Debugging Distributed Programs Using Controlled Re-execution

Neeraj Mittal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
neerajm@cs.utexas.edu

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

ABSTRACT

Distributed programs are hard to write. A distributed debugger equipped with the mechanism to re-execute the traced computation in a controlled *fashion can greatly facilitate the detection and localization of bugs. This approach gives rise to a general problem, called* predicate control problem, *which takes a computation and a safety property specified on the computation, and outputs a controlled computation that maintains the property.*

We define a class of global predicates, called region predicates, *that can be controlled efficiently in a distributed computation. We prove that the synchronization generated by our algorithm is optimal. Further, we introduce the notion of an admissible sequence of events and prove that it is equivalent to the notion of predicate control. We then give an efficient algorithm for the class of disjunctive predicates based on the notion of an admissible sequence.*

1. INTRODUCTION

With the growth of internet, distributed systems are becoming more prevalent. However, correct distributed programs are difficult to write; they often contain *bugs* - mismatch between expected and actual computations. *Debugging* is a process of tracking down the source of such bugs. While the skill and intuition of the programmer play an important role in debugging, effective tools that provide an environment for observing and replaying computations are indispensable. Such tools, called *debuggers*, can greatly facilitate the detection and removal of the bugs.

Debuggers have been widely used for developing traditional sequential programs. However, distributed programs give rise to non-trivial issues which make traditional debuggers inadequate for the task. Firstly, unlike in sequential systems where the bug is based on an observable local state, a bug in a distributed system is often based on a global state that is not easy to observe. Secondly, even after we have detected a bug we may not be able to reproduce it due to inherent non-determinism in a distributed program, brought about by varying processor and channel speeds. Thus unobservability of global states and irreproducibility of distributed computations are the issues that need to be addressed while building a distributed debugging system. This has led to research in the detection of bugs [1, 2, 3, 4, 11, 14, 15] and the replay of distributed computations [7, 9, 13].

The correctness of a distributed program is often specified as a combination of safety and liveness properties that should hold throughout a computation. On detecting violation of a safety property, a programmer can gain considerable insight into the bug, that caused the violation, by learning whether all possible runs or executions* of the computation are unsafe. In that case, the bug cannot be fixed by adding or removing synchronization alone. On the other hand, if it is possible to eliminate unsafe executions by adding synchronization to the computation then *too little* synchronization is likely to be the problem. Further, the knowledge of the exact synchronization needed to maintain a safety property can help locate the bug in the program. The presence of such a mechanism in a debugger can greatly improve its effectiveness. The problem of controlling a computation based on the specification of safety properties on global states, referred to as the *predicate control* problem, is the focus of this paper. Informally, given a distributed computation and a global predicate, if it is possible to maintain the predicate, without violating liveness, by adding synchronization to the computation then the global predicate is controllable in the distributed computation. The synchronization involves adding an arrow from one process execution to another which ensures that the execution after the head of the arrow can proceed only after the execution before the tail has completed. For example, consider the computation in Figure 1(a). The safety property is "the clocks of no two processes drift apart more than 1 unit". The consistent cut C of the computation does not satisfy the safety property as $clock_0 = 0$ and $clock_1 = 2$ implying $|clock_0 - clock_1| = 2 > 1$. However, by adding a synchronization arrow from e_0 to f_1, thereby, forcing e_0 to occur before f_1 eliminates the consis-

*supported in part by the NSF Grants ECS-9907213, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

*each distributed computation corresponds to multiple possible executions of events.

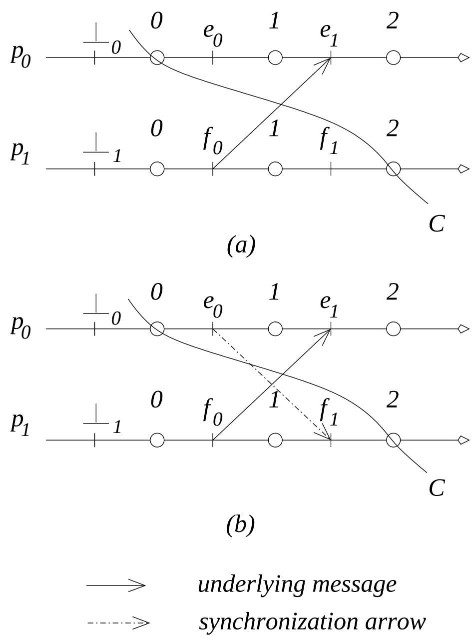

Figure 1: a computation (a) and a controlled computation (b).

tent cuts such as C that violate the given safety property.

Additionally, predicate control can be used to *actively* debug a distributed program [16]. Debugging typically involves multiple iterations of observing a distributed computation and then replaying the traced computation. Active debugging allows the traced computation to be replayed in a controlled fashion. This ability to do a controlled replay, if used judiciously, may accelerate the discovery and localization of bugs. A programmer first detects a bug while observing a certain computation. He then tries to replay the computation with added control, to determine if it would be sufficient to eliminate the bug. This control is in the form of added causal dependencies to the existing trace of the computation and is specified as a safety constraint. For example, the programmer may suspect that the bug is due to an event occurring before another event and specify the required synchronization as a safety property. The programmer may repeat the control mechanism to localize the bug further. He may also determine dependencies between the bugs so that eliminating one bug would eliminate the other. Thus a distributed debugger equipped with predicate control mechanism can prove to be a valuable tool for a programmer.

Further, predicate control has applications in the area of software fault-tolerance [17]. It has been observed that many software failures, especially those caused by synchronization faults, are *transient* in nature and may not recur when the program is re-executed with the same inputs. A common approach to achieving software fault-tolerance is based on simply rolling back the processes to a previous state and then restarting them in the hope that the transient failure will not recur in the new execution [6, 18]. Methods based on this approach rely on chance to recover from a transient software failure. However, it is possible to do better in the special case of synchronization faults. Instead of leaving the recovery to chance, controlled re-execution of the traced computation can be used to ensure that the transient synchronization failure does not occur.

The research in distributed debugging has focussed on mainly two problems: detecting bugs in a distributed computation and replaying the traced computation. In contrast, our approach focuses on adding a control mechanism to a debugger to allow computations to be run under added synchronization to satisfy safety constraints. The predicate control problem was formally introduced by Tarafdar and Garg. They proved that it is NP-complete in general. However, they solved the problem efficiently for the class of disjunctive predicates and mutual exclusion [16, 17]. Besides their work, there is another study [10] that focuses on controlling global predicates within the class of conditional elementary restrictions. Unlike our model of a distributed system, the model in [10] uses an off-line specification of pair-wise mutually exclusive states and does not use causality. Our contributions in this paper are following.

- We identify a class of global predicates, called *region predicates*, that can be controlled efficiently. The class of region predicates is fairly rich and, in some sense, a generalization of the class of stable predicates. Many stable predicates, such as termination and deadlock, belong to this class. From the point of view of predicate control, it contains channel predicates such as "there are at most k messages in any channel at any time", and fairness predicates such as "the difference between the number of times two processes are granted a resource is bounded". We give an efficient algorithm to maintain a region predicate in a computation.

- We prove that the synchronization produced by our algorithm for controlling a region predicate is optimal in the sense that it eliminates all unsafe executions and no safe execution is suppressed, thereby guaranteeing maximum concurrency possible in the controlled computation.

- We introduce the notion of an admissible sequence of events and prove that existence of such a sequence is a necessary and sufficient condition for a predicate to be controllable in a computation. Informally, given a predicate and a computation, an admissible sequence[†] attempts to capture a set of properties satisfied by some non-empty subset of the safe executions of a computation.

- Further, using the notion of an admissible sequence, we transform the problem of controlling a disjunctive predicate in a computation to finding a path in a graph. Our algorithm has $O(n^2p)$ time complexity and $O(np)$ message complexity, where n is the number of processes and p is the maximum number of true-intervals on any process. The complexities are comparable to those in [16]. We also present an algorithm that gives minimum synchronization. Our approach is more general and can be extended to find a control strategy for other classes of predicates.

[†]the sequence may not include all the events in a computation

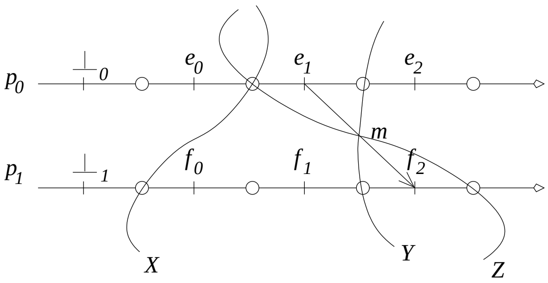

Figure 2: consistent cuts and frontiers.

The organization of the paper is as follows. We present our model of a distributed system and define the problem formally in Section 2. In Section 3, we define region predicates and give an efficient algorithm for their control. We also prove that the synchronization generated by our algorithm is optimal. We define the notion of an admissible sequence of events and prove its equivalence to the notion of predicate control in Section 4. In Section 5, we derive an efficient algorithm for the class of disjunctive predicates based on the notion of an admissible sequence.

2. MODEL AND PROBLEM SPECIFICATION

2.1 Model of a Distributed System

A distributed system consists of a set of processes $P = \{p_0, p_1, \ldots, p_{n-1}\}$. Each process executes a predefined program. Processes do not share any clock or memory; they communicate and synchronize with each other by sending messages over a set of channels. We assume that the messages are not lost, altered or spuriously introduced into a channel. We do not assume that the channels are FIFO.

The *execution* of each process in the distributed system is modeled as a sequence of distinct *events* transforming the *initial state* of the process to a *final state*. We use lowercase letters e and f to represent events, and greek letters α and β to represent sequences of events. The process on which an event e occurs is represented by $e.proc$. We use $e.pred$ and $e.succ$ to denote the previous and the next event of e, respectively, on $e.proc$, if they exist. We use the convention that if $e.succ$ does not exist then $e.succ \notin C$ evaluates to true for any set C of events. For convenience, we assume that for each process p_i there is a special event, called an *initial event* and denoted by $-_i$, that occurs before any other event on that process. Intuitively, $-_i$ initializes the state of p_i. Let $<_P$ denote the order of events on the processes.

The *computation* of a distributed system is modeled as an irreflexive partial order on a set of events. We use $E_\prec$ to denote a distributed computation with a set of events E and a partial order $\prec$, read as "precedes". We also use symbols $\lhd$, read as "before", and $\sqsubset$, read as "under" to represent irreflexive partial orders on sets of events. Let $E.- = \{-_i | i \in [1..n]\}$ be the set of initial events. We assume that E includes $E.-$ and $\prec$ includes $<_P$. Further, events in $E.-$ occur before any event in $E \setminus E.-$, i.e., for each $p_i \in P$ and $e \in E \setminus E.-$, $(-_i, e) \in \prec$, where "$\setminus$" denotes the set difference operation. For a relation $\prec$, $e \preccurlyeq f$ is equivalent to $(e = f) \vee (e \prec f)$. We use $E.\top$ to denote the set of final events on the processes. We use the terms "distributed computation" and "computation" interchangeably.

Figure 2 illustrates the various concepts introduced so far. The distributed system shown in Figure 2 consists of processes p_0 and p_1. In the figure, a circle represents a local state of a process and a bar denotes an event on a process. The events e_1 and f_2 are send and receive events, respectively, of the message m. The set of events $E = \{-_0, e_0, e_1, e_2, -_1, f_0, f_1, f_2\}$. The executions of p_0 and p_1 are given by sequences $-_0 e_0 e_1 e_2$ and $-_1 f_0 f_1 f_2$, respectively. The events e_0 and e_2 are the predecessor and the successor, respectively, of the event e_1, i.e., $e_1.pred = e_0$ and $e_1.succ = e_2$. The order of events on processes $<_P = \{(-_0, e_0), (e_0, e_1), (e_1, e_2), (-_1, f_0), (f_0, f_1), (f_1, f_2)\}^+$. Here, R^+ denotes the irreflexive transitive closure of a relation R. The partial order on the set of events E is the *happened-before* relation defined by Lamport [8], and is given by $\prec = (<_P \cup \{(-_0, f_0), (-_1, e_0), (e_1, f_2)\})^+$. Further, $E.\top = \{e_2, f_2\}$.

2.2 Consistent Cuts, Frontiers and Legal Cuts

A *cut* of a computation $E_\prec$ is a set of events C, where $E.- \subseteq C \subseteq E$, such that for each event e in C, $e.pred$ is also in C (if it exists). Formally,

$$cut(C, E_\prec) \stackrel{\text{def}}{=} (E.- \subseteq C) \wedge \langle \forall\, e : e \notin E.- : e \in C \Rightarrow e.pred \in C \rangle$$

A *frontier* of a cut C is the set of those events in C whose successors are not in C. Formally,

$$C.frontier \stackrel{\text{def}}{=} \{\, e \mid e \in C \text{ and } e.succ \notin C \,\}$$

Observe that if an event in C is also in $E.\top$ then it is trivially in $C.frontier$. A cut C *passes through* an event e iff e is contained in $C.frontier$. A cut C is *consistent* iff for each event e in C, all its preceding events are also in C. Formally,

$$consistent(C, E_\prec) \stackrel{\text{def}}{=} cut(C, E_\prec) \wedge \langle \forall\, e, f :: (e \prec f) \wedge (f \in C) \Rightarrow e \in C \rangle$$

Intuitively, a consistent cut captures the partial computation of a distributed system and its frontier captures the state of a distributed system.

In Figure 2, $X = \{-_0, e_0, -_1\}$, $Y = \{-_0, e_0, e_1, -_1, f_0, f_1\}$ and $Z = \{-_0, e_0, -_1, f_0, f_1, f_2\}$ are cuts of the computation. Here, X and Y are consistent cuts. However, Z is not consistent because $e_1 \prec f_2$ and $f_2 \in Z$ but $e_1 \notin Z$. Further, $X.frontier = \{e_0, -_1\}$, $Y.frontier = \{e_1, f_1\}$ and $Z.frontier = \{e_0, f_2\}$. Finally, X passes through events e_0 and $-_1$.

We now define a *legal cut* that helps us to capture those executions of the computation that respect the order of the events in a given sequence. Informally, if an execution (of a computation) and a sequence of events do not differ on the relative order of any two events then every consistent cut of the execution is legal with respect to the sequence. Formally, DEFINITION 1. **(legal cut)** *A consistent cut C of a computation $E_{\prec}$ is* legal *with respect to a sequence of distinct events α iff for each event α_i in α, if α_i is in C then all its preceding events in α are also in C. Formally,*

$$legal(C, E_{\prec}, \alpha) \stackrel{\text{def}}{=} consistent(C, E_{\prec}) \wedge \langle \forall\, j, k : k \leq j : \alpha_j \in C \;\Rightarrow\; \alpha_k \in C \rangle$$

In Figure 2, Y is legal with respect to sequences $e_0 f_1 f_2$ and $e_0 e_1 e_2$ but not with respect to the sequence $e_0 e_2 f_1$. We use the concept of legality to define the notion of an admissible sequence later.

2.3 Global Predicates

Let X_i be the set of variables associated with process p_i and let $X = \bigcup_i X_i$. A *global predicate* ϕ is a boolean-valued function of the variables in X. We use $\phi.C$ to denote the value of the global predicate ϕ for the cut C. If $\phi.C = true$ then C *satisfies* ϕ or ϕ is *true* for C. A global predicate ϕ is a *local predicate* of process p_i iff it only depends on the variables in X_i. We use the terms "global predicate" and "predicate" interchangeably.

2.4 Problem Specification

Informally, given a distributed computation and a global predicate, if it is possible to maintain the predicate, without violating liveness, by adding synchronization to the computation then the global predicate is controllable in the computation. The predicate is often the safety property of a distributed system. For example, "there are at most k messages in any channel at any time", "no two processes are in the critical section at the same time", or "at least one server is available at any time". The synchronization involves adding an arrow from one process execution to another which ensures that the execution after the head of the arrow can proceed only after the execution before the tail has completed. It can be realized using *control* messages. The implementation details can be found in [16]. Formally,

DEFINITION 2. **(controllable computation)** *A predicate ϕ is* controllable *in a computation $E_{\prec}$ iff there exists an irreflexive partial order $\sqsubset$ on E that extends $\prec$ (i.e., $\prec \subseteq \sqsubset$) such that every consistent cut of $E_{\sqsubset}$ satisfies ϕ.*

Each computation of a distributed system corresponds to multiple ways in which the events can be interleaved to form an execution. An execution is *safe* iff it maintains the given predicate; otherwise it is *unsafe*. The following properties about controllability of a predicate can be easily verified.

- $(\phi \;\Rightarrow\; \psi) \;\wedge\; (\phi$ is controllable in $E_{\prec}) \;\Rightarrow\; \psi$ is controllable in $E_{\prec}$.
- (ϕ is controllable in $E_{\sqsubset}$) $\wedge$ ($\prec \subseteq \sqsubset$) $\Rightarrow$ ϕ is controllable in $E_{\prec}$

The predicate control problem is NP-complete in general. However, it can be solved efficiently for certain classes of predicates including mutual exclusion and disjunctive predicates. In the next section, we introduce another class of predicates namely region predicates for which the problem can be solved in polynomial time.

3. REGION PREDICATES

The definition of a region predicate is based on p-region predicate, where p is a process. Intuitively, a p-region predicate states that, for each event e on p, there exists a minimum and a maximum consistent cut passing through e such that every consistent cut that lies between the two cuts satisfies the predicate. For example, consider the computation in Figure 3 and the fairness predicate "the difference between the number of times p_1 and p_2 are granted a resource is at most 1", i.e., $|alloc_1 - alloc_2| \leq 1$. Consider an event e on p_1 as shown in Figure 3. Immediately after execution of e, $alloc_1 = 2$. For the fairness predicate to hold for a consistent cut passing through e, $1 \leq alloc_2 \leq 3$ should be true. Note that $alloc_2$ is monotonically non-decreasing. Thus there exists an earliest event on p_2, say f_{min}, such that $alloc_2 \geq 1$. Likewise, there exists a latest event on p_2, say f_{max}, such that $alloc_2 \leq 3$. The fairness predicate holds for all consistent cuts that pass through e and an event on p_2 that lies between f_{min} and f_{max} (both inclusive). For any other consistent cut that passes through e, either $alloc_2 < 1$ or $alloc_2 > 3$, and therefore the predicate is false. Observe that the set of consistent cuts of a computation that pass through a set of events forms a lattice. Therefore there exists a minimum consistent cut $C_{min}.e$ passing through e and f_{min} that satisfies ϕ. Similarly, there exists a maximum consistent cut $C_{max}.e$ passing through e and f_{max} for which ϕ is true. Further, every consistent cut that lies between the two cuts satisfies the predicate. Note that the set of consistent cuts passing through e that satisfy the fairness predicate resembles the cross-section of an hourglass. Other examples of p_i-region predicate are,

- any local predicate on p_i.
- at most $k_{i,j}$ messages in the channel from p_i to p_j: $send_{i,j} - receive_{i,j} \leq k_{i,j}$.
- the drift between the clocks of p_i and p_j is bounded: $|clock_i - clock_j| \leq \delta_{i,j}$.
- $x_i < \min\{y_j, y_k\}$, where x_i, y_j and y_k are variables of p_i, p_j and p_k respectively. Moreover, y_j and y_k are monotonically non-decreasing.

DEFINITION 3. **(p-region predicate)** *A predicate ϕ is a* p-region predicate *iff it satisfies the following properties. For every event e on process p,*

- **(weak lattice)** $\phi.C \wedge \phi.C' \;\Rightarrow\; \phi.(C \cap C') \wedge \phi.(C \cup C')$, *where C and C' are consistent cuts that pass through e, and*
- **(weak inclusion)** $\phi.C' \wedge \phi.C'' \wedge (C' \subseteq C \subseteq C'') \;\Rightarrow\; \phi.C$, *where C, C' and C'' are consistent cuts that pass through e.*

The weak lattice property says that the set of consistent cuts passing through an event e that satisfy ϕ form a lattice,

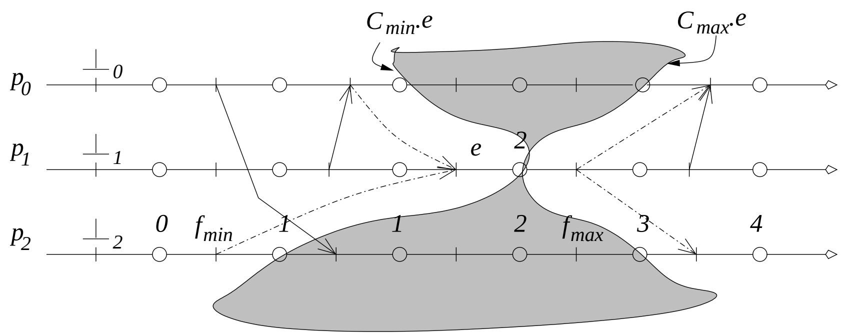

Figure 3: an illustration of a region predicate and the required synchronization.

thereby ensuring that there is a minimum and a maximum consistent cut passing through e for which ϕ is true. The weak inclusion property captures the fact the predicate holds for every consistent cut that lies between the minimum and the maximum consistent cuts. It can be easily proved that the class of p-region predicates, for a process p, is closed under conjunction

We now define a region predicate. A predicate is a *region predicate* iff it can be expressed as a conjunction of p-region predicates (possibly different p's), i.e., it can be written as $\omega_0 \wedge \omega_1 \wedge \cdots \wedge \omega_{m-1}$, where each ω_i is a p-region predicate for some process p. Since $true$ is a p-region predicate for any process p, any region predicate can be written as $\omega_0 \wedge \omega_1 \wedge \cdots \wedge \omega_{n-1}$, where each ω_i is a p_i-region predicate. Consider an event e on process p_i. We denote the minimum and maximum consistent cuts that pass through e and satisfy ω_i by $C_{min}.e$ and $C_{max}.e$, respectively. Note that if no consistent cut that passes through e satisfies ϕ then ϕ is not controllable. Therefore we assume that there is at least one consistent cut that passes through e and satisfies ϕ.

To control ω_i in a computation $E_{\prec}$, we need to ensure that the frontier of any consistent cut (of the controlled computation) always lies between $C_{min}.e$ and $C_{max}.e$, for some event e on p_i. In other words, whenever the computation reaches an event e on p_i all events in $C_{min}.e \setminus \{e\}$ have already been executed, and the computation does not advance beyond $C_{max}.e \setminus \{e\}$ before leaving e. To that effect, we add synchronization arrows from events in $(C_{min}.e).frontier$ (excluding e) to e, and from $e.succ$ to successor of events in $(C_{max}.e).frontier$ (again, excluding e). Formally, synchronization for each event e on p_i, denoted by $\lhd.e$, is defined as follows.

(D 3.1) $\lhd.e \stackrel{\text{def}}{=}$
$$\{\, (f, e) \mid f \in (C_{min}.e).frontier \setminus \{e\} \text{ and } e \notin E.- \,\}$$
$$\bigcup \{\, (e.succ, f.succ) \mid e \notin E.\top,\ f \notin E.\top \text{ and } f \in (C_{max}.e).frontier \setminus \{e\} \,\}$$

Figure 3 illustrates the synchronization for an event e. The synchronization needed to control ϕ in $E_{\prec}$, denoted by $\lhd$, is defined as $\bigcup_{e \in E} \lhd.e$. We now prove that $\lhd$ is both necessary and sufficient synchronization to control ϕ in $E_{\prec}$. Note that for ϕ to be controllable in $E_{\prec}$, it must evaluate to true for the initial consistent cut $E.-$ and the final consistent cut E. In the next lemma, we establish that the synchronization given by $\lhd$ is sufficient by proving that $\lhd$ eliminates all unsafe executions of the computation.

LEMMA 1. **($\lhd$ is sufficient)** *Let $\lhd$ be the synchronization as defined (in D 3.1) for a region predicate ϕ and a computation $E_{\prec}$. If $E.-$ and E satisfy ϕ, and $\prec \cup \lhd$ is acyclic then every consistent cut of $E_{\sqsubset}$, where $\sqsubset$ is any irreflexive partial order that extends $\prec \cup \lhd$, satisfies ϕ.*

PROOF. Consider a consistent cut C of $E_{\sqsubset}$ and an event e on some process p_i that is contained in the frontier of C. We claim that $C_{min}.e \subseteq C \subseteq C_{max}.e$.

We first show that $C_{min}.e \subseteq C$. There are two cases: $e \in E.-$ or $e \notin E.-$. If $e \in E.-$ then $C_{min}.e = E.-$. By definition of a consistent cut, $C \supseteq E.-$ which implies $C \supseteq C_{min}.e$. Therefore assume $e \notin E.-$. Let f_{min} be an event in the frontier of $C_{min}.e$ that occur on some process p_j, where $p_j \neq p_i$. By definition of $\lhd.e$, $(f_{min}, e) \in \lhd.e$ implying $(f_{min}, e) \in \lhd$. Since $\lhd \subseteq \sqsubset$, $f_{min} \sqsubset e$. Further, since C is a consistent cut of $E_{\sqsubset}$ and contains e, it also contains f_{min}. Thus every event in the frontier of $C_{min}.e$ is contained in C. Equivalently, $C_{min}.e \subseteq C$. Likewise, $C \subseteq C_{max}.e$.

This proves our claim that $C_{min}.e \subseteq C \subseteq C_{max}.e$. By definition of $C_{min}.e$ and $C_{max}.e$, $\omega_i.(C_{min}.e)$ and $\omega_i.(C_{max}.e)$ hold. Using the weak inclusion property, $\omega_i.C$ holds. Since p_i was chosen arbitrarily, for each i, $\omega_i.C$ holds. Therefore ϕ is true for C. $\square$

Lemma 2 proves that every controlled computation in which the given region predicate always holds contains $\lhd$, thereby proving that the synchronization $\lhd$ is necessary.

LEMMA 2. **($\lhd$ is necessary)** *If a region predicate ϕ is controllable in a computation $E_{\prec}$ then $E.-$ and E satisfy ϕ. Further, let $\lhd$ be the synchronization as defined (in D 3.1), and $\sqsubset$ be any irreflexive partial order that extends $\prec$ such that every consistent cut of $E_{\sqsubset}$ satisfies ϕ. Then $(\prec \cup \lhd) \subseteq \sqsubset$.*

PROOF. Since $E.-$ and E are consistent cuts of $E_{\sqsubset}$, they satisfy ϕ. We prove that $\lhd \subseteq \sqsubset$ by showing that $\lhd.e \subseteq \sqsubset$, for each event e in E. Consider an event e in E and let

$e.proc = p_i$. Further, consider events f_{min} and f_{max} in the frontiers of $C_{min}.e$ and $C_{max}.e$, respectively, that occur on some process p_j, where $p_j \neq p_i$. We show that if $(f_{min}, e) \in \lhd.e$ then $f_{min} \sqsubset e$, and if $(e.succ, f_{max}.succ) \in \lhd.e$ then $e.succ \sqsubset f_{max}.succ$.

Assume $(f_{min}, e) \in \lhd.e$. By definition of $\lhd.e$, $e \notin E.-$. There are two cases: $f_{min} \in E.-$ or $f_{min} \notin E.-$. If $f_{min} \in E.-$ then $f_{min} \prec e$ which implies $f_{min} \sqsubset e$. Therefore assume $f_{min} \notin E.-$. Further, assume, by the way of contradiction, $f_{min} \not\sqsubset e$. In that case, there exists a consistent cut of $E_{\sqsubset}$, say C, that passes through e but does not contain f_{min}. Since $\omega_i.C$ and $\omega_i.(C_{min}.e)$ hold, $\omega_i.(C \cap C_{min}.e)$ is true (weak lattice property). However, $C \cap C_{min}.e$ is strictly contained in $C_{min}.e$ as $f_{min} \notin (C \cap C_{min}.e)$ but $f_{min} \in C_{min}.e$. This contradicts the fact that $C_{min}.e$ is the minimum consistent cut that passes through e and satisfies ω_i. Similarly, if $(e.succ, f_{max}.succ) \in \lhd.e$ then $e.succ \sqsubset f_{max}.succ$.

Thus, for each e in E, $\lhd.e \subseteq \sqsubset$ implying $\lhd \subseteq \sqsubset$. Since both $\prec$ and $\lhd$ are contained in $\sqsubset$, $(\prec \cup \sqsubset) \subseteq \sqsubset$. $\square$

Theorem 3 combines Lemma 1 and Lemma 2, and gives necessary and sufficient conditions for a region predicate to be controllable.

THEOREM 3. *Let $\lhd$ be the synchronization as defined (in D 3.1) for a region predicate ϕ and a computation $E_{\prec}$. Then ϕ is controllable in $E_{\prec}$ iff (1) $E.-$ and E satisfy ϕ, and (2) $\prec \cup \lhd$ is acyclic.*

PROOF. (if) Let $\sqsubset = (\prec \cup \lhd)^+$. Since $\prec \cup \lhd$ is acyclic, $\sqsubset$ is an irreflexive partial order that extends $\prec \cup \lhd$. Using Lemma 1, every consistent cut of $E_{\sqsubset}$ satisfies ϕ. Thus ϕ is controllable in $E_{\prec}$.

(only if) If ϕ is controllable in $E_{\prec}$ then there exists an irreflexive partial order $\sqsubset$ that extends $\prec$ such that every consistent cut of $E_{\sqsubset}$ satisfies ϕ. Using Lemma 2, $\prec \cup \lhd$ is contained in $\sqsubset$. Since $\sqsubset$ is acyclic, $\prec \cup \lhd$ is acyclic. Also, again using Lemma 2, $E.-$ and E satisfy ϕ. $\square$

We now prove the optimality of our synchronization. We call a synchronization *optimal* iff it eliminates all unsafe executions but does not suppress any safe execution.

THEOREM 4. **($\lhd$ is optimal)** *Let $\lhd$ be the synchronization as defined (in D 3.1) for a region predicate ϕ and a computation $E_{\prec}$. If ϕ is controllable in $E_{\prec}$ then $\lhd$ is optimal.*

PROOF. Assume ϕ is controllable in $E_{\prec}$ and let $\sqsubset = (\prec \cup \lhd)^+$. Using Lemma 2, $\sqsubset$ is an irreflexive partial order. Further, using Lemma 1, every consistent cut of $E_{\sqsubset}$ satisfies ϕ. Thus $\sqsubset$ does not contain any unsafe execution of $E_{\prec}$. It remains to be shown that every safe execution of $E_{\prec}$ is an execution of $E_{\sqsubset}$. Every safe execution of $E_{\prec}$ can be represented by a total order on the set of events E. Let $<$ be a safe execution of $E_{\prec}$. By definition, $\prec \subseteq <$ and every consistent cut of $E_{<}$ satisfies ϕ. Thus, using Lemma 2, $<$ contains $\prec \cup \lhd$. This implies $<$ extends $\sqsubset$ or, in other words, $<$ is an execution of $E_{\sqsubset}$. $\square$

Theorem 3 gives us an efficient way to compute the synchronization needed to control a region predicate in a computation provided we can efficiently compute $C_{min}.e$ and $C_{max}.e$ for each event e. We show that a region predicate satisfies the *linearity* property which gives us an efficient way to compute $C_{min}.e$ and $C_{max}.e$ for each event e in E. Let $e.proc = p_i$ and C be a consistent cut of $E_{\prec}$ that passes through e. The linearity property demands that if ω_i evaluates to false for C then there exists an event f in $C.frontier$, different from e, such that f cannot be a part of the frontier of any consistent cut of $E_{\prec}$ passing through e that satisfies ω_i. Formally,

$$\begin{aligned} \neg\omega_i.C \ \Rightarrow\ \langle \exists f : f \in C.frontier \setminus \{e\} : \\ \langle \nexists C' : C' \text{ is a consistent cut of } E_{\prec} : \\ \omega_i.C' \text{ and } C' \text{ passes through } e \text{ and } f \rangle\rangle \end{aligned}$$

THEOREM 5. *A region predicate satisfies the linearity property.*

PROOF. Let ϕ be a region predicate of a computation $E_{\prec}$. Consider a consistent cut C of $E_{\prec}$. Let e be an event in the frontier of C and $e.proc = p_i$. Assume ω_i evaluates to false for C, and, on the contrary, for each $f \in C.frontier \setminus \{e\}$ there exists a consistent cut of $E_{\prec}$, say C_f, that passes through e and f, and satisfies ω_i. Consider the cuts C_{min} and C_{max} defined as the intersection and the union, respectively, of all C_f's. Observe that C_{min} and C_{max} are consistent cuts of $E_{\prec}$ that pass through e, and $C_{min} \subseteq C \subseteq C_{max}$. Further, using the weak lattice property, $\omega_i.(C_{min})$ and $\omega_i.(C_{max})$ hold as ω_i is true for all C_f's. Thus, using the weak inclusion property, $\omega_i.C$ also holds, a contradiction. $\square$

Figure 4 gives an efficient algorithm to compute $C_{min}.e$ for an event e, given a region predicate ϕ and a computation $E_{\prec}$. The algorithm to compute $C_{max}.e$ is similar and has been omitted. It is easy to see that given a region predicate ϕ and a computation $E_{\prec}$, the complexity of computing $C_{min}.e$ and $C_{max}.e$, for each event e, is $O(|\phi| \cdot |E|^2)$ assuming the time complexity of invoking the linearity property every time is $O(|\phi|)$. Figure 5 describes the algorithm to determine whether a region predicate is controllable in a computation. The algorithm has $O(|\phi| \cdot |E|^2)$ time complexity.

Remark: Note that the class of region predicates is incomparable to the class of linear and post-linear predicates defined by Chase and Garg in [2]. Let C_{ϕ} denote the set of all consistent cuts for which ϕ is true. If ϕ is a linear predicate, C_{ϕ} is an inf-semilattice. Similarly, if ϕ is a post-linear predicate, C_{ϕ} is a sup-semilattice. However, if ϕ is a region predicate, C_{ϕ} may neither be an inf-semilattice or a sup-semilattice. In particular, it is not necessary that $C_{min}.e \subseteq C_{min}.(e.succ)$ or $C_{max}.e \subseteq C_{max}.(e.succ)$. For example, C_{ϕ} when ϕ is "$x < y$", where y is a monotonically

given a computation $E_{\prec}$, a p-region predicate ϕ, and
an event e on process p:

C_{min} := minimum consistent cut that passes through e;

while not done do
  if there exists an event f in $C_{min}.frontier$
    such that $e.succ \prec f$ then
      exit("$C_{min}.e$ does not exist");
  endif;
  if there exists events f and g, $f \neq e$, in $C.frontier$
    such that $f.succ \prec g$ then
      $C_{min} := C_{min} \cup f.succ$;
  else
      /* C_{min} is a consistent cut */
      if $\phi.C$ then exit(C_{min});
      else
        find the event f using linearity property;
        $C_{min} := C_{min} \cup f.succ$;
      endif;
  endif;
endwhile;

Figure 4: an algorithm to compute $C_{min}.e$ for an event e.

given a computation $E_{\prec}$ and a region predicate ϕ:

1. if $E.\bot$ or E does not satisfy ϕ then
      exit("ϕ cannot be controlled in $E_{\prec}$");
2. for each event $e \in E$ do compute $C_{min}.e$ and $C_{max}.e$;
3. compute the synchronization $\lhd$ as defined (in D 3.1);
4. if $\prec \cup \lhd$ is acyclic then exit($\lhd$)
   else exit("ϕ cannot be controlled in $E_{\prec}$");

Figure 5: the algorithm to determine if a region predicate is controllable in a computation.

non-decreasing variable of p_j and x is a non-monotonic variable of p_i, does not form an inf-semilattice.

4. ADMISSIBLE SEQUENCES

In this section, we give an alternative characterization of controllability based on the notion of an admissible sequence. Informally, given a predicate ϕ and a computation $E_{\prec}$, an admissible sequence of events α tries to capture the set of properties satisfied by some non-empty subset S of the safe executions of $E_{\prec}$. Each execution in S traverses through a set of phases. The i^{th} phase starts when α_i is executed and continues until the execution on $\alpha_i.proc$ advances beyond α_i, i.e., $\alpha_i.succ$ is executed. Each execution in S satisfies the following properties. Firstly, for each i, the execution enters the i^{th} phase before the $(i+1)^{st}$ phase. For this to hold, α and $E_{\prec}$ cannot differ on relative order of any two events (agreement property). Secondly, there are no gaps in the traversal of phases implying (1) the initial consistent cut $E.-$ and the final consistent cut E belong to at least one phase (possibly different) (boundary condition), and (2) for each i, the $(i+1)^{st}$ phase is entered before leaving the i^{th} phase (continuity property). Finally, to ensure that no unsafe execution satisfies these properties, all consistent cuts of the computation that are legal with respect to α and belong to at least one phase satisfy the given predicate (weak safety property). Let $|\alpha|$ denote the length of a sequence α. Formally,

DEFINITION 4. **(admissible sequence)** *A sequence of distinct events α is* admissible *with respect to a predicate ϕ and a computation $E_{\prec}$ iff it satisfies the following properties.*

- **(agreement)** *α is consistent with $\prec$, i.e., for each i and j, $i < j \Rightarrow \alpha_j \not\prec \alpha_i$,*
- **(boundary condition)** *$\alpha_0 \in E.-$ and $\alpha_{|\alpha|-1} \in E.\top$,*
- **(continuity)** *for each i, $\alpha_i \notin E.\top \Rightarrow \alpha_i.succ \not\prec \alpha_{i+1}$, and*
- **(weak safety)** *any cut C that is legal with respect to α such that $\alpha_i \in C.frontier$, for some i, satisfies ϕ, i.e., $legal(C, E_{\prec}, \alpha) \wedge \langle \exists \alpha_i :: \alpha_i \in C.frontier \rangle \Rightarrow \phi.C$.*

For example, in Figure 6, the sequence $e_0 e_1 f_2$ is not an admissible sequence as the initial consistent, given by $\{-_0, -_1\}$, does not belong to any phase (the boundary condition is violated). The sequence $-_0 f_1 e_0 f_2$ is not admissible since every execution of the computation executes e_0 before f_1, thereby entering the 2^{nd} phase before the 1^{st} phase (the agreement property is not satisfied). The sequence $-_0 e_0 f_1 f_2$ is not an admissible sequence because every execution of the computation must execute e_1 (and therefore leave e_0) before it can execute f_1, thereby leaving the 1^{st} phase before entering the 2^{nd} phase (the continuity property is violated). Finally, the sequence $-_0 e_0 f_0 e_2$ satisfies the boundary condition, and the agreement and continuity properties.

We now show the equivalence of the notion of an admissible sequence and the notion of controllability. In the next theorem, we prove that the existence of an admissible sequence is a necessary condition for controllability by showing that every safe execution of a computation constitutes an admissible sequence.

THEOREM 6. *If a predicate ϕ is controllable in a computation $E_{\prec}$ then there exists an admissible sequence of events with respect to ϕ and $E_{\prec}$.*

PROOF. Let α be any safe execution of $E_{\prec}$, and $<$ be the total order of events given by α. By construction, α satisfies the agreement property and the boundary condition. Assume, by the way of contradiction, α violates the continuity property. Therefore, for some i, $\alpha_i \prec \alpha_i.succ \prec \alpha_{i+1}$. Since $\prec \subseteq <$, $\alpha_i < \alpha_i.succ < \alpha_{i+1}$. Further, since α contains all events of E, $\alpha_i.succ \in \alpha$. Let $\alpha_j = \alpha_i.succ$. We have $i < j < i+1$, a contradiction. Therefore α satisfies the continuity property. Finally, consider a consistent cut C of $E_{\prec}$ that satisfies the antecedent of the weak safety property. In particular, C is legal with respect to α which implies C is a consistent cut of α (or $E_{<}$). Since α is a safe execution, ϕ is true for C. $\square$

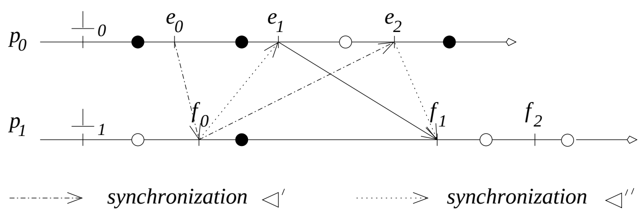

Figure 6: an illustration of the synchronization corresponding to an admissible sequence.

Next, we show that existence of an admissible sequence is a sufficient condition for a given predicate to be controllable. To do so, we first give the synchronization needed to be added to a computation so as to suppress all its unsafe executions. We then prove the synchronization does not eliminate all safe executions. Formally, given an admissible sequence of events α, the required synchronization consists of two types of arrows, denoted by $\lhd'$ and $\lhd''$, defined as follows.

$$\textbf{(D 4.1)} \quad \lhd' \stackrel{\text{def}}{=} \{ (\alpha_i, \alpha_j) \mid 0 \le i < j < |\alpha| \}, \text{ and}$$

$$\textbf{(D 4.2)} \quad \lhd'' \stackrel{\text{def}}{=} \{ (\alpha_{i+1}, \alpha_i.succ) \mid 0 \le i < |\alpha| - 1, \; \alpha_i \notin E.\top \text{ and } \alpha_i.proc \neq \alpha_{i+1}.proc \}$$

Figure 6 illustrates the synchronization for the admissible sequence $\perp_0 e_0 f_0 e_2$. The synchronization given by $\lhd'$ ensures that every execution of the controlled computation enters the phases in the correct order. The synchronization given by $\lhd''$ guarantees that, for each i, every execution of the controlled computation enters the $(i+1)^{st}$ phase before it leaves the i^{th} phase.

THEOREM 7. *If there exists an admissible sequence of events with respect to a predicate ϕ and a computation $E_\prec$ then ϕ is controllable in $E_\prec$.*

PROOF. Let α be an admissible sequence of events with respect to ϕ and $E_\prec$. As explained before, we add the synchronization given by $\lhd'$ (defined in D 4.1) and $\lhd''$ (defined in D 4.2) to $E_\prec$. The proof then reduces to showing that (1) the added synchronization does not create any deadlocks, i.e., there are no cycles, (2) every consistent cut of the controlled computation is legal with respect to α, and (3) every consistent cut of the controlled computation contains at least one event from α in its frontier. Due to the lack of space, the proof is presented elsewhere [12]. $\square$

THEOREM 8. *A predicate ϕ is controllable in a computation $E_\prec$ iff there exists an admissible sequence of events with respect to ϕ and $E_\prec$.*

5. DISJUNCTIVE PREDICATES

In this section, we give an efficient algorithm to solve the predicate control problem for the class of disjunctive predicates. Our algorithm is based on the notion of an admissible sequence introduced in the previous section. Intuitively, a disjunctive predicate states that at least one local condition must be met at all times, or, in other words, a bad combination of local conditions does not occur. For example,

- at least one server is available at all times: $avail_0 \vee avail_1 \vee \cdots \vee avail_{n-1}$.
- two process mutual exclusion: $\neg cs_0 \vee \neg cs_1$.
- at least one philosopher is thinking at any time: $think_0 \vee think_1 \vee \cdots \vee think_{n-1}$.

The special case of k-mutual exclusion problem, when $k = n-1$, belongs to the class of disjunctive predicates. Formally, a global predicate is a *disjunctive predicate* iff it can be expressed as a disjunction of local predicates, i.e., it can be written as $l_0 \vee l_1 \vee \cdots \vee l_{m-1}$, where each l_i is a local predicate of some process. Observe that *false* is a local predicate of any process. Thus any disjunctive predicate can be written as $l_0 \vee l_1 \vee \cdots \vee l_{n-1}$, where each l_i is a local predicate of p_i.

Let ϕ be a disjunctive predicate and $E_\prec$ be a computation. Given an event e on a process p_i, since l_i is a local predicate of p_i, we can calculate the value of l_i for e. An event e on a process p_i is a *true* event iff $l_i.e$ evaluates to true. To compute an admissible sequence of events, we construct a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, called "true event graph" (TEG), as follows. There is a vertex in the graph for each true event in E. Further, there is an edge from vertex e to vertex f iff $e.succ \not\prec f$. The vertex e is labeled as "initial" iff $e \in E.\perp$. Similarly, the vertex e is labeled as "final" iff $e \in E.\top$. We call a path in the graph as *permissible* iff it starts from a vertex labeled "initial" and ends at a vertex labeled "final". We show that there exists a permissible path in $\mathcal{G}$ iff ϕ is controllable in $E_\prec$. Note that there is a one-to-one correspondence between paths in the graph and sequences of true events that satisfy the continuity property. Hereafter, we use them interchangeably. Due to the semantics of disjunction, every path satisfies the weak safety property. Further, by definition, every permissible path satisfies the boundary condition. In the next lemma, we prove that the shortest permissible path satisfies the agreement property.

THEOREM 9. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the TEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$. The shortest permissible path in $\mathcal{G}$, if it exists, corresponds to an admissible sequence of events.*

PROOF. Assume there exists a permissible path in $\mathcal{G}$. Let $\pi = \pi_0 \pi_1 \cdots \pi_{m-1}$ be the shortest permissible path. As argued before, π satisfies the boundary condition, and the continuity and weak safety properties. Assume, by the way of contradiction, π does not respect $\prec$. Therefore there exist vertices π_i and π_j, $i < j$, such that $\pi_j \prec \pi_i$. Note that π_i cannot be an "initial" vertex implying $i > 0$. Since π is the shortest permissible path, there is no edge from π_{i-1} to π_j, otherwise we have a shorter permissible path namely $\pi_0 \pi_1 \cdots \pi_{i-1} \pi_j \cdots \pi_{m-1}$, a contradiction. An absence of edge from π_{i-1} to π_j implies $\pi_{i-1}.succ \prec \pi_j$. Since $\pi_j \prec \pi_i$, $\pi_{i-1}.succ \prec \pi_i$, thereby precluding an edge from π_{i-1} to π_i, a contradiction. Thus π respects $\prec$. In other words, π satisfies the agreement property. $\square$

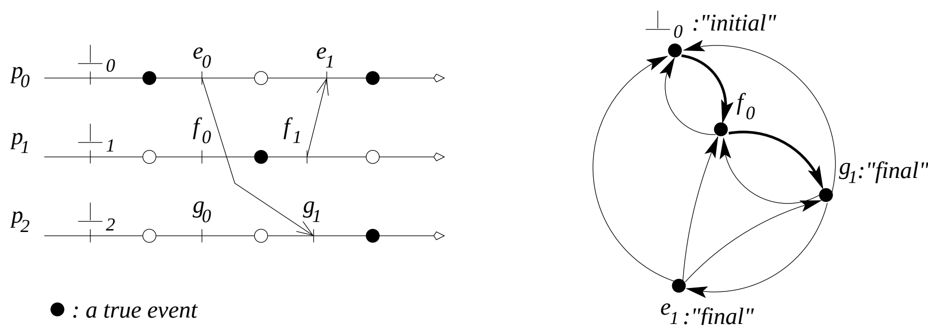

Figure 7: a computation and its corresponding TEG.

We next show that if the given disjunctive predicate is controllable in a computation then there exists a permissible path in the graph.

THEOREM 10. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the TEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$. If ϕ is controllable in $E_\prec$ then there exists a permissible path in $\mathcal{G}$.*

PROOF. Assume ϕ is controllable in $E_\prec$. Therefore there exists an irreflexive partial order $\sqsubset$ that extends $\prec$ such that every consistent cut of $E_\sqsubset$ satisfies ϕ. Without loss of generality, assume $\sqsubset$ is a total order. Further, there exists a vertex labeled "initial" in $\mathcal{G}$, otherwise ϕ evaluates to false for $E.-$. Let π_0 denote such a vertex. Starting from π_0, we construct a permissible path π by adding vertices to the path constructed as yet until we reach a "final" vertex.

Let π_i denote the last vertex reached in the path so far. If π_i is labeled "final", we have a permissible path. Therefore assume π_i is not labeled "final" implying $\pi_i.succ$ exists. Observe that events in any consistent cut of $E_\sqsubset$ are totally ordered because $\sqsubset$ is a total order. Let C_i be the consistent cut of $E_\sqsubset$ such that $\pi_i.succ$ is the last event in the cut. We set π_{i+1} to any true event in the frontier of C_i. Since ϕ satisfies C_i, π_{i+1} exists. Note that $\pi_{i+1} \sqsubseteq \pi_i.succ$ because $\pi_i.succ$ and π_{i+1} are events contained in C_i, and $\pi_i.succ$ is the last event in C_i. Therefore $\pi_i.succ \not\sqsubset \pi_{i+1}$. Since $\prec \subseteq \sqsubset$, $\pi_i.succ \not\prec \pi_{i+1}$ which implies there is an edge from π_i to π_{i+1} in $\mathcal{G}$.

Finally, we need to prove that a vertex labeled "final" is eventually added to the path. Observe that, for each i, $\pi_{i+1}.succ \not\sqsubseteq \pi_i.succ$. This is so because neither π_i nor $\pi_{i+1}.succ$ belong to $C_i.frontier$. Therefore $\pi_i.succ \sqsubset \pi_{i+1}$ implying $C_i \subsetneq C_{i+1}$. This implies that no vertex is revisited while constructing the path. Since number of vertices are finite, a vertex labeled "final" is eventually reached. $\square$

THEOREM 11. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the TEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$. Then ϕ is controllable in $E_\prec$ iff there exists a permissible path in $\mathcal{G}$.*

The previous algorithm can be easily modified to give an admissible sequence that generates *minimum* synchronization. To do so, we assign a weight to each edge in the TEG as follows.

$$w.(e, f) \stackrel{\text{def}}{=} \begin{cases} (0,1) & \text{if } f \preccurlyeq e.succ \\ (1,1) & \text{otherwise} \end{cases}$$

Here, $w.(e, f)$ denotes the weight of edge (e, f). Two weights are compared using lexicographic ordering and added by performing component-wise addition. Note that an admissible sequence generated from a TEG consists only of true events. As a result, we do not need to add the synchronization given by $\lhd'$ (defined in D 4.1) to a computation in order to control a disjunctive predicate. The synchronization given by $\lhd''$ (defined in D 4.3) suffices. This is because the admissible sequence constructed from a TEG satisfies a stronger property than the weak safety property namely $\langle \exists \alpha_i :: \alpha_i \in C_i.frontier \rangle \Rightarrow \phi.C$.

We prove elsewhere [12] that the shortest permissible path the weighted TEG (WTEG) not only constitutes an admissible sequence of events but also gives minimum synchronization. This is important in scenarios where the bandwidth is limited and the number of control messages need to be minimized. Intuitively, the first entry in the weight of an edge (e, f) indicates whether $\prec$ subsumes the synchronization arrow from the event f to the event $e.succ$. The shortest permissible path in a WTEG, therefore, corresponds to a path that minimizes two things. Firstly, it minimizes the number of synchronization arrows in $\lhd''$ that are not contained in $\prec$, i.e., $| \lhd'' \setminus \prec |$. Secondly, among all paths that minimize $| \lhd'' \setminus \prec |$, it gives the path with the smallest number of edges, i.e., the path that minimizes $| \lhd'' |$.

The algorithms presented here have $O(n^2m^2)$ time complexity, where n is the number of processes and m is the maximum number of true events on any process. That is because, in the worst case, there can be as many as $O(nm)$ vertices and $O(n^2m^2)$ edges in TEG. To reduce the number of edges in the graph, we observe that if there is an edge from vertex e to vertex f then there is an edge from vertex g to vertex h for each g and h such that $e \leqslant_P g$ and $h \leqslant_P f$. Thus Theorem 11 still holds if, for each event e and process p, we put an edge from the vertex e to the vertex corresponding to the last event f on p such that $e.succ \not\prec f$. This ensures that there are at most $O(n^2m)$ edges in the graph. Further, by using *true-intervals*[‡] instead of true events to construct

[‡]a true-interval is a sequence of contiguous true events on a process

the graph, we can reduce the time complexity to $O(n^2p)$, where p is the maximum number of true-intervals on any process.

6. CONCLUSIONS AND FUTURE WORK

A distributed debugger equipped with the mechanism to re-execute a traced computation under control can greatly facilitate the detection and localization of bugs. This approach gives rise to predicate control problem. However, the predicate control problem was proved to be NP-complete in general by Tarafdar and Garg. They developed efficient algorithms for the class of disjunctive predicates and mutual exclusion. We extend their work in two ways. Firstly, we define a class of predicates called region predicates that includes channel predicates such as "there are at most k message in any channel at any time", and fairness predicates such as "the difference between the number of times two processes are granted a resource is bounded". We give an efficient algorithm to compute the synchronization needed to control a region predicates in a traced computation. We also prove that the synchronization given by our algorithm guarantees maximum concurrency in the controlled computation. Further, we introduce the notion of an admissible sequence of events and prove its equivalence to the notion of predicate control. Using this notion, we reduce the problem of determining the synchronization for a computation, given a disjunctive predicate, to finding a path in a graph. We also give an algorithm that minimizes the number of synchronization arrows (or control messages) in the controlled computation.

We have extended the notion of an admissible sequence of events to the notion of an admissible sequence of sub-frontiers. A sub-frontier is a set of events that can be a part of the frontier of some consistent cut. Based on this notion, we have developed algorithms for the class of k-disjunctive predicates - predicates that can be expressed as a disjunction of predicates, where each disjunct spans at most k processes.

7. REFERENCES

[1] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] Craig Chase and Vijay K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[3] R. Cooper and Keith Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[4] R. Cypher and E. Leu. Efficient Race Detection for Message-Passing Programs with Nonblocking Sends and Recieves. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 534–541, 1995.

[5] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[6] Y. Huang and Chandra Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *Proceedings of IEEE Fault-Tolerant Computing Symposium*, pages 138–144, June 1993.

[7] Richard Kilgore and Craig Chase. Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages. In *Proceedings of the International Conference on System Sciences*, Hawaii, January 1997.

[8] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[10] A. Maggiolo-Schettini, H. Welde, and J. Winkowski. Modeling a Solution for a Control Problem in Distributed Systems by Restrictions. *Theoretical Computer Science*, 13(1):61–83, January 1981.

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

[12] Neeraj Mittal and Vijay K. Garg. Debugging Distributed Programs Using Controlled Re-execution. Technical Report TR-PDS-2000-002, Parallel and Distributed Systems Group, The University of Texas at Austin, 2000.

[13] R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debgging Message-Passing Programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[14] Scott D. Stoller and Yanhong A. Liu. Efficient Symbolic Detection of Global Properties in Distributed Systems. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, pages 357–368, 1998.

[15] K. Tai. Race Analysis of Traces of Asynchronous Message-Passing Programs. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, pages 261–268, 1997.

[16] Ashis Tarafdar and Vijay K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, 1998.

[17] Ashis Tarafdar and Vijay K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[18] Y. M. Wang, Y. Huang, W. K. Fuchs, Chandra Kintala, and Gaurav Suri. Progressive Retry for Software Failure Recovery in Message-Passing Applications. *IEEE Transactions on Computers*, 46(10):1137–1141, October 1997.