specify the manner in which processes cooperate to maintain a global property. We call problems which require distributed control of a global property *distributed control problems*. Generally, each type of global property has been studied as a separate distributed control problem. Some classic examples are: distributed mutual exclusion [17], distributed resource allocation [3], load balancing [2], and distributed consensus [6]. The safety aspects of each of these problems and many others could be expressed as some global property which must be maintained.

Our approach, *predicate control*, is to define a control mechanism for a distributed system, given a general global property. The global properties corresponding to the example problems stated above are complex. Hence, we might expect that such a generalized approach be limited in its ability to efficiently solve them. However, it is insightful to discover the limitations of the predicate control approach and to determine if there is a class of simple distributed control problems for which such an approach may provide an efficient solution.

Such a generalization attempt is not without precedent. Another important problem in distributed systems is *distributed detection* of global properties. It has been studied in specialized forms such as termination detection [4] and deadlock detection [10]. However, studies in *predicate detection* [7] have generalized the problem to the detection of arbitrary predicates, provided an understanding of its limitations, and defined classes of predicates for which an efficient general solution is possible.

We are aware of two previous studies of the general distributed control problem. One study [13] allows global properties within the class of conditional elementary restrictions [13]. 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 [21] study the on-line maintenance of a class of global properties based

[*]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

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 properties and the class of disjunctive predicates. We also study both the on-line and off-line variants of the control problem.

Our model of a distributed system assumes asynchronous processes communicating with messages. Control of a global property is achieved by superimposing a distinct distributed *control system* on the underlying distributed system. The control system is entirely transparent to the underlying system. It consists of controllers, one for each process, which communicate using independent control messages and which are capable of monitoring and controlling the underlying process. In particular, they can predict the next state of a process and can block the process indefinitely. The underlying process perceives the blocking action as a slowing down in its processor speed. Since the underlying system may have its own blocking, the controller's blocking action may cause deadlocks where none existed before. It is the controller's responsibility to ensure that this does not happen.

In asynchronous distributed systems, we distinguish between a *computation* and an *execution*. This distinction does not exist for sequential systems. Running a distributed system results in many possible computations, each of which corresponds to multiple possible executions. 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. Our problem will be to control the system to suppress bad executions which do not satisfy the required global property.

Since, in the control system we have described, the controllers can only predict the next state of their processes, we say that they are provided with the underlying computation *on-line*. They are expected to restrict the possible executions allowable within that computation to only those which satisfy the stated global property. We call this scenario the *on-line predicate control problem*. If we assume that each controller is provided with the entire underlying computation *a priori*, then a better control strategy should be possible. We call this scenario the *off-line predicate control problem*. Even though the underlying computation is predetermined in the off-line problem, there is still flexibility in the possible executions that the system may go through. The off-line control strategy must restrict these possible executions.

It may seem unreasonable to assume a predetermined underlying computation in off-line predicate control. However, there are some cases where a system may be run once and a trace of its underlying computation (process states and message orderings) may be made. It may then be necessary to run it again maintaining the same computation. Three areas where this naturally occurs are distributed debugging, distributed recovery and distributed simulations.

- In distributed debugging [12], we may discover a bug in a particular computation of the distributed system and might wish to replay the same computation to localize the bug. Since the bug may occur in certain executions of the computation while not in others, we may wish to specify some global property which restricts our attention to suspicious executions.

- In distributed recovery [5], processes may fail and may have to recover in a manner consistent with the message orderings and sequence of events that they have logged in the past. In this case the predetermined computation of the recovering system is specified in the message logs. While replaying the logs, it might be desired to specify some global property among the recovering processes to prevent bad executions which possibly caused the failures in the first place.

- In distributed simulations [9], the message ordering and sequence of events is fixed in simulation time. So, every run of the simulation would follow the same computation but would have multiple possible executions. In optimistic distributed simulation schemes, owing to a bad scheduling of events, the system may have to backtrack and restart. To prevent the same bad scheduling of events, we may specify some global property for the restarted system to maintain.

For the control problem, we could express the global property as a boolean expression of local atomic predicates each corresponding to some local property on a process. We call this boolean expression a *global predicate*. We will also be interested in a class of global predicates called *disjunctive predicates* consisting of a disjunction of local predicates.

The predicate control problem is to construct a control system that does not cause deadlocks and ensures that every possible execution of the controlled distributed system always satisfies a global predicate. A solution to the problem must either construct a control system or inform us that the global predicate can never be satisfied in the underlying distributed computation (for example, if the initial state itself does not satisfy the specified predicate).

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

Next, we try to restrict the problem. We choose the class of disjunctive predicates because they form a simple, yet interesting, class of problems. The corresponding class of conjunctive predicates could be simply controlled by each controller maintaining the local predicate for its process independently. Since a disjunctive predicate specifies that *at least one* of the local predicates remains true at any global state, the controllers must coordinate in order to satisfy it. Although disjunctive predicates seem to be restrictive, there is a category of real world problems that falls within their scope. These are problems which specify that a certain *bad* combination never occurs at the same time. A good example of such a situation is two-process mutual exclusion where we specify that either one process or the other process is not in the critical section at any time. Another example is the classic dining philosopher's problem where deadlocks may be prevented by specifying that at least one of the philosophers must be thinking at any time.

We solve the off-line predicate control problem for disjunctive predicates by constructing a centralized algorithm which takes the given information about the predetermined underlying computation and produces a control strategy. The controllers follow this control strategy to ensure that the predicate is maintained in every global state and no deadlock occurs. The algorithm also determines if no control scheme exists for a given underlying computation. If there are n processes and a local predicate has a maximum of p changes in value during the computation, then the time complexity of our algorithm is $O(n^2p)$ and the message complexity is $O(np)$. As a measure of the concurrency allowed by our control strategy, there are $O(np)$ one-way, two process synchronizations.

Next, we show that it is impossible to solve the problem of on-line predicate control for disjunctive predicates (and hence, for general predicates). This is a result of the controllers ignorance of the future of the underlying computation beyond the next state. 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, we impose certain restrictions to prevent deadlocks and 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 [1, 8, 14, 16, 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.

In Section 2, we define our model and problems. In Section 3, we show that off-line predicate control is NP-hard. In Section 4, we address off-line predicate control for disjunctive predicates and in Section 5, we do the same for the on-line case. In Section 6, we discuss our conclusions.

2 Model and Problem Specification

2.1 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 *loosely-coupled* and *asynchronous*. Message ordering is arbitrary.

2.2 Model of a Distributed Computation Each process, i, executes a sequence of states and events starting with a special start state, $\bot_i$, and ending with 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 the values of 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 [11], $\rightarrow$, 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).

Let S_i be the sequence of local states in process 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) [7] 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.
D3: A single event does not both send and receive a message.

2.3 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 is an established fact that $(\mathcal{G}, \leq)$ is a lattice [15].

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.

2.4 Global Execution 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 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.

2.5 Model of a Control System The *control system* is a distinct distributed system superimposed on the underlying distributed system. The processes of the control system are a set of *controllers*, $C_1, \ldots, C_n$, which communicate using *control messages* over channels that are independent from the underlying system channels. Each controller monitors and controls the underlying process. In particular, a controller is capable of determining the next state of its process and is capable of blocking the process indefinitely. However, 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. The actions of the controllers are specified by a *distributed control strategy*.

2.6 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.

2.7 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 $x \in X_i$. Let $X = \bigcup_i X_i$. A *global predicate*, B, is a boolean-valued function of the variables in X. We use $B(G)$ to denote the value of predicate B in the global state G. 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*.

2.8 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.

3 Off-line Predicate Control is NP-hard

We show that the off-line predicate control problem is NP-hard by showing that a simpler decision problem is NP-complete. The problem is defined as follows:

Satisfying Global Sequence Detection (SGSD):
Given: a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, and a set of variables X partitioned into n subsets $X_1, \ldots, X_n$, and a

global predicate B defined on X
Determine: if B is feasible for S (i.e. if there exists a global sequence in S that satisfies B.)

Theorem 1 *SGSD is NP-complete*

Proof Outline: The problem is in NP because given a candidate global sequence, it takes polynomial time to check that it is a valid global sequence and that it satisfies B. To show that it is NP-hard, we map the satisfiability problem to it. If b is the boolean expression in the satisfiability problem, then for each variable in b we assign a separate process with two states, one true and one false. We define $B = b \vee x$ where x is an extra boolean variable. We define another process for x which starts true, goes through a false state, and ends true again. We then apply SGSD to find a satisfying global sequence. If it finds one, then the global state with $x = false$ will have a satisfying assignment for the variables of b. Conversely, if b is satisfiable, then there must be a satisfying global sequence. $\Box$

Note that the proof demonstrates that SGSD is NP-complete even without any synchronizations. The predicate control problem requires finding a satisfying distributed control strategy, if one exists. This is a search problem the corresponding decision problem of which is to find out if a satisfying distributed control strategy exists.

Satisfying Control Strategy Detection (SCSD):
Given: a deposet, $(S_1, \ldots, S_n, \rightsquigarrow)$, and a set of variables X partitioned into n subsets $X_1, \ldots, X_n$, and a global predicate B defined on X
Determine: if there exists a distributed control strategy that satisfies B for a control system whose underlying computation is the deposet S

We now show that SCSD and SGSD are equivalent. Since the given data for both problems is identical, we don't have to explicitly map an instance of one problem to the other.

Theorem 2 *SCSD and SGSD are equivalent.*

Proof Outline: 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, then we construct an overly restrictive control strategy that allows only that global sequence and no other. $\Box$

Therefore, SCSD is NP-complete as well and the predicate control problem, being the corresponding search problem, is NP-hard.

4 Off-line Predicate Control for Disjunctive Predicates

We now restrict our attention to disjunctive predicates. We state our problem as:

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.

Further, we make the following assumption:

$$\textbf{A1:} \quad \forall i : \; l_i(\bot_i) \;\wedge\; l_i(\top_i)$$

In practice, this assumption is reasonable for a truly distributed system because processes know nothing about one another before and after their computations. Hence, they must start and end in safe states.

Our approach will be to construct a satisfying controlled deposet of S with $\overset{C}{\rightsquigarrow}$. From the controlled deposet, it is easy to construct a control strategy to implement it by using a control message (with a blocked receive) for every tuple in $\overset{C}{\rightsquigarrow}$.

We define an *interval*, I, as a sequence of consecutive states in a process with a beginning state (designated as $I.lo$) and an ending state (designated as $I.hi$). $l_i(I_i)$ denotes that l_i is *true* throughout I_i and I_i is called a *true-interval*. Similarly, $\overline{l_i}(I_i)$ denotes that the local predicate l_i is *false* throughout

```
Algorithm:

Input:     $I[1..n]$ an array of queues of false-intervals in deposet $S$ ($I[i]$ is a queue of false-intervals of $P_i$ with respect
                 to $l_i$ in $\prec$ order). We use $I_i$ to stand for $head(I[i])$.
Output:    $C$       a queue of tuples of local states, initially $\emptyset$ and finally corresponds to the tuples in the $\overset{C}{\leadsto}$ relation.

Variables: $g[1..n] = \bot$          a global state
           $k = 1, k', l, i, x, y$   integers
           $A$                       set of integer tuples

L0    if $SomeQueueEmpty(I)$ exit($C$);                                  (* no control required *)
L1    while $\neg SomeQueueEmpty(I)$ {                                  (* exit when chain reaches a $\top_i$ *)
L2        $A$ := $\{\langle x,\ y\rangle \mid I_x.lo \not\rightarrow I_y.hi \ \wedge\ g[x] \prec I_x.lo\}$;   (* find a true interval which
L3        if ($A = \emptyset$) exit( "No Controller Exists" );             can be maintained while
L4        else $\langle k',\ l\rangle$ := $any(A)$;                         a false interval is crossed *)
L5        $enqueue(C,\ (g[k'],\ I_k.lo))$;                                (* add a $\overset{C}{\leadsto}$ tuple *)
L6        for ($i \in \{0, \ldots, n\},\ i \neq l$) {
L7            while ($next(i) \rightarrow I_l.hi$) {
L8                $g[i]$ := $next(i)$;                                  (* advance $g$
L9                if ($g[i] = I_i.hi$) $dequeue(I[i])$;                     consistently with $\rightarrow$ *)
              }
          }
L10       $g[l]$ := $I_l.hi$;                                            (* cross one false interval *)
L11       $dequeue(I[l])$;
L12       $k$ := $k'$;                                                   (* remember the true interval *)
      }
L13   $k'$ := $any(\ \{x | empty(I[x])\}\ )$;
L14   $enqueue(C,\ (g[k'],\ I_k.lo))$;                                    (* add last $\overset{C}{\leadsto}$ tuple *)
L15   $dequeue(C)$;                                                      (* eliminate dummy initializer *)
L16   $exit(C)$;

Definitions:
$any(Z) =$ random element of non-empty set $Z$
$next(i) = \begin{cases} \top_i & \text{if } I[i] \text{ is empty} \\ I_i.lo & \text{if } g[i] \prec I_i.lo \\ I_i.hi & \text{if } g[i] = I_i.lo \end{cases}$
```

Figure 1: Algorithm for Off-line Predicate Control of Disjunctive Predicates

false-interval I_i. A set of intervals, $I_1, \ldots, I_n$, is said to *overlap*, represented by $overlap(I_1, \ldots, I_n)$, if and only if: $\forall i, j \in \{1, \ldots, n\} : \ (I_i.lo \rightarrow I_j.hi)$. This definition ensures that for an overlapping set of intervals, no process can leave its interval until all other processes have entered their intervals. Therefore, if we have an overlapping set of false-intervals, then every global sequence must contain a global state where all processes are false. This is stated in the following result [7]:

Lemma 1 *In a deposet,* $(S_1, \ldots, S_n, \leadsto)$, *with causal precedence* $(S, \rightarrow)$, *if the following condition is satisfied then then there is no global sequence in* S *which satisfies* $B = l_1 \vee \ldots \vee l_n$.

$$\exists I_1, \ldots, I_n : \ \overline{l_1}(I_1) \wedge \ldots \wedge \overline{l_n}(I_n) \ \wedge \ overlap(I_1, \ldots, I_n)$$

Our algorithm outputs a valid $\overset{C}{\leadsto}$ unless an overlapping set of false-intervals is detected. The approach followed by our algorithm is to always maintain one process in its true interval until it knows that some other process has started its true interval. This causes a chain of true intervals connected by the $\overset{C}{\leadsto}$ relation which ensures that some process is always true. However, we must ensure that the $\overset{C}{\leadsto}$ used in the chain does not interfere with the existing $\rightarrow$. Our algorithm ensures this by maintaining a global state g which advances consistently in the logical time defined by $\rightarrow$. Every $\overset{C}{\leadsto}$ tuple starts at g and points to the future in logical time. The algorithm is listed in **Figure 1**.

Theorem 3 *The procedure in Figure 1 terminates.*

Proof Outline: We show that every reference to $head(I[j])$ (abbreviated to I_j) and $dequeue(I[j])$ operates on a non-empty queue. We also show that the references to *any* and *next* are well-defined. The inner while loop terminates because every two iterations dequeue one false-interval from $I[i]$. The outer while loop terminates because each iteration must dequeue at least one false-interval from I. $\Box$

Theorem 4 *The algorithm in Figure 1 correctly solves the Off-line Predicate Detection Problem for Disjunctive Predicates.*

Proof Outline: We first show that when the algorithm outputs *"No Controller Exists"*, S is infeasible for B. We prove the following property of global state g: either the set of false-intervals that end after it are overlapping, or at least one of those false-intervals y may be crossed while some other process x, which is true at g ($g[x] \prec I_x.lo$), is kept true. This demonstrates the correctness of the abnormal exit in L3.

Next, we show that $\overset{C}{\rightsquigarrow}$ does not interfere with $\rightarrow$ as follows. For the global states defined by g and $next$, no local state in $next$ can precede a state in g in the logical time defined by $\rightarrow$. Since each $\overset{C}{\rightsquigarrow}$ tuple starts at a state in g and points towards a state in $next$ ($I_k.lo$ is $next(k)$), $\overset{C}{\rightsquigarrow}$ cannot interfere with $\rightarrow$.

Lastly we show that every global sequence possible in the controlled deposet of S with $\overset{C}{\rightsquigarrow}$ must satisfy B. Any global state must cut the chain of true intervals connected by $\overset{C}{\rightsquigarrow}$ tuples (since the chain extends from $\bot_i$ to $\top_j$ for some i and j). Either it cuts the chain at a $\overset{C}{\rightsquigarrow}$ tuple violating the causality it imposes, or it cuts the chain in a true interval and so satisfies B. $\Box$

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 in L2 can take $O(n^2)$ time to check every pair of processes. However, an optimized implementation would avoid redundant comparisons in L2 by computing the set A dynamically. Since, in this approach, each new false-intervals would have to be compared with $n-1$ existing false-intervals, the complexity is $O(n^2p)$. The size of $\overset{C}{\rightsquigarrow}$ is $O(np)$ because one tuple is outputed in each iteration of the outer while loop. Therefore, the message complexity of control messages used is $O(np)$. 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.

5 On-line Predicate Control for Disjunctive Predicates

We address the following problem in this section:

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.

First we show that it is, in general, impossible to solve this on-line predicate control problem.

Theorem 5 *The On-Line Predicate Control Problem for Disjunctive Predicates is impossible to solve.*

Proof Outline: For simplicity, we prove by contradiction for the case $n = 2$. The proof can be extended to a general n. Our counterexample consists of two processes P_1 and P_2 which each start at a true state, pass through a false state, and end in a true state. The on-line controllers only know about their next

```
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($req(i)$, $any(\mathcal{C})$);
        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
        $any(Z)$     randomly chosen element of non-empty set $Z$
```

Figure 2: Distributed Control Strategy for On-line Predicate Control with Disjunctive Predicates

states and so we are free to have either process send the other a message after the second state to be received before the third state. The controllers cannot know about this message when they start. Since the position is symmetric, let process P_1 advance to its false second state while P_2 stays in its true first state. We then make P_2 send a message to P_1 after its second state and before P_1's third state. We now have a deadlock where none would have existed if P_2 had been advanced instead. $\Box$

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 similar counterexample to demonstrate impossibility. Since it is impossible to solve the problem as it stands, we make the following assumptions:

A2: $\forall i : P_i$ does not block in states where l_i is *false* .

A3: $\forall i : l_i(\top_i)$

These assumptions essentially allows us to assume that a *false* state will eventually turn *true* without blocking. Since our control strategy will only wait for a process while it is in the *false* state, this prevents a circular wait from occurring, and thus prevents deadlocks. In the two-process mutual exclusion example, these assumptions would correspond to the usual assumption that a process does not block while it is inside a critical section and ends in a non-critical section.

Our control strategy is similar to that used in the off-line case. One process remains true until it is sure that some other process is true. At any time, the process bearing such a responsibility is called the *scapegoat*. In our algorithm, listed in **Figure 2**, when the scapegoat reaches a false interval, it simply sends a request to some other process asking it to become the scapegoat and waits for an acknowledgment.

It is easy to prove the correctness of this strategy, namely that:

Theorem 6 *The distributed control strategy listed in* **Figure 2** *does not deadlock.*

Theorem 7 *The distributed control strategy listed in* **Figure 2** *satisfies B.*

The k-mutual exclusion problem [1, 8, 14, 16, 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 simple 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 [1, 8, 14, 16, 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.

6 Conclusions

We have defined the predicate control problem, a generalization of distributed control problems, and have defined two different scenarios for the problem – the on-line and off-line scenarios. Although we have shown that it is NP-hard to solve the general off-line problem, we demonstrate that the restricted problem for the class of disjunctive predicates may be solved efficiently. For the on-line scenario, we have demonstrated the impossibility of finding a solution even for the limited class of disjunctive predicates. However, if we impose restrictions on the underlying computation, a solution is possible. The on-line predicate control problem for disjunctive predicates is equivalent to the $(n-1)$-mutual exclusion problem, a special case of the k-mutual exclusion problem. However, we have shown that it is possible to provide a simpler and more efficient solution to the $(n-1)$-mutual exclusion problem than can be obtained by specializing existing solutions to the k-mutual exclusion problem.

There are a number of interesting directions for future research into the predicate control problem. We have shown that efficient solutions exist for off-line and on-line control of disjunctive predicates. However, a number of existing distributed control problems such as general mutual exclusion do not fall into this class and yet have been solved. It would be interesting to study other classes of predicates to see if a general control strategy may be devised for them. A possible approach would be to try to extend the solutions for disjunctive predicates to conjuncts of disjunctive predicates.

Another interesting problem would be to study the $(n-1)$-mutual exclusion problem more thoroughly since we have demonstrated that it is a special and useful case of the k-mutual exclusion problem. Further, it may lead to insights into solutions for the k-mutual exclusion problem for large k.

References

[1] 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.

[2] T. Casavant and J. Kuhl. A taxonomy of scheduling in general purpose distributed computer systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[3] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632 – 646, October 1984.

[4] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1 – 4, August 1980.

[5] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University, 1996.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374 – 382, April 1985.

[7] V. K. Garg. *Principles of Distributed Systems.* Kluwer Academic Publishers, 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] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404 – 425, July 1985.

[10] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303 – 328, December 1987.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.

[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] 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 Appendix: Proofs

Theorem 8 *SGSD is NP-complete*

Proof: We first show that the problem is in NP. Given a sequence of global states, g, we have to check that it is a valid global sequence and that every global state in it satisfies B. In order to check that it is a valid global sequence, we check that each of the global states g_k in g is consistent, that for all consecutive pairs of global states, $g_k \leq g_{k+1}$, and that restricting g to a single process i would produce either S_i or a stutter of it.

- We can check that g_k is consistent in polynomial time (by using vector clocks and examining all pairs of states from g_k). There are only a polynomial number of global states in the global sequence because each global state in the sequence must contain at least one new local state.
- We can check that $g_k \leq g_{k+1}$ in polynomial time by checking that $\forall i : g_k[i] \preceq g_{k+1}[i]$ (using vector clocks). Again, there are a polynomial number of global states in g.
- For each of the processes, we can restrict g to i and check that the restriction is either S_i or a stutter of it in polynomial time by starting at the beginning of the restriction and striking off elements of S_i every time the state changes.
- For each global state g_k in g we need to check that it satisfies B. This can be done in polynomial time since there are only a polynomial number of such states and B can be evaluated in polynomial time.

We now show NP-completeness by reducing the satisfiability problem of a boolean expression to SGSD.

Let b be the given boolean expression and let it use m boolean variables $x_1, \ldots, x_m$. We define a deposet, $\{S_1, \ldots, S_m, S_{m+1}, \emptyset\}$, such that for each P_i such that $i \in \{1, \ldots, m\}$, $X_i = \{x_i\}$ and S_i consists of exactly two states, $\bot_i = s_i^t$ in which x_i is *true* and $\top_i = s_i^f$ in which x_i is *false* . We introduce an extra variable x_{m+1} such that for process P_{m+1}, $X_{m+1} = \{x_{m+1}\}$ and S_{m+1} is a sequence of three states $\bot_{m+1} = s_{m+1}^{t1}$ in which x_{m+1} is *true* , s_{m+1}^f in which x_{m+1} is *false* and the $\top_{m+1} = s_{m+1}^{t2}$ in which x_{m+1} is *true* again. Note that there are no messages in the deposet. Now we define $B = b \vee x_{m+1}$ so that it is obviously *true* in $\bot$ and in $\top$.

Given a global sequence for which every global state satisfies B, by the definition of a global sequence, there must be some global state containing the local state, s_{m+1}^f where x_{m+1} is *false* . Since this global state must satisfy B, we can find a truth assignment from the states of $P_1, \ldots, P_m$ which satisfies b.

Given a truth assignment which satisfies b, we construct a global state h containing s_{m+1}^f, and for each S_i such that $i \in \{1, \ldots, m\}$, it contains s_i^t if the variable x_i is *true* in the given truth assignment or s_i^f, otherwise. The sequence $\bot, h, \top$ is a global sequence because each state is consistent (because there are no synchronizations) and restricting it to each process P_i results in either S_i or a stutter of it. The predicate B evaluates to *true* in $\bot$ and $\top$ because x_{m+1} is *true* and B evaluates to *true* in h because of the given truth assignment.

This shows that we can find a truth assignment to b iff we can find a global sequence in S which satisfies B. $\Box$

Theorem 9 *SCSD and SGSD are equivalent.*

Proof: If we solve SCSD and find a satisfying control strategy, then we simulate a global execution of the system with the control strategy and find a satisfying global sequence.

If we solve SGSD and find a satisfying global sequence g, then we can define a satisfying control strategy 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 ($k < l$ and $k, l \in \{1, \ldots, m\}$) 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 $\overset{C}{\rightarrow}$. Every global state not in g is inconsistent for $\overset{C}{\rightarrow}$ because it must consist of at least two local states which uniquely belong to distinct global states of g. These local states must be connected by $\overset{C}{\rightsquigarrow}$ by our construction 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. Hence, this is a satisfying control strategy.

Thus, SCSD and SGSD are equivalent decision problems. $\square$

In order to prove **Theorem 3** and **Theorem 4**, we prove the following lemma which expresses some useful invariants of the outer while loop in the algorithm listed in **Figure 1**.

Lemma 2
Immediately before L2 the following invariant holds:

C1: $\neg SomeQueueEmpty(I) \ \wedge$
C2: $\forall i : g[i] \preceq I_i.lo \ \wedge$
C3: $\forall i : (g[i] \prec I_i.lo) \Rightarrow (g[i] = \bot_i \vee \textit{(interval between } g[i] \textit{ and } I_i.lo \textit{ is a true interval)}\ ) \ \wedge$
C4: $\exists x, y : (I_x.lo \not\rightarrow I_y.hi \ \wedge \ g[x] \prec I_x.lo) \ \vee \ \neg(\exists \textit{ a satisfying global sequence}) \wedge$
C5: $\forall i : g[i] = \bot_i \ \vee \ g[i] \prec I_i.lo \ \vee \ \exists j : (g[i] \rightarrow g[j] \ \wedge \ g[j] \prec I_j.lo) \ \wedge$
C6: $\forall i, j : next(i) \not\rightarrow g[j]$

Proof:
C1 is invariant directly from the while loop condition. This establishes that $\forall i : I_i$ is always a well-defined interval at L2. We use this property implicitly in the remainder of this proof.

We now prove **C2** $\wedge$ **C3** $\wedge$ **C4** $\wedge$ **C5** $\wedge$ **C6** by induction on the number of occurrences of L2 in the execution of the algorithm.

Base: At the first occurrence of L2:

C2: obvious {definition of $\bot$}

C3: obvious

C4: if $\neg$($\exists$ a satisfying global sequence), we are done.
so assume $\exists$ a satisfying global sequence
so $\exists x, y : (I_x.lo \not\rightarrow I_y.hi)$ {Lemma 1}
so let $(I_x.lo \not\rightarrow I_y.hi)$
$g[x] = \bot_x \prec I_x.lo$ {A1, definition of false-interval}
so $\exists x, y : (I_x.lo \not\rightarrow I_y.hi \ \wedge \ g[x] \prec I_x.lo)$

C5: obvious

C6: obvious {D1, definition of $\bot_i$ }

Induction: Let IH refer to the inductive hypothesis. We use primed variables (e.g. g', I') to indicate their values in the current occurrence of L2 and unprimed variables (e.g. g, I) to indicate their values in the previous occurrence. Lines numbers in the algorithm refer to their occurrences in the previous iteration (corresponding to IH).

C2: $g'[l] \prec I'_l.lo$ {L10, L11, order of $I[l]$ is $\prec$}

$\forall i \neq l : g'[i] \preceq I'_i.lo$

{if loop L7 iterates then by {L8, L9, defn. of next} else by {IH **C2**}}

so $\forall i : g'[i] \preceq I'_i.lo$

C3: let $i \in \{1, \ldots, n\}$

if $g'[i] \not\prec I'_i.lo$ then we are done.

P1 so assume $g'[i] \prec I'_i.lo$

consider two cases:

Case 1: $g'[i] = g[i]$

$I'_i = I_i$ {Case 1, L7, L8, L9}

so we are done. {IH **C3**}

Case 2: $g'[i] \neq g[i]$

consider two cases:

Case 2.1: $i = l$

$g'[i]$ is hi of interval immediately before I'_i

{Case 2.1, L10, L11}

so interval between $g'[i]$ and $I'_i.lo$ is a true interval

{definition of intervals, $I[i]$ ordered in $\prec$}

Case 2.2: $i \neq l$

$g'[i]$ is hi of interval immediately before I'_i

{final iteration of L7-9, by P1: $g'[i] \neq I'_i.lo$}

so interval between $g'[i]$ and $I'_i.lo$ is a true interval

{definition of intervals, $I[i]$ ordered in $\prec$}

C4: if $\neg(\exists$ a satisfying global sequence), we are done.

so assume $\exists$ a satisfying global sequence

so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi)$ {Lemma 1}

P2 so let $(I'_x.lo \not\rightarrow I'_y.hi)$

if $g'[x] \prec I'_x.lo$ then we are done

so assume $g'[x] \not\prec I'_x.lo$

P3 so $g'[x] = I'_x.lo$ {**C2** above}

$\forall i \neq l : (g'[i] = g[i]) \vee (g'[i] \rightarrow g'[l])$ {if loop L7 doesn't iterate}

{L7, L8, L9 ensure that at the end of the loop, $next'(i) \rightarrow I_l.hi$.

by definition of $next(i)$: $g'[i] \prec next'(i)$.

so $g'[i] \rightarrow I_l.hi$.

by L10: $g'[l] = I_l.hi$.

so $g'[i] \rightarrow g'[l]$.}

P4 so $\forall i : (i = l) \ \vee \ (g'[i] \rightarrow g'[l]) \ \vee \ (g'[i] = g[i])$

so we have 3 cases for x:

Case 1: $x = l$

P5	$g'[l] \prec I'_l.lo$	{L10, L11}
P6	$g'[x] = g'[l]$	{Case 1}
P7	$I'_x.lo = g'[x]$	{P3}
	so $I'_x.lo \prec I'_l.lo$	{P5, P6, P7}
P8	so $I'_l.lo \not\rightarrow I'_y.hi$	{P2}
	so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi \ \wedge \ g'[x] \prec I'_x.lo)$	{P5, P8}

Case 2: $g'[x] \rightarrow g'[l]$

P9	$g'[l] \prec I'_l.lo$	{L10, L11}
P10	$g'[x] \rightarrow g'[l]$	{Case 2}
P11	$I'_x.lo = g'[x]$	{P3}
	so $I'_x.lo \rightarrow I'_l.lo$	{P9, P10, P11}
P12	so $I'_l.lo \not\rightarrow I'_y.hi$	{P2}
	so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi \ \wedge \ g'[x] \prec I'_x.lo)$	{P9, P12}

Case 3: $g'[x] = g[x]$

P13	$g[x] \neq \bot_x$	{P3, Case 3, A1, definition of interval}
	$I_x = I'_x$	{Case 3, L7, L8, L9}
P14	so $g[x] = I_x.lo$	{P3, Case 3}
	$\exists j : g[x] \rightarrow g[j] \ \wedge \ g[j] \prec I_j.lo$	{IH **C5**, P13, P14}
P15	so let $g[x] \rightarrow g[j] \ \wedge \ g[j] \prec I_j.lo$	
we have 3 cases for j:		{P4}

Case 3.1 $j = l$

P16	$g'[l] \prec I'_l.lo$	{L10, L11}
P17	$g'[j] = g'[l]$	{Case 3.1}
P18	$g[j] \prec g'[j]$	{Case 3.1, L2, L4, L10, L11}
P19	$g[x] \rightarrow g[j]$	{P15}
P20	$g'[x] = g[x]$	{Case 3}
P21	$I'_x.lo = g'[x]$	{P3}
	so $I'_x.lo \rightarrow I'_l.lo$	{P16 - P21}
P22	so $I'_l.lo \not\rightarrow I'_y.lo$	{P2}
	so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi \ \wedge \ g'[x] \prec I'_x.lo)$	{P16, P22}

Case 3.2 $g'[j] \rightarrow g'[l]$

P23	$g'[l] \prec I'_l.lo$	{L10, L11}
P24	$g'[j] \rightarrow g'[l]$	{Case 3.2}
P25	$g[j] \preceq g'[j]$	{Case 3.2, L7, L8, L9, definition of *next*}
P26	$g[x] \rightarrow g[j]$	{P15}
P27	$g'[x] = g[x]$	{Case 3}
P28	$I'_x.lo = g'[x]$	{P3}
	so $I'_x.lo \rightarrow I'_l.lo$	{P23 - P28}
P29	so $I'_l.lo \not\rightarrow I'_y.lo$	{P2}
	so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi \ \wedge \ g'[x] \prec I'_x.lo)$	{P23, P29}

Case 3.3 $g'[j] = g[j]$

	$I'_j = I_j$	{Case 3.3, L7, L8, L9}
P30	so $g'[j] \prec I'_j.lo$	{P15, Case 3.3}
P31	$g[j] = g'[j]$	{Case 3.3}
P32	$g[x] \rightarrow g[j]$	{P15}
P33	$g'[x] = g[x]$	{Case 3}

P34 $I'_x.lo = g'[x]$ {P3}
so $I'_x.lo \rightarrow I'_j.lo$ {P30 - P34}
P35 so $I'_j.lo \not\rightarrow I'_y.lo$ {P2}
so $\exists x, y : (I'_x.lo \not\rightarrow I'_y.hi \ \wedge \ g'[x] \prec I'_x.lo)$ {P30, P35}

C5: let $i \in \{1, \ldots, n\}$
we have three cases for i: {P4}

Case 1: $i = l$

$g'[l] \prec I'_l.lo$ {L10, L11}
so $g'[i] \prec I'_i.lo$ {Case 1}

Case 2: $g'[i] \rightarrow g'[l]$

$g'[l] \prec I'_l.lo$ {L10, L11}
so $g'[i] \rightarrow g'[l] \ \wedge \ g'[l] \prec I'_l.lo$ {Case 2}

Case 3: $g'[i] = g[i]$

$I_i = I'_i$ {Case 3, L7, L8, L9}
so $g'[i] = \perp_i \ \vee \ g'[i] \prec I'_i.lo \ \vee \ \exists j : (g'[i] \rightarrow g[j] \ \wedge \ g[j] \prec I_j.lo)$
{IH **C5**, Case 3}
if $g'[i] = \perp_i \ \vee \ g'[i] \prec I'_i.lo$, we are done.
P36 so let $g'[i] \rightarrow g[j] \ \wedge \ g[j] \prec I_j.lo$
we have three cases for j: {P4}

Case 3.1 $j = l$
$g[l] \prec g'[l]$ {L2, L4, L10, L11}
P37 so $g'[i] \rightarrow g'[l]$ {Case 3.1, P36}
$g'[l] \prec I'_l.lo$ {L10, L11}
so $g'[i] \rightarrow g'[l] \ \wedge \ g'[l] \prec I'_l.lo$ {P37}

Case 3.2 $g'[j] \rightarrow g'[l]$
if $g'[j] = g[j]$ then same as Case 3.3.
so let $g'[j] \neq g[j]$
so $g[j] \prec g'[j]$ {Case 3.2, L7, L8, L9}
P38 so $g'[i] \rightarrow g'[l]$ {P36, Case 3.2}
$g'[l] \prec I'_l.lo$ {L10, L11}
so $g'[i] \rightarrow g'[l] \ \wedge \ g'[l] \prec I'_l.lo$ {P38}

Case 3.3 $g'[j] = g[j]$
$I_j = I'_j$ {Case 3.3, L7, L8, L9}
so, $g'[i] \rightarrow g'[j] \ \wedge \ g'[j] \prec I'_j.lo$ {Case 3.3, P36}

C6: let $i, j \in \{1, \ldots, n\}$
we have 3 cases for j: {P4}

Case 1: $j = l$

$next'(i) \not\rightarrow g'[l]$
{if $(i = l)$ then by {definition of $next$} else by {L7, L8, L9, L10}}
so $next'(i) \not\rightarrow g'[j]$ {Case 1}

Case 2: $g'[j] \rightarrow g'[l]$

$next'(i) \not\rightarrow g'[l]$
{if $(i = l)$ then by {definition of *next*} else by {L7, L8, L9, L10}}
so $next'(i) \not\rightarrow g'[j]$ {Case 2}

Case 3: $g'[j] = g[j]$

P39	$next(i) \preceq next'(i)$	{definition of *next*, IH **C2**, Case 3, L7 - L9}
	$next(i) \not\rightarrow g[j]$	{IH **C6**}
	so $next'(i) \not\rightarrow g'[j]$	{Case 3, P39}

□

Theorem 10 *The procedure in Figure 1 terminates.*

Proof: We must show that each term used is well defined in the execution and that each loop terminates. The terms which may be undefined are:

- I_j for $j \in \{l, x, y, k, i\}$ used throughout the program. I_j is an abbreviation of $head(I[j])$, which is not defined if $I[j]$ is empty.
  - at L2 use of I_x, I_y: **Lemma 2:C1** states that no queue in I is empty at the start of L2. So the usage is valid.
  - at L5 use of I_k: because of **Lemma 2:C1** and since there are no *dequeue*'s between the start of L2 and the start of L5.
  - at L7 use of I_l: because of **Lemma 2:C1** and since there is no *dequeue* of $I[l]$ between the start of L2 and an occurrence in L7.
  - at L10 use of I_l: same as above.
  - at L14 use of I_k: The check in L0 makes sure that the outer while-loop iterates at least once. Consider the final iteration of the loop. At L4, a k' is chosen and $I[k']$ cannot be dequeued at L9 because L2 and L7 ensure that the loop L7-9 never iterates for P'_k. Since $k' \neq l$, it can't be dequeued at L11. Queue $I[k]$ was not changed in the last iteration. At the start it couldn't have been empty because of **Lemma 2:C1**. So it is non-empty at the end. Since there is dequeue from then until L14, the usage is valid.
  - in the definition of *next*: this is valid because the definition first checks if $I[i]$ is empty.
- $any(A)$ at line L4. A must not be empty and this is ensured in L3.
- $next(i)$ at lines L7 and L8. At the start of L2, by **Lemma 2:C2** $\forall i : next(i)$ is valid. **Lemma 2:C2** is also an invariant at the start of L8 in the inner while-loop (though not at the start of L9). Hence the usage of $next(i)$ is valid.
- *dequeue*'s The *dequeue* in L15 is always preceded by at least one *enqueue* and is the only *dequeue* of C. The *dequeue* at L11 is valid because of **Lemma 2:C1**. The dequeue at L9 is valid because the definition of *next*, **D2**, and the termination check in the inner while-loop prevents a *dequeue* when $I[i]$ is empty.

We now show that the loops must terminate.

- The inner while-loop terminates because the queue $I[i]$ is finite and keeps reducing every two iterations. In the worst case, by the definition of *next*, the loop must terminate when $I[i]$ becomes empty.

- The for loop has a finite index range.
- The outer while loop terminates because I is finite and reduces by at least one interval (L11) in every iteration.

□

Theorem 11 *The algorithm in Figure 1 correctly solves the Off-line Predicate Detection Problem for Disjunctive Predicates.*

Proof: There are three parts:
Part 1: *If the algorithm exits with "No Controller Exists" then there is no controller which satisfies B.*
Proof: Lemma 2:C4 is true at the start of L2. This ensures that at L3 A is empty only if there is no global sequence which satisfies B. Since any controller which satisfies B can be executed to produce a global sequence which satisfies B, there can be no controller which satisfies B.
Part 2: *The output, C, is a valid $\overset{C}{\leadsto}$ relation that does not interfere with $\to$.*
Proof: It is easy to see that C defines a relation on states of deposet S. We must show that $\overset{C}{\leadsto}$ causes no cycles with $\to$. We consider two cases - cycles containing exactly one $\overset{C}{\leadsto}$ tuple and cycles containing two or more $\overset{C}{\leadsto}$ tuples.

- *Claim: There are no cycles containing exactly one $\overset{C}{\leadsto}$ in the transitive closure of $\overset{C}{\leadsto}$ and $\to$.*
  We prove this by contradiction. Let there be such a cycle with a tuple added in either line L5 or in line L14 as $g[k']\overset{C}{\leadsto}I_k.lo$. We consider these two cases together.

P40	$g[k']\overset{C}{\leadsto}I_k.lo$	{given}
P41	$I_k.lo = next(k)$	{There must be a previous iteration of the outer while loop because: in the case of $g[k']\overset{C}{\leadsto}I_k.lo$ being added in L5, it can't be the first iteration because that tuple would be dequeued in line L15 and in the case of its being added in L14, the check in line L0 would make sure that the loop iterates at least once. In the previous iteration, L2, L4 indicate that the k' for that iteration (the k in our current iteration) is such that $g[k'] \prec I'_k.lo$. This would remain unaffected by the loop in lines L7 - L9 because of the conditions in L2 and L4. So after line L12, this changes to $g[k] \prec I_k.lo$. By the definition of $next$, $I_k.lo = next(k)$.}
	$next(k)\not\to g[k']$	{ **Lemma 2:C6** is true before L2. It remains true until L5. We can show in a similar manner as the inductive step of the proof of **C6** that it remains true until L14.}
P42	so $I_k.lo\not\to g[k']$	{P41}

  P40 and P42 contradict the existence of a cycle in the transitive closure of $\to$ and $\overset{C}{\leadsto}$ with a single $\overset{C}{\leadsto}$ tuple, $g[k']\overset{C}{\leadsto}I_k.lo$.

- *Claim: There are no cycles containing two or more $\overset{C}{\leadsto}$'s in the transitive closure of $\overset{C}{\leadsto}$ and $\to$.*
  Again, we prove this result by contradiction. Let there be such a cycle. Let the superscript m on a variable (e.g. g^m, k'^m) represent its value after m iterations of the outer while loop. So $g^m[k'^m]\overset{C}{\leadsto}I^m_{k^m}.lo$ represents the $\overset{C}{\leadsto}$ tuple in the cycle added after m iterations of the outer while loop. Let $g^i[k'^i]\overset{C}{\leadsto}I^i_{k^i}.lo$ be the last $\overset{C}{\leadsto}$ tuple in the cycle to be added in the algorithm and let $g^j[k'^j]\overset{C}{\leadsto}I^j_{k^j}.lo$ be the next $\overset{C}{\leadsto}$ tuple in the cycle.

P43 so $I^i_{k^i}.lo \to g^j[k'^j]$
P44 $g^j[k'^j] \preceq g^i[k'^j]$ {since $i > j$ and g advances w.r.t. $\preceq$ with each iteration}
$I^i_{k^i}.lo = next^i(k^i)$ {following the same reasoning as in the previous Claim}
P45 so, $next^i(k^i) \to g^i[k'^j]$ {P43, P44}
P46 $next^i(k^i) \not\to g^i[k'^j]$ {following the same reasoning as in the previous Claim}

P45 and P46 are contradictory.

Part3: *For every consistent global state, g, in the new controlled deposet with the new $\overset{C}{\to}$ partial order $B(g)$ is* true .
Proof: We use the same notation used in the second Case in Part 2. Let there be m iterations of the outer while loop in the algorithm. Therefore we have m tuples of $\overset{C}{\rightsquigarrow}$, $m-1$ of which are added in L5 in the loop and the last is added in L14 after the loop terminates. Each of these correspond to:

P47 $g^i[k'^i] \overset{C}{\rightsquigarrow} I^i_{k^i}.lo$ for $i \in \{1, \ldots, m\}$
$g^i[k'^i] \prec I^i_{k'^i}.lo$ for $i \in \{1, \ldots, m-1\}$ {L2, L4 (note: $(i = m)$ excluded)}
P48 so $g^i[k'^i] = \perp_{k'^i}$ $\vee$ (interval between $g^i[k'^i]$ and $I^i_{k'^i}.lo$ is a true interval) for $i \in \{1, \ldots, m-1\}$
{**Lemma 1:C3**}
We simplify the notation as:
P49 $(b_{p_i}, t_{p_i}) = (g^i[k'^i], I^i_{k'^i}.lo)$ for $i \in \{1, \ldots, m-1\}$
where $p_i = k'^i$
We define:
P50 $(b_{p_0}, t_{p_0}) = (\perp_{p_0}, I^1_{k^1}.lo)$
and:
P51 $(b_{p_m}, t_{p_m}) = (g^m[k'^m], \top_{p_m})$
P52 so (b_{p_i}, t_{p_i}) *strictly* includes a true interval for $i \in \{0, \ldots, m\}$
{for $i \in \{1, \ldots, m-1\}$: P48, using **A1** if $b_{p_i} = \perp_{p_i}$
for $i = 0$: we can show from the algorithm that $I^1_{k^1}.lo$ is the start of the first false interval
for $i = m$: L13 shows that g has crossed the last false interval}
$I^i_{k'^i}.lo = I^{i+1}_{k^{i+1}}.lo$ for $i \in \{1, \ldots, m-1\}$
{L2,L4 ensure that $I[k']$ is not affected in the loop L7-L9 and L12 makes k' into k for the next iteration}
P53 so $b_{p_{i+1}} \overset{C}{\rightsquigarrow} t_{p_i}$ for $i \in \{0, \ldots, m-1\}$
{ P47, P49, P50, P51}

Now, we prove by contradiction. Assume G is a consistent global state in the controlled deposet for which $\forall i : \neg l_i(G[i])$

Claim: $\forall i \in \{0, \ldots, m\} : t_{p_i} \preceq G[p_i]$
We prove this claim by induction:
Base: $t_{p_0} \preceq G[p_0]$ {P50, **A1**, P52, definition of $\perp_{p_0}$}
Induction:
$t_{p_i} \preceq G[p_i]$ {inductive hypothesis}
so $b_{p_{i+1}} \overset{C}{\to} G[p_i]$ {P53}
so $G[p_{i+1}] \not\preceq b_{p_{i+1}}$ {otherwise, $G[p_{i+1}] \overset{C}{\to} G[p_i]$ violating consistency of G}
so $b_{p_{i+1}} \prec G[p_{i+1}]$
so $t_{p_{i+1}} \preceq G[p_{i+1}]$ {P52}
$\Box${*Claim*}

In particular:

$t_{p_m} \preceq G[p_m]$

so $G[p_m] = \top_{p_m}$ $\{$P51, definition of $\top_{p_m}\}$

This contradicts **A1**

□

Theorem 12 *The On-Line Predicate Control Problem for Disjunctive Predicates is impossible to solve.*

Proof: We prove impossibility by contradiction. We assume that there is a solution and demonstrate its invalidity using a counterexample. Our counterexample is for the case $n = 2$ for simplicity. It is easy to extend it to the general case of n processes. The counterexample consists of a deposet S and a disjunctive predicate B that is feasible for S. We show that any control strategy that is defined on-line could be forced to deadlock.

Let $S_1 = s^t_{11}, s^f_{12}, s^t_{13}$ and let $S_2 = s^t_{21}, s^f_{22}, s^t_{23}$. 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 distributed strategy starts with control in the global state (s^t_{11}, s^t_{21}) and is aware of the next events and states but is ignorant of what comes after the second states. Since the situation is perfectly symmetrical, it has a choice of any of two possible next global states in the global sequence corresponding to which process advances. Without loss of generality, let P_1 advance so that the next global state is (s^f_{12}, s^t_{21}). (Note that (s^f_{12}, s^f_{22}) would cause B to be violated.) After this decision is made, we play the adversary and impose $s^f_{22} \leadsto s^t_{13}$. This would mean that P_1 has to wait at s^f_{12} for a message to be received from P_1 after state s^f_{22}. The system cannot advance to (s^f_{12}, s^f_{22}) because this would violate B. Hence, the system is deadlocked.

Our counterexample is not valid until we show that B was indeed feasible for S . This is demonstrated by the global sequence, $(s^t_{11}, s^t_{21}), (s^t_{11}, s^f_{22}), (s^f_{12}, s^t_{23}), (s^t_{13}, s^t_{23})$, which satisfies B.

□

Theorem 13 *The distributed control strategy listed in* **Figure 2** *does not deadlock.*

Proof: 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 **A2 and A3**). Therefore, there can be no deadlock. □

Theorem 14 *The distributed control strategy listed in* **Figure 2** *satisfies B.*

Proof: 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. The strategy also ensures that the scapegoat is *true* . This ensures that every possible global state satisfies B. □