Debugging Distributed Programs Using Controlled Re-execution

Neeraj Mittal and Vijay K. Garg

TR-PDS-2000-002   May   2000

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

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 on 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

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

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 [CL85, CG98, CM91, CL95, HPR93, MC88, SL98, Tai97] and the replay of distributed computations [KC97, LMC87, NM95].

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

Additionally, predicate control can be used to *actively* debug a distributed program [TG98]. 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

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

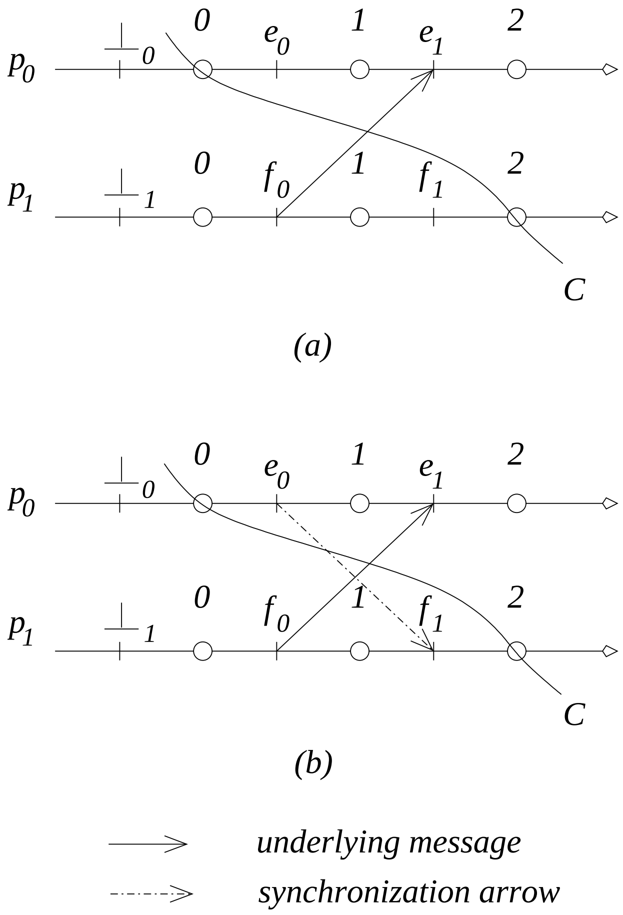

Figure 1: a computation (a) and a controlled computation (b).

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 [TG99]. 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 [HK93, WHF$^+$97]. 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 [TG98, TG99]. Besides their work, there is another study [MSWW81] that focuses on controlling global predicates within the class of conditional elementary restrictions. Unlike our model of a distributed system, the model in [MSWW81] 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 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 to control a predicate in a computation. Informally, given a predicate and a computation, an admissible sequence[†] attempts to capture the set of properties satisfied by some non-empty subset of the safe executions of the 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 $O(n^2t)$ time complexity and $O(nt)$ message complexity, where n is the number of processes and t is the maximum number of true-intervals on any process. The complexities are comparable to those in [TG98]. We also present an algorithm that gives minimum

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

synchronization. Our approach is more general and can be extended to find a control strategy for other classes of predicates.

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 $\bot_i$, that occurs before any other event on that process. Intuitively, $\bot_i$ initializes the state of p_i. Let $<_P$ denote the order of events on the processes. In our model, a *local state* of a process can be captured by the prefix of events on the process. A prefix of events can be uniquely characterized by the last event in the prefix. Thus there is a one-to-one correspondence between an event and a local state.

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", $\sqsubset$, read as "under", and $<$, read as "less than", to represent irreflexive partial orders on sets of events. Let $E.\bot = \{ \bot_i \mid i \in [1..n] \}$ be the set of initial events. We assume that E includes $E.\bot$ and $\prec$ includes $<_P$. Further, events in $E.\bot$ occur before any event in $E \setminus E.\bot$, that is, for each $p_i \in P$ and $e \in E \setminus E.\bot$, $(\bot_i, e) \in \prec$, 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.

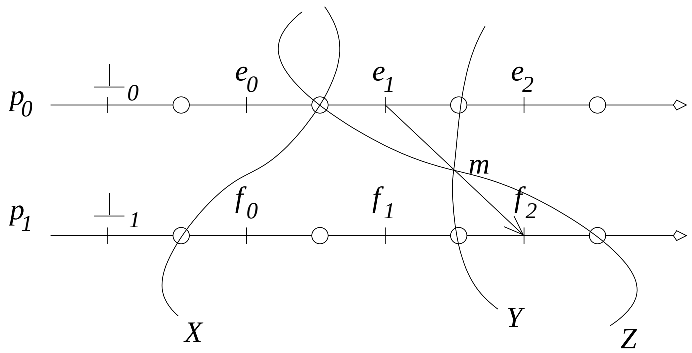

Figure 2: consistent cuts and frontiers.

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 $E = \{\perp_0, e_0, e_1, e_2, \perp_1, f_0, f_1, f_2\}$. The executions of p_0 and p_1 are given by sequences $\perp_0 e_0 e_1 e_2$ and $\perp_1 f_0 f_1 f_2$, respectively. The events e_0 and e_2 are the predecessor and the successor, respectively, of the event e_1, that is, $e_1.pred = e_0$ and $e_1.succ = e_2$. The order of events on processes $<_P = \{(\perp_0, e_0), (e_0, e_1), (e_1, e_2), (\perp_1, f_0), (f_0, f_1), (f_1, f_2)\}^+$. Here, R^+ denotes the irreflexive transitive closure of a relation R. The partial order on E, $\prec = (<_P \cup \{(\perp_0, f_0), (\perp_1, e_0), (e_1, f_2)\})^+$, the *happened-before* relation defined by Lamport [Lam78]. 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 $E.\perp \subseteq C \subseteq E$, such that for each event e in C, $e.pred$ is also in C (if it exists). Formally,

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

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 the distributed system. For a cut C, $C.p_i$ denotes the set of events in C that occur on process p_i.

In Figure 2, $X = \{\bot_0, e_0, \bot_1\}$, $Y = \{\bot_0, e_0, e_1, \bot_1, f_0, f_1\}$ and $Z = \{\bot_0, e_0, \bot_1, f_0, f_1, f_2\}$ are cuts of the computation. Here, X and Y are consistent cuts. However, Z is not consistent since $e_1 \prec f_2$ and $f_2 \in Z$ but $e_1 \notin Z$. Further, $X.\mathit{frontier} = \{e_0, \bot_1\}$, $Y.\mathit{frontier} = \{e_1, f_1\}$ and $Z.\mathit{frontier} = \{e_0, f_2\}$. $X.p_0 = \{\bot_0, e_0\}$ and $X.p_1 = \{\bot_1\}$. Finally, X passes through events e_0 and $\bot_1$.

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,*

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

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 $\phi.C = \mathit{true}$ 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 [TG98]. 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 $\prec$ (that is, $\prec \subseteq \sqsubset$) 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 $E_{\prec}$.
- $(\phi$ is controllable in $E_{\prec}) \vee (\psi$ is controllable in $E_{\prec}) \Rightarrow \phi \vee \psi$ is controllable in $E_{\prec}$.
- $(\phi$ is controllable in $E_{\sqsubset}) \wedge (\prec \subseteq \sqsubset) \Rightarrow \phi$ is controllable in $E_{\prec}$.
- ϕ is an invariant in $E_{\prec} \Rightarrow \phi$ 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", that is, $|alloc_1 - alloc_2| \leq 1$. 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.

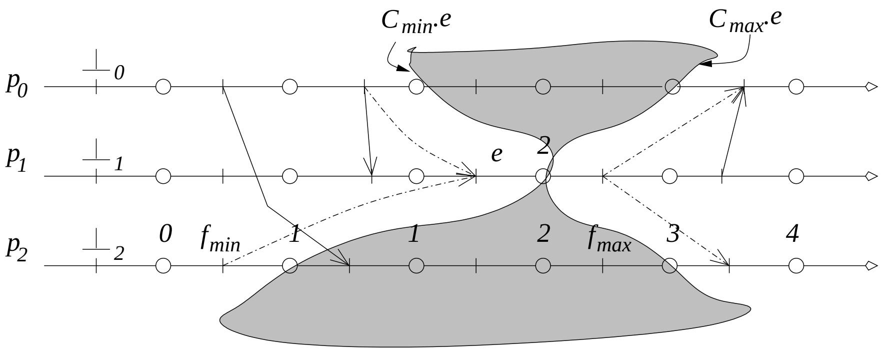

Figure 3: an illustration of a region predicate and the required synchronization.

- 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, 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 (Theorem 20 in Appendix A).

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), that is, 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 ω_i then ϕ

is not controllable. Therefore we assume that there is at least one consistent cut that passes through e and satisfies ω_i.

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:

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

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 $E.\bot$ 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 $E.\bot$ 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: $e \in E.\bot$ or $e \notin E.\bot$. If $e \in E.\bot$ then $C_{min}.e = E.\bot$. By definition of a consistent cut, $C \supseteq E.\bot$ which implies $C \supseteq C_{min}.e$. Therefore assume $e \notin E.\bot$. 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. ∎

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 $E.\bot$ 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 $E.\bot$ 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 $\lhd.e$, $e \notin E.\bot$. There are two cases: $f_{min} \in E.\bot$ or $f_{min} \notin E.\bot$. If $f_{min} \in E.\bot$ then $f_{min} \prec e$ which implies $f_{min} \sqsubset e$. Therefore assume $f_{min} \notin E.\bot$. 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 $\sqsubset$, $(\prec \cup \sqsubset) \subseteq \sqsubset$. ∎

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) $E.\bot$ 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, $E.\bot$ and E satisfy ϕ. ∎

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, $<$ extends $\prec \cup \lhd$. This implies $<$ extends $\sqsubset$ or, in other words, $<$ is an execution of $E_{\sqsubset}$. ∎

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,

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

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. ∎

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 [CG98]. 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,

```
given a computation E≺, 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 ≺ f then
            exit("C_min.e does not exist");
    endif;
    if there exist events f and g, f ≠ e, in C_min.frontier such that f.succ ≺ g then
            C_min := C_min ∪ f.succ;
    else
            /* C_min is a consistent cut */
            if φ.C_min then exit(C_min);
            else
                find the event f using the linearity property;
                if f is the last event on its process then
                    exit("C_min.e does not exist");
                else
                    C_min := C_min ∪ f.succ;
                endif;
            endif;
    endif;
endwhile;
```

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

```
given a computation E≺ and a region predicate φ:

1. if E.⊥ or E does not satisfy φ then exit("φ cannot be controlled in E≺");
2. for each event e ∈ E do compute C_min.e and C_max.e;
3. compute the synchronization ◁ as defined (in D 3.1);
4. if ≺ ∪ ◁ is acyclic then exit(◁) else exit("φ cannot be controlled in E≺");
```

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

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 $C_{max}.e \subseteq C_{max}.(e.succ)$. In Figure 6, $X = C_{min}.e$ and $Y = C_{min}.(e.succ)$ satisfy ϕ but $Z = X \cap Y$ does not. For example, C_ϕ when ϕ is "$x < y$", where y is a monotonically 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}

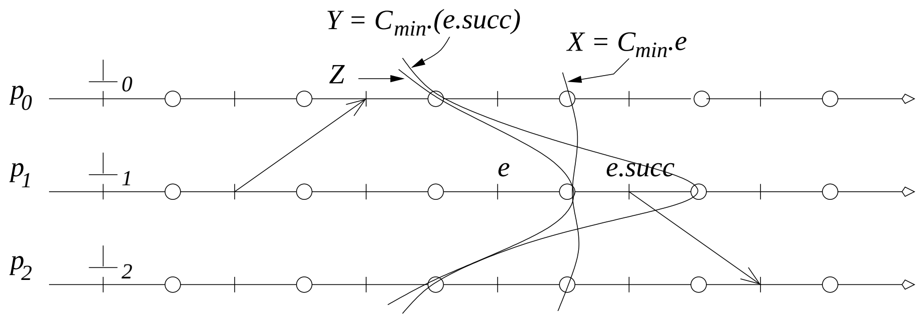

Figure 6: C_ϕ, for a region predicate ϕ, does not form an inf-semilattice.

phase starts when α_i is executed and continues until the execution on $\alpha_i.proc$ advances beyond α_i, that is, $\alpha_i.succ$ 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 $E.\bot$ and the final consistent cut E belong to at least one, possibly different, phase (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 $\prec$, that is, for each i and j, $i < j \Rightarrow \alpha_j \not\prec \alpha_i$,*
- **(boundary condition)** *$\alpha_0 \in E.\bot$ 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, that is, $legal(C, E_\prec, \alpha) \wedge \langle \exists\, \alpha_i :: \alpha_i \in C.frontier \rangle \Rightarrow \phi.C$.*

For example, in Figure 7, the sequence $e_0 e_1 f_2$ is not an admissible sequence as the initial consistent, given by $\{\bot_0, \bot_1\}$, does not belong to any phase (the boundary condition is violated). The sequence $\bot_0 f_1 e_0 f_2$ 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 $\bot_0 e_0 f_1 f_2$ 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 $\bot_0 e_0 f_0 e_2$ satisfies the boundary condition, and the agreement and continuity properties.

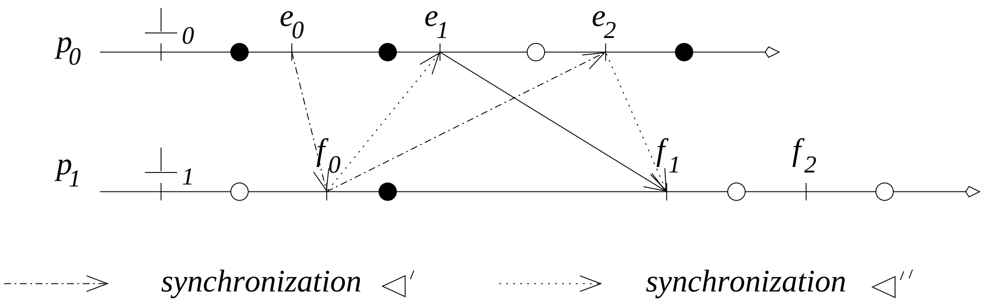

Figure 7: an illustration of the synchronization corresponding to an admissible sequence.

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. ■

Next, we show that the 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 7 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. Lemma 7 proves that $\lhd'$ does not create any deadlocks. Further, 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. Lemma 8 proves that if $\lhd''$ creates any cycle at all then it also creates a cycle involving exactly one arrow of type $\lhd''$. Lemma 7 and Lemma 8 are stated here for the sake of completeness but proved in Appendix A.

Lemma 7 *Let α be a sequence of events of a computation $E_\prec$. If the order of events in α, denoted by $<$, is consistent with $\prec$ then $(\prec \cup <)^+$ defines an irreflexive partial order on E.*

Lemma 8 *Let $\lhd$ be a relation from a chain of events α of a computation $E_\prec$ to E, that is, $\lhd \subseteq \alpha \times E$. If there is a cycle in $\prec \cup \lhd$ involving m edges of type $\lhd$, $m > 1$, then there is a cycle involving fewer than m edges of type $\lhd$.*

Finally, Theorem 9 establishes that the existence of an admissible sequence is a sufficient condition for a predicate to be controllable in a computation.

Theorem 9 *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, that is, there are no cycles, and (2) every consistent of the controlled computation satisfies ϕ.

We add the synchronization it in two steps. We first add $\lhd'$ to $\prec$. Lemma 7 ascertains that $\prec' = (\prec \cup \lhd')^+$ is an irreflexive partial order. We then add $\lhd''$ to $\prec'$. Lemma 8 ensures that $\prec' \cup \lhd''$ is acyclic provided it does not contain any cycle involving a single edge of type $\lhd''$. Assume the contrary. Observe that there is no cycle in $\prec' \cup \lhd''$ of the form $e \lhd'' e$. Therefore any cycle involving a single edge of type $\lhd''$, if it exists, is of the form $\alpha_{i+1} \lhd'' \alpha_i.succ \prec' \alpha_{i+1}$. Since α satisfies the continuity property, $\alpha_i.succ \not\prec \alpha_{i+1}$ implying there exists an edge of type $\lhd'$ in the path from $\alpha_i.succ$ to α_{i+1}. In other words, for some j and k, $\alpha_i.succ \preccurlyeq \alpha_j \lhd' \alpha_k \preccurlyeq' \alpha_{i+1}$. Note that α_j and α_{i+1} are ordered by $\lhd'$. As a result, $\alpha_j \lhd' \alpha_{i+1}$, otherwise there is a cycle in $\prec'$. Further, since $\alpha_i \prec \alpha_j$, using the agreement property, $\alpha_i \lhd' \alpha_j$. Thus $\alpha_i \lhd' \alpha_j \lhd' \alpha_{i+1}$ implying $i < j < i+1$, a contradiction. Consequently $\prec' \cup \lhd''$ is acyclic.

Let $\sqsubset$ denote the irreflexive transitive closure of $\prec' \cup \lhd''$, that is, $\sqsubset = (\prec' \cup \lhd'')^+$. Since $\prec' \cup \lhd''$ is acyclic, $\sqsubset$ is an irreflexive partial order. It remains to be shown that every consistent cut of $E_\sqsubset$ satisfies ϕ. Consider a consistent cut C of $E_\sqsubset$. Observe that C is a consistent cut of $E_\prec$ and is legal with respect to α. It suffices to show that, for some i, $\alpha_i \in C.frontier$ and the weak safety property guarantees that ϕ is true for C. To that effect, we first prove that if $\alpha_i \notin E.\top$ then $\alpha_{i+1} \sqsubseteq \alpha_i.succ$. There are two cases: $\alpha_i.proc \neq \alpha_{i+1}.proc$ or $\alpha_i.proc = \alpha_{i+1}.proc$. In the first case, $\alpha_{i+1} \lhd'' \alpha_i.succ$ which implies $\alpha_{i+1} \sqsubset \alpha_i.succ$. In the second case, using the agreement property, $\alpha_i \prec \alpha_{i+1}$ implying $\alpha_i.succ \preccurlyeq \alpha_{i+1}$. Further, since α satisfies the continuity property, $\alpha_i.succ \not\prec \alpha_{i+1}$ which implies $\alpha_i.succ = \alpha_{i+1}$. This proves that if $\alpha_i \notin E.\top$ then $\alpha_{i+1} \sqsubseteq \alpha_i.succ$.

Now, since α satisfies the boundary condition, $\alpha_0 \in E.\bot$ implying $\alpha_0 \in C$. Thus (1) $\alpha_0 \in C$, (2) $\alpha_{|\alpha|-1} \in E.\top$, and (3) for each i, if $\alpha_i \notin E.\top$ then $\alpha_{i+1} \sqsubseteq \alpha_i.succ$. We claim that α contains at least one event from $C.frontier$. Assume the contrary. Since $\alpha_0 \in E.\bot$ and $\alpha_0 \notin C.frontier$, $\alpha_0.succ \in C$. Consequently, $\alpha_1 \in C$. It can be proved by induction that $\alpha_{|\alpha|-1} \in C$. Now, since $\alpha_{|\alpha|-1} \in E.\top$, we have $\alpha_{|\alpha|-1} \in C.frontier$, a contradiction. Therefore, for some i, $\alpha_i \in C.frontier$ implying C satisfies ϕ. ∎

Theorem 10 *A predicate ϕ is controllable in a computation $E_\prec$ iff there exists an admissible sequence of events with respect to ϕ and $E_\prec$.*

Proof: Follows from Theorem 6 and Theorem 9. ∎

5 Disjunctive Predicates

In this section, we give an efficient algorithm to solve the predicate control problem for the class of disjunctive predicates based on the notion of admissible sequences. 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, that is, 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 $e \in E.\bot$. 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

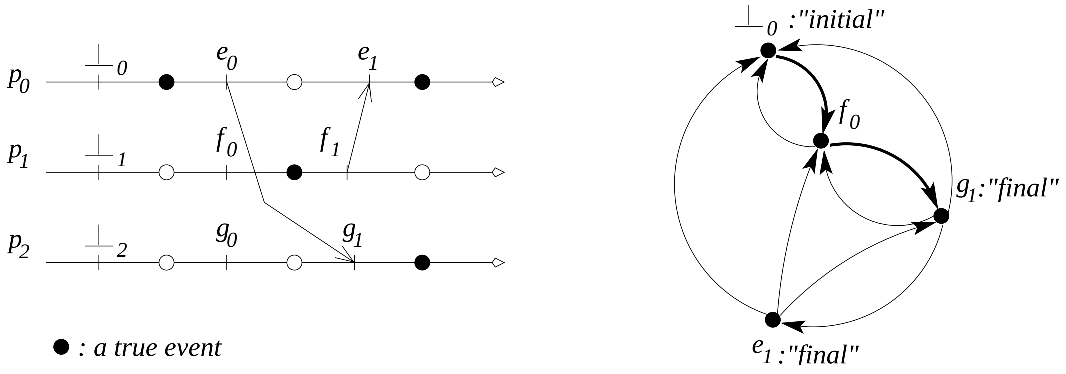

Figure 8: a computation and its corresponding TEG.

"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, by definition, every permissible path in the graph satisfies the boundary condition. Due to the semantics of disjunction, every path in the graph satisfies the weak safety property. Further, there is a one-to-one correspondence between paths in the graph and sequences of true events that satisfy the continuity property. We prove that the shortest permissible path in the graph also satisfies the agreement property.

Theorem 11 *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 with respect to ϕ and $E_{\prec}$.*

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. Our obligation is to prove that π satisfies the agreement property. Assume, by the way of contradiction, π does not respect $\prec$ (see Figure 9). Therefore there exist vertices π_i and π_j, $i < j$, such that $\pi_j \prec \pi_i$. Note that π_j cannot be an "initial" vertex, otherwise the path $\pi_j \pi_{j+1} \cdots \pi_{m-1}$ is a shorter permissible path. Further, π_i cannot be an "initial" vertex, otherwise $\pi_i \prec \pi_j$ implying $\prec$ has a cycle. Thus $i > 0$ which implies π_{i-1} exists. We claim that there is an edge from π_{i-1} to π_j in $\mathcal{G}$. If not,

$$
\begin{array}{ll}
 & \{ \text{ there is no edge from } \pi_{i-1} \text{ to } \pi_j \text{ in } \mathcal{G} \ \} \\
 & \pi_{i-1}.succ \prec \pi_j \\
 & \{ \ \pi_j \prec \pi_i \ \} \\
\Rightarrow & \pi_{i-1}.succ \prec \pi_i \\
 & \{ \text{ definition of an edge in } \mathcal{G} \ \} \\
\equiv & (\pi_{i-1}, \pi_i) \notin \mathcal{E}
\end{array}
$$

$\{$ since π is a path in $\mathcal{G}$, $(\pi_{i-1}, \pi_i) \in \mathcal{E}$ $\}$

$\Rightarrow$ a contradiction

Therefore $\pi_{i-1}.succ \not\prec \pi_j$ which implies there is an edge from π_{i-1} to π_j in $\mathcal{G}$. Observer that $\pi' = \pi_0\pi_1 \cdots \pi_{i-1}\pi_j \cdots \pi_{m-1}$ is a valid permissible path in $\mathcal{G}$. Moreover, π' is shorter than π, a contradiction. Thus π respects $\prec$. That is, π satisfies the agreement property. ∎

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

Theorem 12 *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 $E.\bot$. 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. Since $\pi_i.succ \in C_i$ but $\pi_{i+1}.succ \notin C_i$, $\pi_i.succ \sqsubset \pi_{i+1}.succ$ 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. ∎

Theorem 13 *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 $\mathcal{G}$.*

Proof: Follows from Theorem 10, Theorem 11 and Theorem 12. ∎

The algorithm presented here has $O(n^2m^2)$ 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 $h \leqslant_P f$.

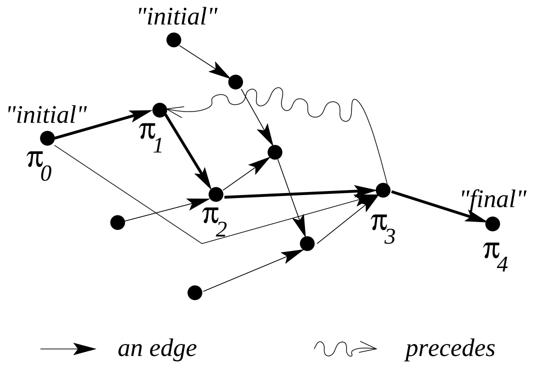

Figure 9: an illustration of a permissible path in TEG.

Thus Theorem 13 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 true event f on p such that $e.succ \not\preccurlyeq f$. This ensures that there are at most $O(n^2 m)$ edges in the graph. Further, by using *true-intervals*[‡] instead of true events to construct the graph, we can reduce the time complexity to $O(n^2 t)$, where t is the maximum number of true-intervals on any process.

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 4.2) suffices. This is because the admissible sequence constructed from a TEG satisfies a stronger property than the weak safety property, namely $\langle \exists \alpha_i :: \alpha_i \in C_i.\mathit{frontier} \rangle \;\Rightarrow\; \phi.C$.

We prove that the shortest permissible path in 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 message 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 $\prec$, that is, $|\lhd'' \setminus \prec|$. Secondly, among all paths that minimize $|\lhd'' \setminus \prec|$, it gives the path with the smallest number of edges, that is, the path that minimizes $|\lhd''|$. Figure 10 gives an example of

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

a WTEG[§]. The next theorem proves that the shortest permissible path in a WTEG constitutes an admissible sequence of events.

Theorem 14 *Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ be the WTEG 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 with respect to ϕ and $E_\prec$.*

Proof: Let $\pi = \pi_0\pi_1\cdots\pi_{m-1}$ be the shortest permissible path in $\mathcal{G}_w$. As argued before, π satisfies the boundary condition, and the continuity and weak safety properties. Our obligation is to prove that π satisfies the agreement property. Assume, by the way of contradiction, π does not respect $\prec$. Therefore there exists vertices π_i and π_j, $i < j$, such that $\pi_j \prec \pi_i$. Note that π_j cannot be an "initial" vertex, otherwise the path $\pi_j\pi_{j+1}\cdots\pi_{m-1}$ is a shorter permissible path. Further, π_i cannot be an "initial" vertex as $\pi_i \prec \pi_j$ implying $\prec$ has a cycle. Thus $i > 0$ which implies π_{i-1} exists. We claim that there is an edge from π_{i-1} to π_j in $\mathcal{G}_w$. The claim is similar to the one made in the proof of Theorem 11 which also contains its proof. Let π' denote the path $\pi_0\pi_1\cdots\pi_{i-1}\pi_j\cdots\pi_{m-1}$. There are two cases: either $w.(\pi_{i-1},\pi_j) = (0,1)$ or $w.(\pi_{i-1},\pi_j) = (1,1)$. If $w.(\pi_{i-1},\pi_j) = (0,1)$ then π' has a smaller weight than π. This contradicts the fact that π is the shortest permissible path in $\mathcal{G}_w$. Therefore assume $w.(\pi_{i-1},\pi_j) = (1,1)$. We show that the path from π_{i-1} to π_j contains at least one edge with weight $(1,1)$. In particular, we prove that the weight of edge from π_{i-1} to π_i is $(1,1)$. To that effect, we have,

$$
\begin{array}{ll}
 & \{\ w.(\pi_{i-1},\pi_j) = (1,1)\ \} \\
 & \pi_j \not\preceq \pi_{i-1}.succ \\
 & \{\ \pi_j \prec \pi_i\ \} \\
\Rightarrow & \pi_i \not\preceq \pi_{i-1}.succ \\
 & \{\ \text{definition of } w\ \} \\
\equiv & w.(\pi_{i-1},\pi_i) = (1,1)
\end{array}
$$

As a result, π' has a smaller weight than π, a contradiction. Thus π satisfies the agreement property implying the sequence of events corresponding to π constitutes to an admissible sequence of events with respect to ϕ and $E_\prec$. ■

For a tuple t, let $t.fst$ and $t.snd$ represent the first and second entries, respectively. Further, let $n.\mathcal{G}_w$ denote the first entry of the weight of the shortest permissible path, if it exists, in $\mathcal{G}_w$, that is, $|\lhd'' \setminus \prec|$, where $\lhd''$ is the synchronization (defined in D 4.2) corresponding to the path. In the next lemma, we prove that a single synchronization arrow reduces the value of $n.\mathcal{G}_w$ by at most one.

[§]some edges have been omitted from the graph for the purpose of clarity

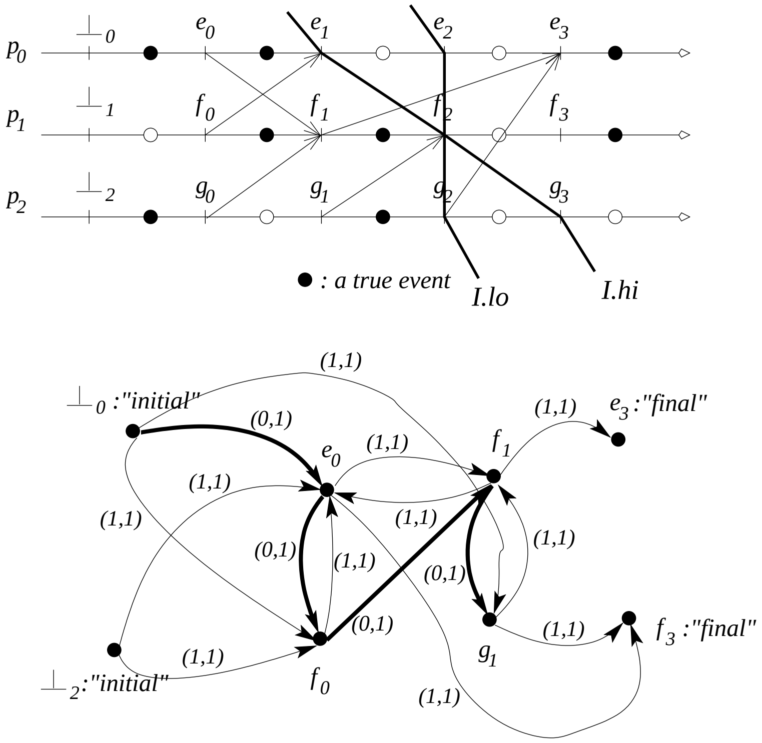

Figure 10: an illustration of a WTEG.

Lemma 15 (bounded reduction) *Let $E_{\prec}$ and $E_{\prec'}$ be computations such that $\prec \subseteq \prec'$ and $E_{\prec'}$ has exactly one additional message than $E_{\prec}$. Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ and $\mathcal{G}'_w = (\mathcal{V}'_w, \mathcal{E}'_w)$ be the WTEGs corresponding to a disjunctive predicate ϕ and computations $E_{\prec}$ and $E_{\prec'}$, respectively. If ϕ is controllable in $E_{\prec'}$ then $n.\mathcal{G}_w - n.\mathcal{G}'_w \leq 1$.*

Proof: Since ϕ is controllable in $E_{\prec'}$, using Theorem 13, there exists a permissible path in $\mathcal{G}'_w$. Let $\pi = \pi_0\pi_1 \cdots \pi_{m-1}$ be the shortest permissible path in $\mathcal{G}'_w$. For convenience, we use w and w' to denote the weight functions of $\mathcal{G}_w$ and $\mathcal{G}'_w$, respectively. Note that, for each e and f, if $e.succ \not\preccurlyeq' f$ then $e.succ \not\preccurlyeq f$. Thus every edge of $\mathcal{G}'_w$ is also an edge of $\mathcal{G}_w$ implying π is a path in $\mathcal{G}_w$. Further, observe that,

$$w.\pi.fst \geq n.\mathcal{G}_w, \tag{1}$$

$$w'.\pi.fst = n.\mathcal{G}'_w, \text{ and} \tag{2}$$

$$\langle \forall\, e, f : (e,f) \in \mathcal{E}'_w : w.(e,f) = (0,1) \;\Rightarrow\; w'.(e,f) = (0,1) \rangle. \tag{3}$$

We claim that $w.\pi.fst - w'.\pi.fst \leq 1$. Assume the contrary. Therefore, from (3), we can infer that there exist at least two distinct edges in π, say (π_i, π_{i+1}) and (π_j, π_{j+1}), such that their weight in $\mathcal{G}_w$ is $(1,1)$ but in $\mathcal{G}'_w$ is $(0,1)$. Equivalently, $\pi_{i+1} \not\preccurlyeq \pi_i.succ$ and $\pi_{j+1} \not\preccurlyeq \pi_j.succ$ but $\pi_{i+1} \preccurlyeq' \pi_i.succ$ and $\pi_{j+1} \preccurlyeq' \pi_j.succ$. Let the additional message in $E_{\prec'}$ be from event e

to event f. Observe that there must exist a path from π_{i+1} to $\pi_i.succ$ in $E_{\prec'}$ that involves the message from e to f. Likewise, there must exist a path from π_{j+1} to $\pi_j.succ$ that involves the message from e to f. Thus,

$$\pi_{i+1} \preccurlyeq e \text{ and } f \preccurlyeq \pi_i.succ, \text{ and} \tag{4}$$
$$\pi_{j+1} \preccurlyeq e \text{ and } f \preccurlyeq \pi_j.succ. \tag{5}$$

Without loss of generality assume $i < j$. There are two cases: either $(\pi_i, \pi_{j+1}) \notin \mathcal{E}'_w$ or $(\pi_i, \pi_{j+1}) \in \mathcal{E}'_w$.

Case 1 $[(\pi_i, \pi_{j+1}) \notin \mathcal{E}'_w]$**:** We have,

$$
\begin{aligned}
& \{\ (\pi_i, \pi_{j+1}) \notin \mathcal{E}'_w\ \} \\
& \pi_i.succ \prec' \pi_{j+1} \\
& \{\ \text{using (5) and } \prec \subseteq \prec'\ \} \\
\Rightarrow\ & \pi_i.succ \prec' e \\
& \{\ \text{using (4) and } \prec \subseteq \prec'\ \} \\
\Rightarrow\ & f \prec' e \\
& \{\ e \prec' f\ \} \\
\Rightarrow\ & \prec' \text{ has a cycle, a contradiction.}
\end{aligned}
$$

Case 2 $[(\pi_i, \pi_{j+1}) \in \mathcal{E}'_w]$**:** Let π' denote the path $\pi_0\pi_1 \cdots \pi_i\pi_{j+1} \cdots \pi_{m-1}$. Note that (4) and (5) imply that $\pi_{j+1} \preccurlyeq' \pi_i.succ$. Thus $w'.(\pi_i, \pi_{j+1}) = (0, 1)$ implying π' has a smaller weight than π, a contradiction.

In either case there is a contradiction. Thus,

$$w.\pi.fst - w'.\pi.fst \leq 1. \tag{6}$$

Finally,

$$
\begin{aligned}
& \{\ \text{using (1)}\ \} \\
& n.\mathcal{G}_w \leq w.\pi.fst \\
& \{\ \text{using (6)}\ \} \\
\Rightarrow\ & n.\mathcal{G}_w \leq w'.\pi.fst + 1 \\
& \{\ \text{using (2)}\ \}
\end{aligned}
$$

$$\Rightarrow \quad n.\mathcal{G}_w \leq n.\mathcal{G}'_w + 1$$

$$\{ \text{ algebra } \}$$

$$\equiv \quad n.\mathcal{G}_w - n.\mathcal{G}'_w \leq 1$$

This proves the lemma. ∎

We now prove that if all executions of $E_\prec$ are safe then $n.\mathcal{G}_w = 0$. Assume ϕ is controllable in $E_\prec$ and $n.\mathcal{G}_w > 0$. Let $\mathcal{V}_w^0$ denote the subset of vertices of $\mathcal{V}_w$ that are reachable from an "initial" vertex via edges with weight $(0,1)$ only. There is no "final" vertex in $\mathcal{V}_w^0$, otherwise there is a permissible path in $\mathcal{G}_w$ consisting only of edges with weight $(0,1)$ implying $n.\mathcal{G}_w = 0$. For each process p_i, we define a set, denoted by $\mathcal{I}_i$, that contains a contiguous sequence of events of p_i as follows. The first event of $\mathcal{I}_i$, represented by $\mathcal{I}_i.lo$, is defined as the successor of the last event of p_i in $\mathcal{V}_w^0$. Observe that $\mathcal{I}_i.lo$ is uniquely defined because events of p_i are totally ordered. In case there is no event of p_i in $\mathcal{V}_w^0$, $\mathcal{I}_i.lo$ is set to $\bot_i$. It can be easily proved that $\mathcal{I}_i.lo$ is not a true event. The last event of $\mathcal{I}_i$, denoted by $\mathcal{I}_i.hi$, is the event of p_i that occurs after (including) $\mathcal{I}_i.lo$ such that its successor is a true event. If there is no true event on p_i after $\mathcal{I}_i.lo$ then $\mathcal{I}_i.hi$ is set to the "final" event of p_i. It is easy to see that no event in $\mathcal{I}_i$ is a true event. For convenience,

$$\begin{aligned}
\mathcal{I} &\stackrel{\text{def}}{=} \bigcup_{0 \leq i < n} \mathcal{I}_i, \\
\mathcal{I}.lo &\stackrel{\text{def}}{=} \{ \mathcal{I}_i.lo \mid 0 \leq i < n \}, \\
\mathcal{I}.hi &\stackrel{\text{def}}{=} \{ \mathcal{I}_i.hi \mid 0 \leq i < n \}, \text{ and} \\
\mathcal{I}.hi.succ &\stackrel{\text{def}}{=} \{ e.succ \mid e \in \mathcal{I}.hi \text{ and } e \notin E.\top \}
\end{aligned}$$

Consider the WTEG in Figure 10. The edges with weight $(0,1)$ have been drawn in thick. Here, $\mathcal{V}_w^0 = \{\bot_0, e_0, f_0, f_1, \bot_2, g_1\}$. Further, $\mathcal{I}_0.lo = e_1$, $\mathcal{I}_0.hi = e_2$ and $\mathcal{I}_0 = \{e_1, e_2\}$. Likewise, $\mathcal{I}_2.lo = g_2$, $\mathcal{I}_2.hi = g_3$ and $\mathcal{I}_2 = \{g_2, g_3\}$. Finally, $\mathcal{I} = \{e_1, e_2, f_2, g_2, g_3\}$ and $\mathcal{I}.hi.succ = \{e_3, f_3\}$.

Observe that (1) $\mathcal{I}.hi.succ \neq \emptyset$, (2) all events in $\mathcal{I}.hi.succ$ are true events, and (3) no event in $\mathcal{I}.hi.succ$ is in $\mathcal{V}_w^0$. We prove that there exists a consistent cut of $E_\prec$ such that its frontier contains events only from $\mathcal{I}$, thereby implying that the cut does not satisfy ϕ. To that effect, we define a relation $\sqsubset$ as follows:

$$\textbf{(D 5.1)} \quad \sqsubset \stackrel{\text{def}}{=} \{ (e, f) \mid e \in \mathcal{I}.lo \text{ and } f \in \mathcal{I}.hi.succ \}$$

Intuitively, $\sqsubset$ adds an arrow from each event in $\mathcal{I}.lo$ to the successor of every event in $\mathcal{I}.hi$, if it exists, thereby making $\mathcal{I}$ inevitable. That is, every execution of the computation contains a consistent cut whose frontier lies completely within $\mathcal{I}$. We first prove that $\prec \cup \sqsubset$ is acyclic.

Lemma 16 *Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ be the WTEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$ such that ϕ is controllable in $E_\prec$ and $n.\mathcal{G}_w > 0$. Further, let $\sqsubset$ be the relation as defined (in D 5.1). Then $\prec \cup \sqsubset$ is acyclic.*

Proof: Note that $\sqsubset$ is a relation from $\mathcal{I}.lo$ to $\mathcal{I}.hi.succ$. Since $\mathcal{I}.lo \cap \mathcal{I}.hi.succ = \emptyset$, $\sqsubset$ is acyclic. Consider a path $e \sqsubset f \prec g \sqsubset h$ in $\prec \cup \sqsubset$ (events e, f, g and h need not be all distinct, that is, an event or a sequence of events may be repeated in the path). Using definition of $\sqsubset$, $f \in \mathcal{I}.hi.succ$ and $g \in \mathcal{I}.lo$. Moreover, by definition of $\mathcal{I}$, f and the predecessor of g, say u, are true events with u contained in $\mathcal{V}_w^0$ but f not contained in $\mathcal{V}_w^0$. There is an edge from u to f in $\mathcal{G}_w$ with weight $(0, 1)$ because $f \preccurlyeq u.succ\ (= g)$. Since there is a path from an "initial" vertex to u via edges with weight $(0, 1)$, f is reachable from an "initial" vertex via edges with weight $(0, 1)$ only. This implies that f is contained in $\mathcal{V}_w^0$, a contradiction. Thus there is no path in $\prec \cup \sqsubset$ of the form $e \sqsubset f \prec g \sqsubset h$ implying $\prec \cup \sqsubset$ is acyclic. ∎

We next prove that if $n.\mathcal{G}_w > 0$ then some consistent cut of $E_\prec$ does not satisfy ϕ.

Theorem 17 *Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ be the WTEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$ such that ϕ is controllable in $E_\prec$ and $n.\mathcal{G}_w > 0$. Then there exists a consistent cut of $E_\prec$ that does not satisfy ϕ.*

Proof: Let $\sqsubset$ be the relation as defined (in D 5.1). Using Lemma 16, $\prec \cup \sqsubset$ is acyclic. Let $<$ be any total order that extends $\prec \cup \sqsubset$. Consider a consistent cut, say C, of $E_<$ that contains $\mathcal{I}.lo$ such that the last event of C, denoted by $C.last$, is an event of $\mathcal{I}.lo$. Observe that C is also a consistent cut of $E_\prec$. We claim that the frontier of C contains events from $\mathcal{I}$ only.

We first show that $\mathcal{I}.hi.succ \cap C = \emptyset$. Assume the contrary. Therefore there exists an event in $\mathcal{I}.hi.succ$, say e, that is contained in C. Since $\mathcal{I}.lo \cap \mathcal{I}.hi.succ = \emptyset$, $e \neq C.last$. By definition of $\sqsubset$, for each $f \in \mathcal{I}.lo$, $f \sqsubset e$. This implies that $C.last < e$ because $C.last \in \mathcal{I}.lo$. Further, since e is contained in C, $e < C.last$. Thus $<$ has a cycle, a contradiction.

Now, let g be an event in the frontier of C and $g.proc = p_i$. As $\mathcal{I}.lo \subseteq C$, $\mathcal{I}_i.lo \leqslant_P g$. Further, $g \leqslant_P \mathcal{I}_i.hi$ because $\mathcal{I}_i.hi.succ$, if it exists, is not contained in C. Thus g is an event in $\mathcal{I}_i$. Since g was chosen arbitrarily, every event in the frontier of C is an event in $\mathcal{I}$. Since no event in $\mathcal{I}$ is a true event, ϕ is false for C. ∎

Corollary 18 *Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ be the WTEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$. If every consistent cut of $E_\prec$ satisfies ϕ then $n.\mathcal{G}_w = 0$.*

Finally, we prove that the shortest permissible path in $\mathcal{G}_w$ gives a synchronization that is minimum.

Theorem 19 *Let $\mathcal{G}_w = (\mathcal{V}_w, \mathcal{E}_w)$ be the WTEG corresponding to a disjunctive predicate ϕ and a computation $E_\prec$. The shortest permissible path in $\mathcal{G}_w$, if it exists, corresponds to the minimum synchronization needed to control ϕ in $E_\prec$.*

Proof: Assume there exists a permissible path in $\mathcal{G}_w$. Therefore ϕ is controllable in $E_{\prec}$. Let $\lhd_s$ denote the minimum synchronization needed to control ϕ in $E_{\prec}$. Further, let $\{\mathcal{G}_w^{(k)}\}$ represent the sequence of WTEGs generated by adding synchronization arrows from $\lhd_s$ one-by-one, where $\mathcal{G}_w^{(0)} = \mathcal{G}_w$. Using the bounded reduction lemma,

$$n.\mathcal{G}_w^{(i)} - n.\mathcal{G}_w^{(i+1)} \leq 1, \quad 0 \leq i < |\lhd_s|$$

Adding the above inequality for all values of i we get,

$$
\begin{aligned}
& n.\mathcal{G}_w^{(0)} - n.\mathcal{G}_w^{|\lhd_s|} \leq |\lhd_s| \\
& \{ \text{ Corollary 18 } \} \\
\equiv\ & n.\mathcal{G}_w - 0 \leq |\lhd_s| \\
& \{ \text{ algebra } \} \\
\equiv\ & n.\mathcal{G}_w \leq |\lhd_s| \\
& \{ \lhd_s \text{ is the minimum synchronization } \} \\
\equiv\ & n.\mathcal{G}_w = |\lhd_s|
\end{aligned}
$$

Thus the shortest permissible path in $\mathcal{G}_w$ gives the minimum synchronization. ■

6 Conclusions

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 predicate control problem. 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.

References

[CG98] Craig Chase and Vijay K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CL85] 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.

[CL95] R. Cypher and E. Leu. Efficient Race Detection for Message-Passing Programs with Nonblocking Sends and Receives. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 534–541, 1995.

[CM91] 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.

[Fid91] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[HK93] 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.

[HPR93] M. Hurfin, N. Plouzeau, and Michel Raynal. Detecting Atomic Sequences of Predicates in Distributed Computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 32–42, 1993.

[KC97] 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.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[MC88] B. P. Miller and J. Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the IEEE 8th International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[MSWW81] 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.

[NM95] R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[SL98] Scott D. Stoller and Yanhong A. Liu. Efficient Symbolic Detection of Global Properties in Distributed Systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, 1998.

[Tai97] K. Tai. Race Analysis of Traces of Asynchronous Message-Passing Programs. In *Proceedings of the IEEE 17th International Conference on Distributed Computing Systems*, pages 261–268, 1997.

[TG98] Ashis Tarafdar and Vijay K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the IEEE 9th Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, 1998.

[TG99] 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.

[WHF+97] 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.

Appendix A

The Class of p-Region Predicates is Closed under Conjunction

Theorem 20 *If ϕ and ψ are p-region predicates then $\phi \wedge \psi$ is a p-region predicate.*

Proof: We need to show that the $\phi \wedge \psi$ satisfies the weak lattice and weak inclusion properties. Let e be an event on process p, and C, C' and C'' be consistent cuts of the computation that contain e in their frontiers. Assume $(\phi \wedge \psi).C$ and $(\phi \wedge \psi).C'$ hold. Therefore $\phi.C$, $\psi.C$, $\phi.C'$ and $\psi.C'$ hold. Using the weak lattice property for ϕ, $\phi.(C \cap C')$ and $\phi.(C \cup C')$ hold. Similarly, using the weak lattice property for ψ, $\psi.(C \cap C')$ and $\psi.(C \cup C')$ are true. Thus $(\phi \wedge \psi).(C \cap C')$ and $(\phi \wedge \psi).(C \cup C')$ hold which implies $\phi \wedge \psi$ satisfies the weak lattice property.

Now, assume $(\phi \wedge \psi).C'$ and $(\phi \wedge \psi).C''$ hold, and $C' \subseteq C \subseteq C''$. Therefore $\phi.C'$, $\psi.C'$, $\phi.C''$ and $\psi.C'$ and $\psi.C''$ hold. Using the weak inclusion property for ϕ, $\phi.C$ is true. Similarly, using the weak inclusion property for ψ, $\psi.C$ holds. Thus $(\phi \wedge \psi).C$ holds implying $\phi \wedge \psi$ satisfies the weak inclusion property. ∎

Proofs for Lemma 7 and Lemma 8

Lemma 7 *Let α be a sequence of events of a computation $E_{\prec}$ and $<$ denote the order of events in α. If $\prec$ and $<$ do not differ on the relative order of any two events then $(\prec \cup <)^+$ defines an irreflexive partial order on E.*

Proof: We show that $\prec \cup <$ is acyclic which ascertains that $(\prec \cup <)^+$ is an irreflexive partial order. Observe that both $\prec$ and $<$ are transitive relation. Thus any cycle in $\prec \cup <$, if it exists, consists of alternating edges of type $\prec$ and $<$. We first prove that if there is a cycle with m, $m > 1$, edges of type $<$ then there is a cycle with fewer than m edges of type $<$. We then show that there is no cycle that involves a single edge of type $<$.

Let $e_0 < e_1 \prec e_2 < e_3 \prec \cdots < e_{2m-1} \prec e_0$ be a cycle in $\prec \cup <$. Observe that e_0 and e_2 are the events in α and are, therefore, ordered by $<$. Since $<$ is a total order, either $e_0 \leqslant e_2$ or $e_2 \leqslant e_0$. In the first case, we have $e_0 \leqslant e_2 < e_3 \prec \cdots < e_{2m-1} \prec e_0$. Since $<$ is transitive, we obtain $e_0 < e_3 \prec \cdots < e_{2m-1} \prec e_0$. In the second case, we have $e_0 < e_1 \prec e_2 \leqslant e_0$. Again, since $<$ is transitive, we obtain $e_1 \prec e_2 < e_1$. In either case, there is a cycle with fewer than m edges of type $<$. This proves our first proposition.

Let $e_0 < e_1 \prec e_0$ be a cycle involving a single edge of type $<$. Since $e_0 < e_1$ and $e_1 \prec e_0$, $\prec$ and $<$ do not agree on the relative order e_0 and e_1, a contradiction. This proves our second proposition. Thus $\prec \cup <$ is acyclic. ∎

Lemma 8 *Let $\lhd$ be a relation from a chain of events α of a computation $E_{\prec}$ to E, i.e., $\lhd \subseteq \alpha \times E$. If there is a cycle in $\prec \cup \lhd$ involving m edges of type $\lhd$, $m > 1$, then there is a cycle involving fewer than m edges of type $\lhd$.*

Proof: Let $e_0 \lhd e_1 \preccurlyeq \cdots \preccurlyeq e_i \lhd e_{i+1} \preccurlyeq \cdots \preccurlyeq e_0$ be a cycle in $\prec \cup \lhd$. Observe that if $e_0 = e_i$ then we have a shorter cycle namely $e_0 \lhd e_{i+1} \preccurlyeq \cdots \preccurlyeq e_0$. Therefore assume $e_0 \neq e_i$. Since $\lhd$ is a relation from α to E, e_0 and e_i are events in α. Further, since all events in α are ordered by $\prec$ (α is a chain of events), either $e_0 \prec e_i$ or $e_i \prec e_0$. In either case, we have a cycle with fewer than m edges of type $\lhd$. ■

Appendix B

Controllability and Real World

To control a predicate in a computation, we add synchronization in the form of an arrow from one process execution to another. This synchronization can be realized in the real world by using control messages. In the real world, an event is either a *send*, a *receive* or an *internal* event, referred to as a *simple event*. As a result of adding synchronization, an event in a controlled computation may no longer be a simple send or receive event. After ignoring edges due to transitivity and between events on the same process, an event may have multiple incoming and/or outgoing edges; an incoming edge corresponds to the receive of a message and an outgoing edge corresponds to the send of a message. We call such an event as a *complex event*. Thus the controlled computation given by a partial order might not correspond to any computation in the real world. To overcome this problem, we give a transformation that converts any computation given by a partial order on a set of events, referred to as an *abstract computation*, to a computation in the real world, referred to as a *concrete computation*. The concrete computation consists of only simple events (except initial events) in contrast to the abstract computation which, in general, contains complex events. Further, our transformation ensures that if a predicate holds throughout the abstract computation then it also holds throughout the corresponding concrete computation.

Figure 11 shows the transformation, denoted by γ, at work. A complex event in the abstract computation is mapped to a sequence of simple events; there is an event for every receive or send of a message. Moreover, the receive events occur before the send events in the sequence. For example, in Figure 11, a complex event g is mapped to the sequence of simple events $\gamma(e) = g'g''g'''$. The events g' and g'' are receive events, and the event g''' is a send event. For simple events, the transformation reduces to the identity mapping.

The value of a predicate for a cut is evaluated using the values of the variables immediately after the execution of the events in the cut's frontier. Our transformation, in general, generates additional events. To be able to evaluate the predicate for an arbitrary cut in the concrete computation, we need to assign values to the relevant variables after execution of every event including the additional events generated by our mapping.

Let the value of an event e, denoted by $e.v$, be the value of the relevant variables immediately after execution of e. We assume the value of an event is unique for a process. Consider an event p in the transformation $\gamma(e)$ of an event e of an abstract computation $E_{\prec}$, $e \notin E.\bot$. If p is a send event or the last receive event in $\gamma(e)$ then $p.v = e.v$, otherwise $p.v = e.pred.v$. Let $\sigma(e)$ denote the set of events in the concrete computation that have the same value as e. It can be easily verified that $\sigma(e)$ constitutes a contiguous set of events on $e.proc$. Let $\sigma(e).lo$ and $\sigma(e).hi$ denote the first and the last event, respectively, in σ. Observe that, for each event $e \in E \setminus E.\bot$, $\sigma(e.pred).hi.succ = \sigma(e).lo$. Further, for a message from an event e to an event f in the abstract computation, let $\gamma.\langle e, f\rangle.snd$ and $\gamma.\langle e, f\rangle.rcv$ denote the send and receive events, respectively,

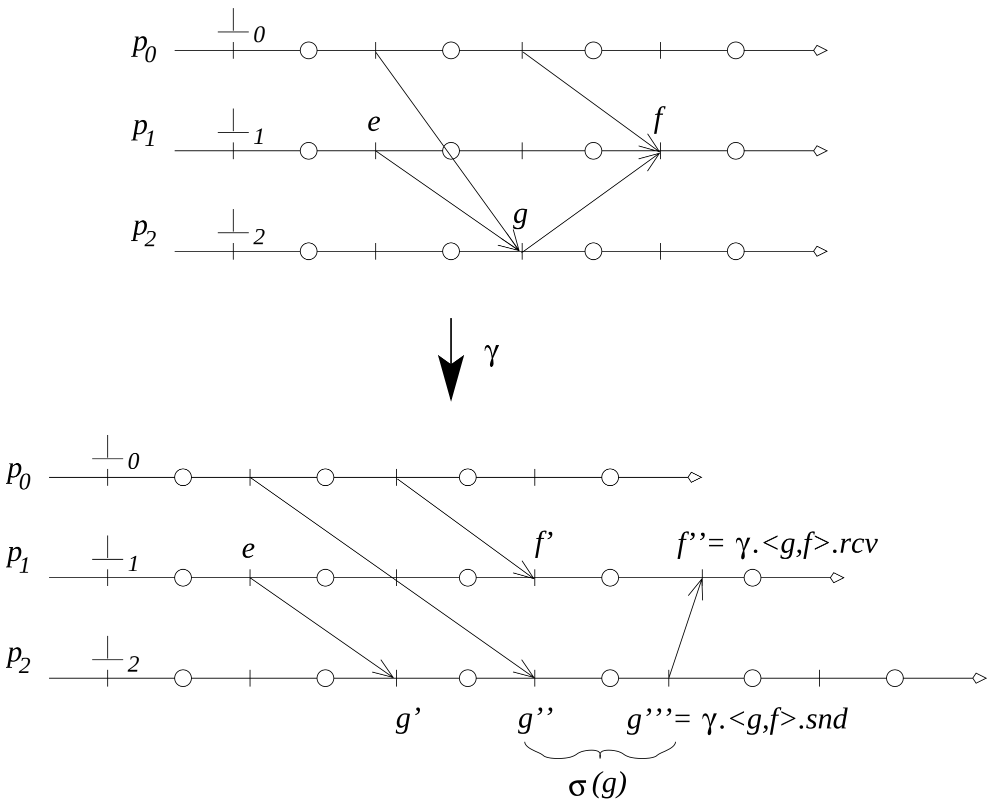

Figure 11: the transformation.

of the corresponding message in the concrete computation.

Lemma 21 *If e.succ precedes (or is equal to) f in an abstract computation then $\sigma(e).hi.succ$ precedes (or is equal to) $\sigma(f).lo$ in the concrete computation. Formally,*

$$e.succ \preccurlyeq f \;\Rightarrow\; \sigma(e).hi.succ \preccurlyeq \sigma(f).lo$$

Proof: The proof is by induction on length of the smallest message chain, n, from $e.succ$ to f.

Base Case ($n = 0$): In this case, the events e and f belong to the same process, and e occurs strictly before f, i.e., $e <_P f$. Therefore $\sigma(e).hi$ occurs before $\sigma(f).lo$ which implies $\sigma(e).hi.succ \preccurlyeq \sigma(f).lo$.

Induction Step ($n > 0$): Assume the implication holds if the smallest message chain from $e.succ$ to f is of length $n-1$. We prove that the implication holds if length of the smallest message chain from $e.succ$ to f is n. Let the last message in the smallest chain from $e.succ$ to f be from event u to event v. Thus $e.succ \preccurlyeq u$ and $v \leqslant_P f$. Using induction, $\sigma(e).hi.succ \preccurlyeq \sigma(u).lo$. Further, $\sigma(v).lo \preccurlyeq \sigma(f).lo$. Using properties of the transformation γ, $\gamma.\langle u, v\rangle.snd \in \sigma(u)$

and $\gamma.\langle u,v\rangle.rcv \preccurlyeq \sigma(v).lo$. Combining the results, $\sigma(e).hi.succ \preccurlyeq \sigma(u).lo \preccurlyeq \gamma.\langle u,v\rangle.snd$ and $\gamma.\langle u,v\rangle.rcv \preccurlyeq \sigma(v).lo \preccurlyeq \sigma(f).lo$. Consequently $\sigma(e).hi.succ \preccurlyeq \sigma(f).lo$.

Therefore, by induction, the theorem holds. ■

Corollary 22 *If e.succ precedes (or is equal to) f in an abstract computation then, for each $p \in \sigma(e)$ and $q \in \sigma(f)$, p.succ precedes (or is equal to) q in the concrete computation. Formally,*

$$e.succ \preccurlyeq f \;\Rightarrow\; \langle \forall\, p, q : p \in \sigma(e), q \in \sigma(f) : p.succ \preccurlyeq q \rangle$$

We now prove that if all executions of an abstract computation are safe then all executions of the corresponding concrete computation are also safe. We first introduce some notation. For an event p in the concrete computation, let $e = \sigma^{-1}(p)$ denote the event in the abstract computation such that $p \in \sigma(e)$ (equivalently, $p.v = e.v$).

Theorem 23 *If an abstract computation has no unsafe executions then all executions of the corresponding concrete computation are safe.*

Proof: Consider a consistent cut C_{cc} of the concrete computation. Let C_{ab} denote the cut of the abstract computation whose frontier is $\{\, \sigma^{-1}(p) \mid p \in C_{cc}.frontier\}$. Observe that σ^{-1} defines a one-to-one correspondence between the events of $C_{cc}.frontier$ and $C_{ab}.frontier$. We claim that C_{ab} is a consistent cut. Assume the contrary. Thus there exist events e and f in $C_{ab}.frontier$ such that $e.succ \preccurlyeq f$. Let p and q be the events of $C_{cc}.frontier$ such that $\sigma^{-1}(p) = e$ and $\sigma^{-1}(q) = f$. Using Corollary 22, $p.succ \preccurlyeq q$. Thus C_{cc} is not a consistent cut, a contradiction. Further, the value of each event p in $C_{cc}.frontier$ is equal to the value of the corresponding event $\sigma^{-1}(p)$ in $C_{ab}.frontier$ (definition of σ^{-1}) implying $\phi.C_{cc} \equiv \phi.C_{ab}$, for any ϕ. Therefore if $\phi.C_{ab}$ holds then ϕ is true for C_{cc}. Since every consistent cut of the abstract computation satisfies the given predicate, the predicate holds for every consistent cut of the concrete computation. ■

Intuitively, the transformation generates a computation in which a process first sends all messages one-by-one. It then waits for all messages that are supposed to be received. Finally, it processes the received messages and changes its state.