Predicate Control in Distributed Systems

Ashis Tarafdar and Vijay K. Garg

TR-PDS-1997-006 May 1997

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

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 involve controlling a distributed system to maintain invariant global properties. Usually, the complexity of synchronization to maintain these invariants is part of distributed algorithm design. We propose an alternative approach in which the invariant may be specified to a supervisory control system. The control system then bears the burden of synchronizing the distributed system in order to maintain the invariant. The *predicate control problem* involves transforming a general global invariant into a synchronization strategy for a control system.

We formalize the predicate control problem in both off-line and on-line scenarios. We motivate its relevance with real-world examples. We prove that general off-line predicate control 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.

1 Introduction

An intrinsic problem in asynchronous distributed systems is that although no one process can have a global view of the system, we still require the system as a whole to maintain invariant global properties. Such global invariants are required in many important problems, such as: mutual exclusion, resource allocation, load balancing and consensus. Distributed algorithms usually incorporate a synchronization strategy to ensure that a required global invariant is maintained. An alternative approach is to specify the global invariant to a control system which then bears the responsibility of maintaining it. This separates the complexity of synchronization from distributed algorithm design. The *predicate control problem* involves transforming a general global invariant into a synchronization strategy for a control system.

Consider, for example, a global invariant that specifies that two processes P_1 and P_2 may not be in their critical sections CS_1 and CS_2 at the same time (Figure 1). This is the well-known distributed mutual exclusion problem for two processes. In solving the predicate control problem, we must specify how the control system would synchronize the system to ensure mutual exclusion.

*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

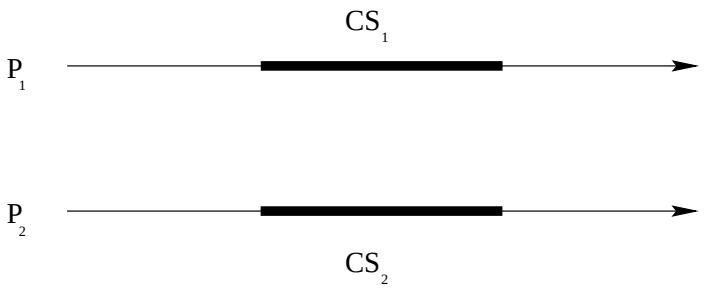

Figure 1: Example: Mutual exclusion

It has two choices: to either allow P_1 to enter CS_1 or to allow P_2 to enter CS_2. Suppose it first allows CS_1 to be entered. It may later discover that before CS_1 can complete, CS_2 must be entered because of an underlying synchronization in the system to be controlled. If this had been known before-hand, the control system would have caused CS_2 to be entered first instead. However, without this pre-knowledge the problem of deciding which critical section to enter is impossible to solve. (Solutions to mutual exclusion get around this by assuming that critical sections do not block.) This example demonstrates the need for two distinct scenarios while studying the predicate control problem: the *off-line* scenario, in which we assume that all synchronizations of the system to be controlled are known *a priori*, and the *on-line* scenario, in which we don't assume such pre-knowledge. In solving the on-line predicate control problem, we will have to make assumptions about the system to be controlled in order to ensure that a solution is feasible.

The main contributions of this paper are as follows. We formulate the new predicate control problem under both off-line and on-line scenarios. (Section 2). Later in this section, we motivate the relevance of the problem in both of these scenarios to real-world situations. We establish that general off-line predicate control is NP-hard (Section 3). We solve both the on-line and off-line problems for the class of *disjunctive predicates* (Sections 4 and 5). Later in this section, we demonstrate how these solutions for disjunctive predicates could be applied to solving real-world problems.

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 assumes asynchronous processes communicating with messages. Control is achieved by superimposing a distinct distributed *control system* on the underlying distributed system as follows. Assign one *controller* 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 blocking it indefinitely. The control mechanism is transparent because the underlying process would perceive the blocking action as a decrease in processor speed. 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. 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 is to control the system to suppress bad executions which do not satisfy the required global predicate. In general, the global predicate is any boolean expression on local atomic predicates that may be efficiently evaluated.

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. It is usually with the purpose of scheduling these interactions in real-time that global synchronization is required. For example, it is often required to suppress executions which violate mutual exclusion leading to the design of distributed mutual exclusion protocols.

We define the distributed control problem to be either off-line or on-line depending on whether the future computation of the underlying system is known *a priori* or not. A good analogy is the scheduling problem in real-time systems which has both on-line and off-line scenarios depending on whether the task set is known *a priori* or not. In general, an on-line problem would be harder to solve than the corresponding off-line problem.

Motivation for On-line Predicate Control. On-line predicate control captures the common case of synchronization in systems for which the future computation is unknown. We present two examples.

Assuming an invariant global property often simplifies the design of a distributed system. One well-known example is distributed mutual exclusion where the designer is provided a critical section primitive. In the on-line predicate control approach, we achieve the same effect by specifying to the control system a global predicate – no two processes are in the critical sections at the same time. The distributed system is then run under the control of the control system. The controller on each process would block it from entering a critical section when required and would communicate with other controllers to determine when an entry is safe. This would have the same effect as existing distributed mutual exclusion protocols.

Often, one would like to perform a safety check on an untrusted distributed system. For example, consider a distributed system of servers. Each server runs a background job that sends a periodic "I am alive" message to an external party. It is important that the external party receive the message, and so we require that the distributed system ensure that at least one server is not serving clients at any given time. However, the distributed system may be faulty, and so we place a control system on it with a specified global predicate – at least one server must be available at any time. The control system monitors the state of each server and just before the last one is about to service a client, it blocks it so that it is available for the background job. It remains blocked until the control system is sure that some other server is available.

Motivation for Off-line Predicate Control. Off-line predicate control makes the strong assumption that the future computation of the underlying distributed system is known *a priori*. Therefore, it arises in specialized scenarios such as distributed debugging and recovery where a fixed computation is being replayed.

Consider debugging the software for a distributed manufacturing system, one unit of which fills soda in bottles and another physically distributed unit places a cap on each bottle. (While debugging we may simulate the actual manufacturing system.) It is found that for a certain

computation, some executions place the cap on the bottle before filling soda. However, other executions do not. Clearly there is a synchronization bug in our system. We identify two decision points FILL and CAP in the distributed program, one on each process, where the processes decide to fill the bottle and place the cap respectively. We specify the global predicate – if we have reached CAP then FILL must have happened – to ensure that CAP happens after FILL. We also feed the trace of the computation to the control system. We now replay the computation [5] with the control system now ensuring that the decision points FILL and CAP occur in the right order. If in some executions, the system still caps the bottle before filling, then we have localized the bug to somewhere between the decision points and the output commit points. We can continue restricting the allowed executions within the computation to further localize the bug.

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 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.

Off-line Disjunctive Predicate Control. Since the general problem is NP-hard, we restrict our study to a simple, yet interesting, 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.

We solve off-line disjunctive predicate control by constructing a centralized algorithm which takes 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 that no deadlock occurs. The algorithm also determines if no control strategy exists. If there are n processes and a local predicate has a maximum of p false intervals 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.

On-line Disjunctive Predicate Control. 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 controller's ignorance of the future of the underlying computation. 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, 3, 7, 9, 13]. 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.

Applications. We previously presented real-world problems to motivate the relevance of general predicate control. In addition, we now describe applications which can benefit from our solutions for the class of disjunctive predicates.

Some examples of disjunctive predicates:

(1) $\neg cs_1 \vee \neg cs_2$ (two-process 1-mutual exclusion)

(2) $avail_1 \vee avail_2 \vee \ldots avail_n$ (at least one server is available)
(3) $(crossed\ FILL) \vee \neg(CAP)$ (FILL must happen before CAP)
(4) $think_1 \vee think_2 \vee \ldots think_n$ (at least one philosopher is thinking)
(5) $\neg cs_1 \vee \neg cs_2 \vee \ldots \neg cs_n$ ($(n-1)$-mutual exclusion)

Example (1) shows that mutual exclusion is a disjunctive predicate for two processes (however, not for more processes). Examples (2) and (3) correspond to the real-world examples that we presented for the on-line and off-line scenarios respectively. Example (4) corresponds to preventing deadlocks in the dining philosopher's problem by ensuring that at least one philosopher is thinking at all times. Example (5) is the specification of the $(n-1)$-mutual exclusion problem which allows at most $(n-1)$ processes inside the critical section at the same time.

Comparison with Related Work. Our approach of specifying a global predicate to a control system may be viewed as a way to abstract out distributed synchronization through the use of a high-level primitive. Distributed mutual-exclusion [10] and barrier synchronization are other approaches to abstracting distributed synchronization to high-level primitives. Our approach is more general because it can not only express both of these primitives but also expresses a wider range of global synchronization constraints (neither primitive can express an asymmetric constraint).

We are aware of two previous studies of controlling distributed systems to maintain classes of global predicates. One study [6] allows global properties within the class of conditional elementary restrictions [6]. 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. [11] and [14] 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 predicates and the class of disjunctive predicates. We also study both the on-line and off-line variants of the control problem.

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 asynchronous and has no shared memory. Message ordering is arbitrary.

2.2 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 [4], $\rightarrow$, 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) [2] 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.

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 has been established that $(\mathcal{G}, \leq)$ is a lattice [8].

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

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 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 blocking the process indefinitely. 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*.

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. 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*.

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.

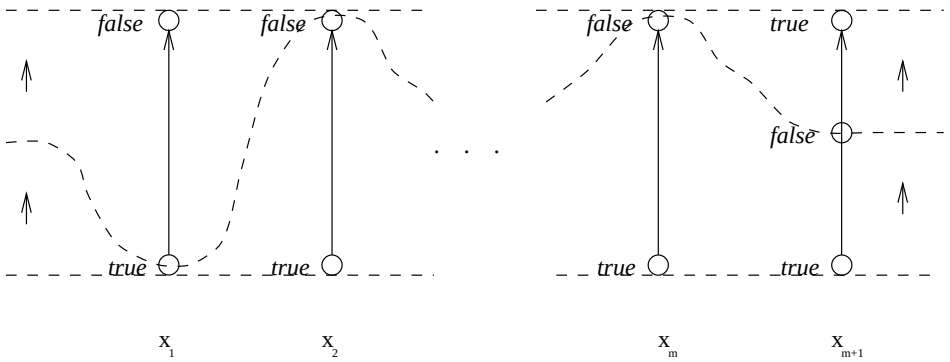

Figure 2: Proof: SGSD is NP-complete

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

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 2). We define a process for an extra boolean variable x_{m+1} which starts $true$, goes through a $false$ state, and ends $true$ again. We define $B = b \vee x_{m+1}$ and then apply SGSD to find a satisfying global sequence. If it finds one, then the global state with $x_{m+1} = false$ will have a satisfying assignment for the variables of b. Conversely, if b is satisfiable, then there must be a satisfying global sequence. □

The problem of detecting if a satisfying control strategy exists turns out to be equivalent to 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. □

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.

4 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. (Exactly as in the proof of Lemma 2.)

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

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

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 (Figure 3). So, if we have an overlapping set of false-intervals, then every global sequence must contain a global state

in which B is false and no satisfying control strategy exists. This is stated in the following result borrowed from *inevitable predicate detection* [2]:

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 such that one can be crossed before the other 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 4. g is constantly maintained as the current global state in the algorithm. For P_i we will only be concerned with local states which are $\bot_i$ or $\top_i$ or the $I_i.lo$ or $I_i.hi$ for some false-interval I_i. g will only consist of such local states. The predicates defined at the top of 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. $next(i)$ defines the next local state of interest after $g[i]$. In the algorithm, $select(Z)$ selects an arbitrary element from a non-empty set Z.

The central property of our algorithm is that it maintains a global state g to satisfy two important properties. First, no state in $next$ can causally precede a local state in g. Secondly, a state $g[i]$ must either be $\bot_i$, or it must be $true$ itself, or it must causally precede another $g[j]$ which is $true$. These two invariants are specified in the following lemma.

Lemma 4 *The algorithm in Figure 4 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 $g = \bot$.

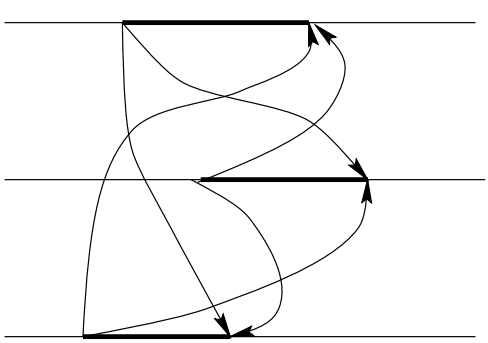

Figure 3: Overlapping intervals

Predicates:

$false(i) \equiv \exists I_i : g[i] = I_i.lo$

$true(i) \equiv \neg false(i)$

$$N(i) \equiv \begin{cases} min\ I_i :: g[i] \preceq I_i.lo & \text{if exists} \\ null & \text{if doesn't exist} \end{cases}$$

$$next(i) \equiv \begin{cases} \top_i & \text{if } N(i) = null \\ N(i).lo & \text{if } true(i) \\ N(i).hi & \text{if } false(i) \end{cases}$$

$ValidPairs() \equiv \{(i,j) : true(i) \wedge crossable(N(i), N(j))\}$

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

```
L1      while (∀i ∈ {1,...,n} : N(i) ≠ null) {          (* exit when chain reaches a ⊤_i *)
L2          if (ValidPairs() = ∅)                       (* find a true interval which
L3              exit( "No Controller Exists" );           can be maintained while
L4          ⟨ k', l ⟩ := select( ValidPairs() );          a false interval is crossed *)
L5          AddControl(C, k', k);
L6          t := N(l).hi;
L7          for (i ∈ {0,...,n})
L8              while (next(i) ->_ t)                   (* advance g
L9                  g[i] := next(i);                      consistently with → *)
L10         k := k';                                    (* remember last true interval *)
        }
L11     k' := select( {i| N(i) = null} );
L12     AddControl(C, k', k);
L13     exit(C);

Procedures:
L14     AddControl(C, k', k) {
L15         if (g[k'] = ⊥_k')
L16             C := ∅;                                 (* start chain *)
L17         else if (k ≠ k')                            (* optimization *)
L18             enqueue(C, ⟨ g[k'], next(k) ⟩);         (* add output tuple *)
        }
```

Figure 4: Algorithm for Off-line Predicate Control of Disjunctive Predicates

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 $next(i) \not\underline{\rightarrow} t$ 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 $g[i] = \bot_i$ or $\exists j : (g[i] \underline{\rightarrow} g'[j] \wedge true'(j))$. (where primed variables indicate the values before the update.) If $g[i] = \bot_i$, 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 4 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 $\top_i \underline{\not\rightarrow} t$. So in a finite number of iterations we have $next(i) \underline{\not\rightarrow} t$ 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 4 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 $g[i] \neq \bot_i$. 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 $N(k).lo \neq \bot_k$. 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 4 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 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 4 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.*

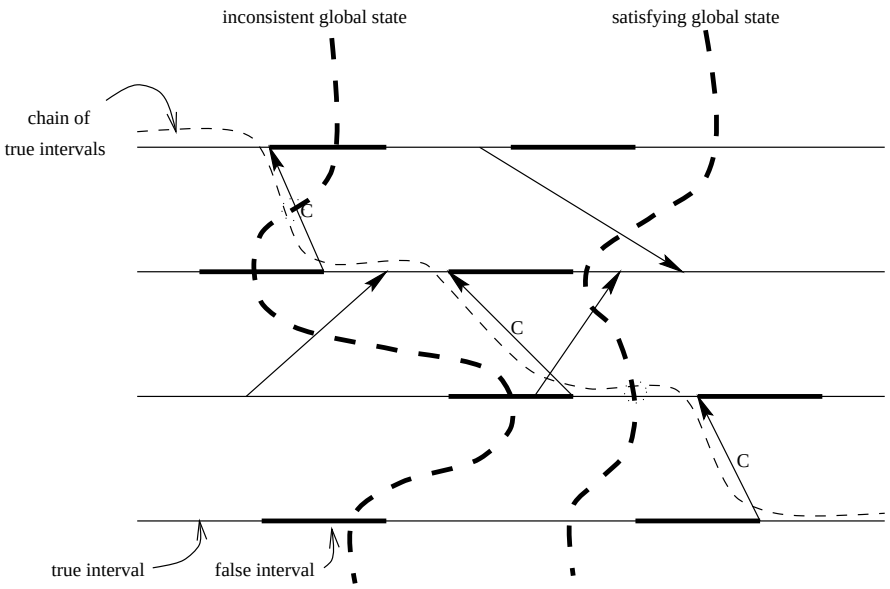

Figure 5: Proof: Lemma 8

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 $(g[k], next(k))$ [1] 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 5). 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 4 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 would avoid redundant comparisons by computing the set $ValidPairs()$ dynamically.

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

Since, in this approach, each new false-intervals would have 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 outputed 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.

5 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.

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

Theorem 3 *The On-Line Predicate Control Problem for Disjunctive Predicates is impossible to solve.*

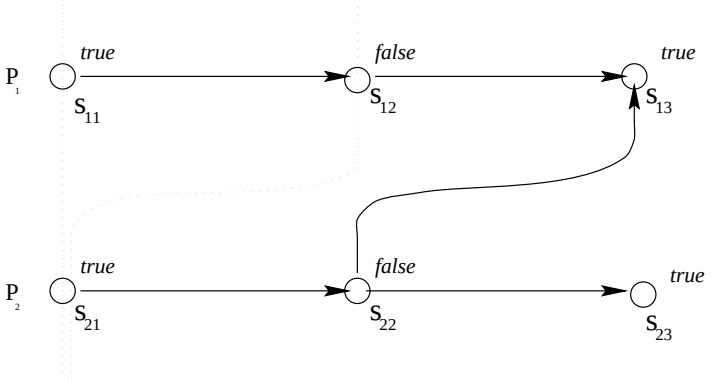

Figure 6: Impossibility of On-line Predicate Control

Proof: (by contradiction) For simplicity, we prove for the case $n = 2$. The proof is easily extensible to general n. 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,

Distributed Control Strategy for Controller, C_i:

Input:

l_i	a boolean function that takes a state as input

On-line Input:

s	current state of the underlying computation
s'	next state of the underlying computation

Variables:

$scapegoat = init(i)$	boolean
$pending = false$	boolean
j, k	integer

Control Actions:

- $scapegoat \wedge \neg l_i(s')$:
  send($req(i)$, $select(\mathcal{C})$);
  receive(ack);
  $scapegoat := false$;

- $pending \wedge l_i(s)$:
  $pending := false$;
  $scapegoat := true$;
  send(ack, C_k);

- received($req(j)$):
  if $l_i(s)$ then
  $scapegoat := true$;
  send(ack, C_j);
  else
  $pending := true$;
  $k := j$;

Definitions:

$init(i)$	*true* for one i and *false* for others
$\mathcal{C}$	set of all controllers
$select(Z)$	arbitrary element of non-empty set Z

Figure 7: Distributed Control Strategy for On-line Predicate Control with Disjunctive Predicates

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^f_{12}, s^t_{21}). (Note that (s^f_{12}, s^f_{22}) would cause B to be violated.) However, since $s^f_{22} \rightsquigarrow s^t_{13}$, 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. □

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:

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. In our server availability example, these assumptions correspond to a server not blocking during service and ending in an available state.

Our control strategy is listed in Figure 7. 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 7 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 [1, 3, 7, 9, 13] 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 [12] 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, 3, 7, 9, 13] 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 and Discussion

We have defined and motivated the predicate control problem in two scenarios – on-line and off-line. We have demonstrated 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 have also shown 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 described real-world applications of our solutions to disjunctive predicate control.

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 do not overlap. 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.

A number of issues would arise while trying to implement the control model. Controllers must have an effective way of monitoring the underlying process to detect changes in local properties. Controllers must also be able to interrupt and block an underlying process. A good programming construct for the control methodology also needs to be designed.

It is noteworthy that off-line predicate control allows us more flexibility than on-line predicate control. For example, we have shown that we can solve off-line predicate control for two-process mutual exclusion even if critical sections are allowed to block while this cannot be done on-line. Off-line scenarios of synchronization problems have not been as well-studied as on-line scenarios (because they arise less frequently). We would expect to continue to find problems that can not be solved on-line but can be solved off-line.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215 – 226. Elsevier Science Publishers B. V. (North Holland), 1989.

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

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

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

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

[13] 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.

[14] 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.