Distrib. Comput. (2004) 17: 107–130
Digital Object Identifier (DOI) 10.1007/s00446-003-0104-x

DISTRIBUTED COMPUTING

# Finding missing synchronization in a distributed computation using controlled re-execution*

**Neeraj Mittal**[1], **Vijay K. Garg**[2,**]

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA (e-mail: neerajm@utdallas.edu)
[2] Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA
(e-mail: garg@ece.utexas.edu)

Received: June 19, 2002 / Accepted: November 7, 2003
Published online: March 1, 2004 – © Springer-Verlag 2004

**Abstract.** Correct distributed programs are hard to write. Not surprisingly, distributed systems are especially vulnerable to software faults. Testing and debugging is an important way to improve the reliability of distributed systems. 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 of *predicate control*, which takes a computation and a safety property specified on the computation as inputs, and produces a controlled computation, with added synchronization, that maintains the given safety property as output. We devise efficient control algorithms for two classes of useful predicates, namely region predicates and disjunctive predicates. For the former, we prove that the control algorithm is *optimal* in the sense that it guarantees maximum concurrency possible in the controlled computation. For the latter, we prove that our control algorithm generates the *least* number of synchronization dependencies and therefore has *optimal* message-complexity. Furthermore, we provide a necessary and sufficient condition under which it is possible to efficiently compute a *minimal* controlling synchronization for a general predicate. We also give an algorithm to compute such a synchronization under the condition provided.

**Keywords:** Distributed system – Debugging – Software-fault tolerance – Controlled re-execution – Predicate control

## 1 Introduction

Inherent non-determinism in distributed programs and presence of multiple threads of control makes it difficult to write correct distributed software. Not surprisingly, distributed systems are especially vulnerable to software faults. Dealing with software faults requires efforts at multiple levels [22]. Early in the software cycle, design methodologies, technologies and techniques that are aimed at preventing the introduction of faults into the design can be used (*fault prevention*). Later, the implementation can be verified using testing, and the faults thereby exposed can be removed using debugging (*fault removal*). In spite of extensive testing and debugging, software faults may persist even in production quality software. *Fault tolerance* can be used as an extra layer of protection to provide acceptable level of performance and safety at runtime after a fault becomes active.

Testing and debugging has been widely used for developing traditional sequential programs. *Testing* involves executing the program for a specific input sequence and then validating the output obtained with respect to the given safety and liveness properties. On discovering a fault in the computation during testing phase, the next step is to analyze the computation to locate the source of the fault using *debugging*. While the skill and intuition of the programmer play an important role in debugging, tools that provide an effective environment for debugging are indispensable. For example, suppose testing detects a violation of safety property in an execution of a distributed program. Then a programmer can gain considerable insight into the bug that caused the violation by learning whether the violation occurs irrespective of the order in which events are executed. In that case, the bug cannot be fixed by adding or removing synchronization alone. On the other hand, if it is possible to eliminate all violations (of safety property) by adding synchronization to the computation, without creating a deadlock, then *too little* synchronization is likely to be the problem. Furthermore, the knowledge of the exact synchronization needed to maintain a safety property can facilitate the localization of the bug in the program. The problem of finding a synchronization required to maintain a safety property in a computation is referred to as the *predicate control* problem [20].

Informally, given a distributed computation and a global predicate, if it is possible to maintain the predicate, without violating liveness, by adding synchronization (one or more synchronization dependencies) to the computation, then the global predicate is controllable in the distributed computation. A synchronization dependency involves adding an arrow from one process execution to another which ensures that the

---

* A preliminary version of the results in this paper first appeared in [15].

** Supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

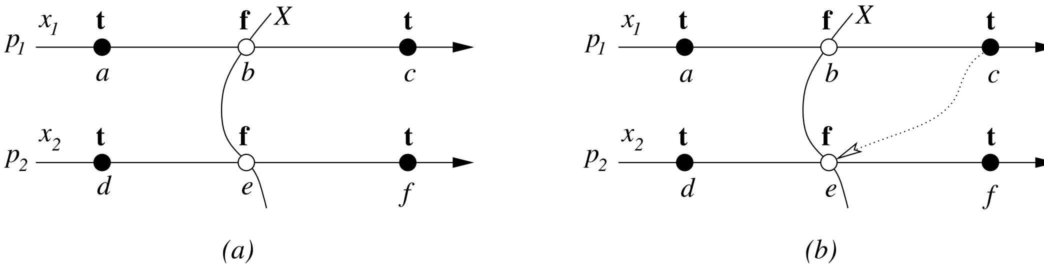


**Fig. 1. a** A computation, and **b** a controlled computation

execution after the head of the arrow can proceed only after the execution before the tail has completed; it can be realized using a control message. We believe that a distributed debugger equipped with predicate control mechanism can prove to be a valuable tool for a programmer.

*Example 1* Figure 1 depicts a computation involving two processes, namely $p_1$ and $p_2$. Processes $p_1$ and $p_2$ host boolean variables $x_1$ and $x_2$ respectively. In the figure circles denote events; a circle is coloured black if the relevant variable evaluates to true for the corresponding event (*e.g.*, $a$, $c$, $d$). Suppose the safety property is $x_1 \vee x_2$. Clearly, the consistent cut $X$ does not satisfy $x_1 \vee x_2$. However, on adding a synchronization dependency from event $c$ to event $e$, it can be ensured that the predicate $x_1 \vee x_2$ is never falsified. □

Predicate control has applications in the area of software-fault tolerance [21] as well. 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 [6,23]. 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. A study by Tarafdar [19] indicates that controlled re-execution is an effective and desirable method for tolerating race faults.

The research in distributed debugging has focused on mainly two problems: detecting bugs in a distributed computation (*e.g.*, [1–3,7,17,18]) and replaying the traced computation [9,11,14]. 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 first defined by Tarafdar and Garg in [20]. They prove that the problem is NP-complete in general. However, they solve the problem efficiently for classes of disjunctive predicates and mutual exclusion predicates [20,21]. Besides their work, there is another study [12] that focuses on controlling global predicates within the class of conditional elementary restrictions. Unlike our model of a distributed system, the model in [12] uses an off-line specification of pair-wise mutually exclusive states and does not use causality. Our contributions in this paper are as follows:

- We identify a class of useful global predicates, called *region predicates*, for which efficient control algorithms can be provided. Roughly speaking, a region predicate divides the state-space of a computation into regions satisfying two properties. Firstly, the set of consistent cuts that "lie" in a region forms a lattice. Secondly, each region is convex. Some examples of region predicates include "conjunction of local predicates" and "all processes are approximately synchronized".
- We present an efficient algorithm to compute the synchronization required to control a region predicate in a computation. We further demonstrate that the synchronization generated by our algorithm 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. The time- and message-complexities of the algorithm are $O(n|E|^2)$ and $O(n|E|)$, respectively, where $n$ is the number of processes and $E$ is the set of events in the computation.
- We introduce the notion of admissible sequence of events and establish that existence of such a sequence is a necessary and sufficient condition for controllability of a predicate in a computation. An admissible sequence gives a total order on some subset of events in the computation such that the sequence satisfies four simple properties, which are defined later in the paper. Intuitively, executing the events in the order specified by the sequence ensures that the given predicate is never falsified in the computation.
- Using the notion of admissible sequence, we give an efficient control algorithm for a disjunctive predicate. The time- and message-complexities of the algorithm are $O(n|T| + |E|)$ and $O(|T|)$, respectively, where $n$ is the number of processes, $E$ is the set of events, and $T$ is the set of true-intervals in the computation. The complexities are identical to that of Tarafdar and Garg's algorithm [20]. We further modify the algorithm to generate a controlling synchronization with the *least* number of synchronization dependencies, that is, with the *optimal* message-complexity. We believe that our approach is more general and can be extended to find a control strategy for other classes of predicates as well.
- Finally, we provide a necessary and sufficient condition under which it is possible to efficiently compute a *minimal* controlling synchronization for a general predicate. We also give an efficient algorithm to compute such a synchronization under the condition provided.

The paper is organized as follows. Section 2 presents our system model and the notation used in this paper. We define the problem formally in Sect. 3. In Sect. 4, we introduce the class of region predicates, provide an efficient algorithm for

their control, and prove the optimality of the controlling synchronization produced. In Sect. 5, we define the notion of admissible sequence, using which we give an efficient algorithm for controlling a disjunctive predicate. We further modify the algorithm to generate a *minimum* controlling synchronization. For a general predicate satisfying certain condition, we provide an efficient algorithm to compute a *minimal* controlling synchronization in Sect. 6. Finally, Sect. 7 concludes the paper and presents directions for future research.

## 2 Model and notation

In this section we formally describe the model and notation used in this paper. Our model is based on the Lamport's *happened-before* model [10].

### 2.1 Distributed computations

We assume an asynchronous distributed system with the set of processes $P = \{p_1, p_2, \ldots, p_n\}$. 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 channels are reliable, that is, messages are not lost, altered or spuriously introduced into a channel. We do not assume FIFO channels.

A *local computation* of a process is given by a sequence of events that transforms the *initial state* of the process into a *final state*. At each step, the *local state* is captured by the initial state together with the (sub)sequence of events that have been executed up to that step. Each event is either an *internal event* or an *external event*. An external event could be a *send event* or a *receive event* or both. An event causes the local state of a process to be updated. Additionally, a send event causes a message or a set of messages to be sent and a receive event causes a message or a set of messages to be received. We assume the presence of fictitious *initial events* on each process $p_i$, denoted by $\bot_i$. The initial event occurs before any other event on the process and initializes the state of that process. We denote the last event on process $p_i$, called the *final event*, by $\top_i$. Let $\bot$ and $\top$ denote the set of all initial events and final events, respectively.

Let $proc(e)$ denote the process on which event $e$ occurs. The predecessor and successor events of $e$ on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist. Observe that an initial event does not have a predecessor and a final event does not have a successor.

We model a *distributed computation* (or simply a *computation*) by an irreflexive partial order on a set of events. We use $\langle E, \rightarrow \rangle$ to denote a distributed computation with the set of events $E$ and the partial order $\rightarrow$. The partial order $\rightarrow$ is given by the Lamport's *happened-before relation* (or *causality relation*) [10] which is defined as the smallest transitive relation satisfying the following properties:

1. if events $e$ and $f$ occur on the same process, and $e$ occurred before $f$ in real time then $e$ happened-before $f$, and
2. if events $e$ and $f$ correspond to the send and receive, respectively, of a message then $e$ happened-before $f$.

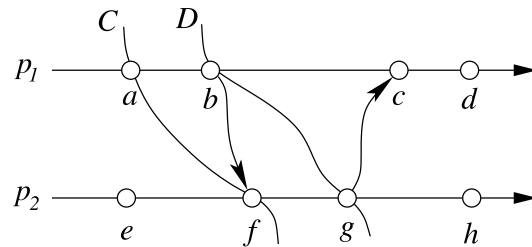


**Fig. 2.** An example of a computation

Given a computation $\langle E, \rightarrow \rangle$, we denote the order in which events are executed on processes by $\stackrel{P}{\rightarrow}$ which is referred to as *process order*. Note that the projection of $\stackrel{P}{\rightarrow}$ onto the events of any single process is a total order. The reflexive closure of an irreflexive partial order $\leadsto$ is represented by $\underline{\leadsto}$ and its transitive closure is denoted by $\leadsto^+$. A *run* or *interleaving* of a computation $\langle E, \rightarrow \rangle$ is some total order on events $E$ that is consistent with the partial order $\rightarrow$. The following example illustrates the various concepts.

*Example 2* Figure 2 depicts a distributed computation involving two processes, namely $p_1$ and $p_2$. The local computation of each process advances from left to right as shown in the figure. The circles represent events and the arrows denote messages. The local computation of $p_1$ is given by the sequence $a\ b\ c\ d$. The event $b$ is a send event, the event $f$ is a receive event and the event $d$ is an internal event. Here, $\bot_1 = a$ and $\bot_2 = e$ whereas $\top_1 = c$ and $\top_2 = h$. Also, $proc(b) = p_1, pred(b) = a$ and $succ(e) = c$. The set of events $E = \{a, b, c, d, e, f, g, h\}$ and the happened-before order $\rightarrow = \{(a, b), (b, c), (c, d), (e, f), (f, g), (g, h), (b, f), (g, c)\}^+$. The process order $\stackrel{P}{\rightarrow}$ is given by $\{(a, b),\ (b, c),\ (c, d), (e, f), (f, g),\ (g, h)\}^+$. Finally, $a\ e\ bfg\ h\ c\ d$ is a run of the computation. □

### 2.2 Cuts, consistent cuts and frontiers

The state of a distributed system, called the *global state*, is given by the collective state of processes. The equivalent notion based on events is called *cut* and is defined as a subset of events containing all initial events such that it contains an event only if its predecessor, if it exists, also belongs to the subset. Formally,

$$C \text{ is a cut} \\ \triangleq \\ (\bot \subseteq C) \ \wedge \langle \forall\, e : e \in C : e \notin \bot \ \Rightarrow\ pred(e) \in C \rangle$$

The *frontier* of a cut $C$ is defined as the set of those events in $C$ whose successors are not in $C$. Formally,

$$frontier(C) \ \triangleq\ \{\, e \in C \mid e \notin \top \ \Rightarrow\ succ(e) \notin C \,\}$$

We say that a cut *passes through* an event if the event is included in its frontier. Not every cut can occur during system execution. A cut is said to be *consistent* if it contains an event only if it also contains all events that happened-before it. Formally,

$$C \text{ is a consistent cut} \\ \triangleq \\ (C \text{ is a cut}) \ \wedge \langle \forall\, e, f : e \rightarrow f : f \in C \Rightarrow e \in C \rangle$$

In particular, only those cuts which are consistent can possibly occur during an execution. Lastly, two events are *consistent* if there exists a consistent cut that passes through both the events, otherwise they are *inconsistent*. It can be verified that events $e$ and $f$ are inconsistent if and only if either $succ(e) \underline{\rightarrow} f$ or $succ(f) \underline{\rightarrow} e$. The next example illustrates the various concepts.

*Example 3* Consider the computation in Fig. 2. The cut $C = \{a, e, f\}$ and $D = \{a, b, e, f, g\}$. The cut $D$ is consistent whereas $C$ is not. Here, $frontier(C) = \{a, f\}$ and $frontier(D) = \{b, g\}$. The events $b$ and $f$ are consistent whereas events $a$ and $f$ are not. □

### 2.3 Global predicates

A *global predicate* (or simply a *predicate*) is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the state resulting after executing all events in the cut. If a predicate $b$ evaluates to true for a consistent cut $C$, we say that "$C$ satisfies $b$" and denote it by $C \models b$. A global predicate is *local* if it depends on variables of a single process. Note that it is possible to evaluate a local predicate with respect to an event on the appropriate process. In case the predicate evaluates to true, the event is called a *true event*; otherwise, it is called a *false event*. We use $e \models b$ to denote the fact that the event $e$ satisfies the local predicate $b$.

A run is called *safe* with respect to a predicate if every consistent cut of the run satisfies the predicate; otherwise, the run is *unsafe*.

## 3 Problem statement

We say that two relations $R$ and $S$ *interfere* if $R \cup S$ contains a cycle; otherwise they do not interfere. Now, intuitively, a predicate is *controllable* in a computation if it is possible to make the computation "stricter" such that every consistent cut of the resulting computation satisfies the predicate. More precisely, a predicate $b$ is controllable in a computation $\langle E, \rightarrow \rangle$ if there exists a set of synchronization dependencies $\stackrel{S}{\rightarrow}$ such that (1) $\stackrel{S}{\rightarrow}$ does not interfere with $\rightarrow$, that is, $(\rightarrow \cup \stackrel{S}{\rightarrow})$ is acyclic, and (2) every consistent cut of $\langle E, \leadsto \rangle$, where $\leadsto = (\rightarrow \cup \stackrel{S}{\rightarrow})^+$, satisfies $b$ [20]. We call the synchronization $\stackrel{S}{\rightarrow}$ as a *controlling synchronization* and the computation $\langle E, \leadsto \rangle$ as the *controlled computation*. It can be verified that a predicate is controllable in a computation if and only if there exists a safe run of the computation with respect to the predicate [20]. Note that a synchronization dependency from an event $e$ to an event $f$ means that $f$ cannot be executed until $e$ has been executed and can be implemented using a control message.

We say that a predicate is *invariant* in a computation if the predicate evaluates to true for every consistent cut of the computation. Given a predicate $b$ and a computation $\langle E, \rightarrow \rangle$, we use $\langle E, \rightarrow \rangle \models controllable : b$ to denote the fact that $b$ is controllable in $\langle E, \rightarrow \rangle$. The expression $\langle E, \rightarrow \rangle \models invariant : b$ can be similarly interpreted. Note that the problem of evaluating $controllable : b$ is only concerned with answering whether there exists a controlling synchronization for $b$ and not with actually finding one, in case it exists.

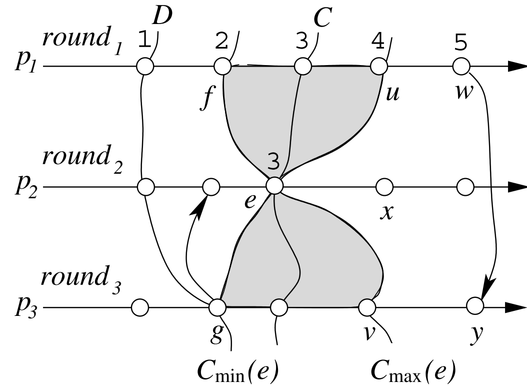


**Fig. 3.** An example of a $p$-region predicate

*Remark 1* Some of the proofs in this paper are presented in calculational style often used by Dijkstra in his proofs. As such, in order to prove a statement of the form "If A holds then B holds", sometimes, starting from B, we provide a series of steps leading up to A; every pair of consecutive steps C and D either satisfies C ≡ D (read as "C is equivalent to D") or C ⇐ D (read as "C follows from D"). The latter, in turn, means that D ⇒ C (read as "D implies C"). □

## 4 Controlling region predicates

In this section, we first define the class of region predicates. We then present an efficient algorithm to compute the synchronization required to control a region predicate in a computation. We further demonstrate that the synchronization generated by our algorithm 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 first define a region predicate with respect to a process, called *$p$-region predicate*. Informally, a $p$-region predicate partitions the set of consistent cuts satisfying the predicate into a set of regions, one for each event on process $p$, satisfying certain properties. Firstly, the set of consistent cuts that "lie" in a region forms a lattice. In other words, given two consistent cuts that belong to the region, the cuts given by set intersection and set union also belong to the region. Secondly, each region is convex[1]. In other words, a consistent cut that "lies" between two consistent cuts contained in the region also belongs to the region. The following example illustrates the concept of $p$-region predicate.

*Example 4* Consider a distributed computation shown in Fig. 3 in which processes execute a sequence of (asynchronous) rounds, and the predicate "processes $p_1$ and $p_2$ are

[1] Mathematically, a region $R$ is said to be convex if given two points $x$ and $y$ in $R$, every point that lies between $x$ and $y$ also belongs to $R$. For instance, sphere is convex but donut is not.

approximately synchronized". The predicate can be expressed mathematically as $|round_1 - round_2| \leqslant \Delta_{12}$ with $\Delta_{12}$ set to 1. Consider the event $e$ on $p_2$ depicted in the figure. Immediately after executing $e$, the value of $round_2$ is 3. Since $round_1$ is monotonically non-decreasing, there exist earliest and latest events on $p_1$, in this case $f$ and $u$, respectively, such that the predicate holds. Furthermore, the predicate holds for every event on $p_1$ that lies between $f$ and $u$. The region corresponding to $e$ (the shaded area resembling the cross-section of an hourglass in the figure) is bounded on the left by the least consistent cut passing through $e$ and $f$ and on the right by the greatest consistent cut passing through $e$ and $u$. The consistent cut $C$ lies in the region whereas the consistent cut $D$ does not. It can be verified that the region is actually convex and the set of consistent cuts that belong to the region forms a lattice. □

A $p$-region predicate is formally defined as follows:

**Definition 1 (p-region predicate)** *A predicate $b$ is a $p$-*region predicate *if it satisfies the following properties. For each event $e$ on process $p$,*

- **(weak lattice)** *If two consistent cuts that pass through $e$ satisfy the predicate then so do the consistent cuts given by their set intersection and set union. Formally,*

$$(e \in frontier(C_1) \cap frontier(C_2)) \wedge (C_1 \models b) \wedge (C_2 \models b) \Rightarrow (C_1 \cap C_2 \models b) \wedge (C_1 \cup C_2 \models b)$$

- **(weak convexity)** *If two consistent cuts that pass through $e$ satisfy the predicate then so does the consistent cut that lies between the two. Formally,*

$$(e \in frontier(C_1) \cap frontier(C_2))) \wedge (C_1 \models b) \wedge (C_2 \models b) \wedge (C_1 \subseteq C \subseteq C_2) \Rightarrow C \models b$$

We call the two properties "weak" because they are only satisfied by those consistent cuts that satisfy the predicate *and pass through a given event*, and not by *all* consistent cuts that satisfy the predicate. (Specifically, the first property is a weaker version of the property described in [13], namely that the set of consistent cuts forms a lattice.) Some examples of $p_i$-region predicates encountered in distributed systems are as follows:

- any local predicate on $p_i$
- "bounded" number of messages in transit from $p_i$ to $p_j$: $send_{ij} - recv_{ij} \leqslant \Delta_{ij}$
- "almost" fair resource allocation between $p_i$ and $p_j$, when the system is heavily loaded: $|alloc_i - alloc_j| \leqslant \Delta_{ij}$
- "bounded" drift between the clocks of $p_i$ and $p_j$: $|clock_i - clock_j| \leqslant \Delta_{ij}$
- $p_i$ and $p_j$ are "approximately" synchronized: $|round_i - round_j| \leqslant \Delta_{ij}$
- $x_i < \min\{y_j, y_k\}$, where $x_i$, $y_j$ and $z_k$ are variables on $p_i$, $p_j$ and $p_k$, respectively, with $y_j$ and $y_k$ monotonically non-decreasing.

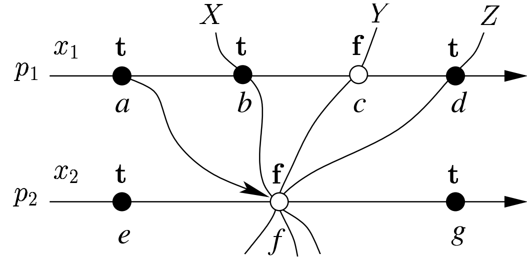


**Fig. 4.** An example to illustrate a non-region predicate

Given two $p$-region predicates, their conjunction is also a $p$-region predicate as established by the next theorem.

**Theorem 1** *The class of $p$-region predicates is closed under conjunction.*

The proof of the theorem can be found in the appendix. A *region predicate* is a conjunction of $p$-region predicates with possibly different $p$'s. It can be verified that the predicate representing termination is actually a region predicate. We next give an example of a predicate that is not a region predicate.

*Example 5* Consider the computation shown in Fig. 4 and the predicate $x_1 \vee x_2$. As shown, $X$ and $Z$ are two consistent cuts satisfying $x_1 \vee x_2$ that pass through the event $f$ on process $p_2$. The consistent cut $Y$ lies between the cuts $X$ and $Z$. However, $Y$ does not satisfy $x_1 \vee x_2$. Clearly, the predicate $x_1 \vee x_2$ does not satisfy the weak convexity property and therefore is not a $p_2$-region predicate. □

Note that, for each $p$, true is a $p$-region predicate. Thus a region predicate $b$ can be written as conjunction of $n$ predicates such that the $i^{th}$ conjunct, denoted by $b^{(i)}$, is a $p_i$-region predicate. Given an event $e$ on process $p_i$, we denote the *least* consistent cut passing through $e$ that satisfies $b^{(i)}$ by $C_{\min}(e)$. Similarly, we denote the *greatest* consistent cut passing through $e$ that satisfies $b^{(i)}$ by $C_{\max}(e)$. If no consistent cut exists that passes through $e$ and satisfies $b^{(i)}$, then neither $C_{\min}(e)$ nor $C_{\max}(e)$ exists. Trivially, $b^{(i)}$ (and hence $b$) cannot be controlled in the computation if $C_{\min}(e)$ and $C_{\max}(e)$ do not exist. However, if there exists at least one consistent cut passing through $e$ that satisfies $b^{(i)}$ then both $C_{\min}(e)$ and $C_{\max}(e)$ exist and are uniquely defined. This is because, from the weak lattice property, the set of such consistent cuts forms a lattice under set containment ($\subseteq$) implying that the set has a minimum (corresponds to $C_{\min}(e)$) and a maximum (corresponds to $C_{\max}(e)$).

### 4.1 Finding a controlling synchronization

In this section, we first derive a necessary and sufficient condition for a region predicate to be controllable in a computation. Specifically, we show that for a region predicate, whenever it is controllable, it is possible to give the *smallest* controlling synchronization. The synchronization is smallest in the sense that it is contained in every other controlling synchronization. Therefore testing for controllability of a region predicate can be reduced to testing for existence of the smallest controlling synchronization. Next, we describe an efficient algorithm to compute the smallest controlling synchronization in case the region predicate is indeed controllable.

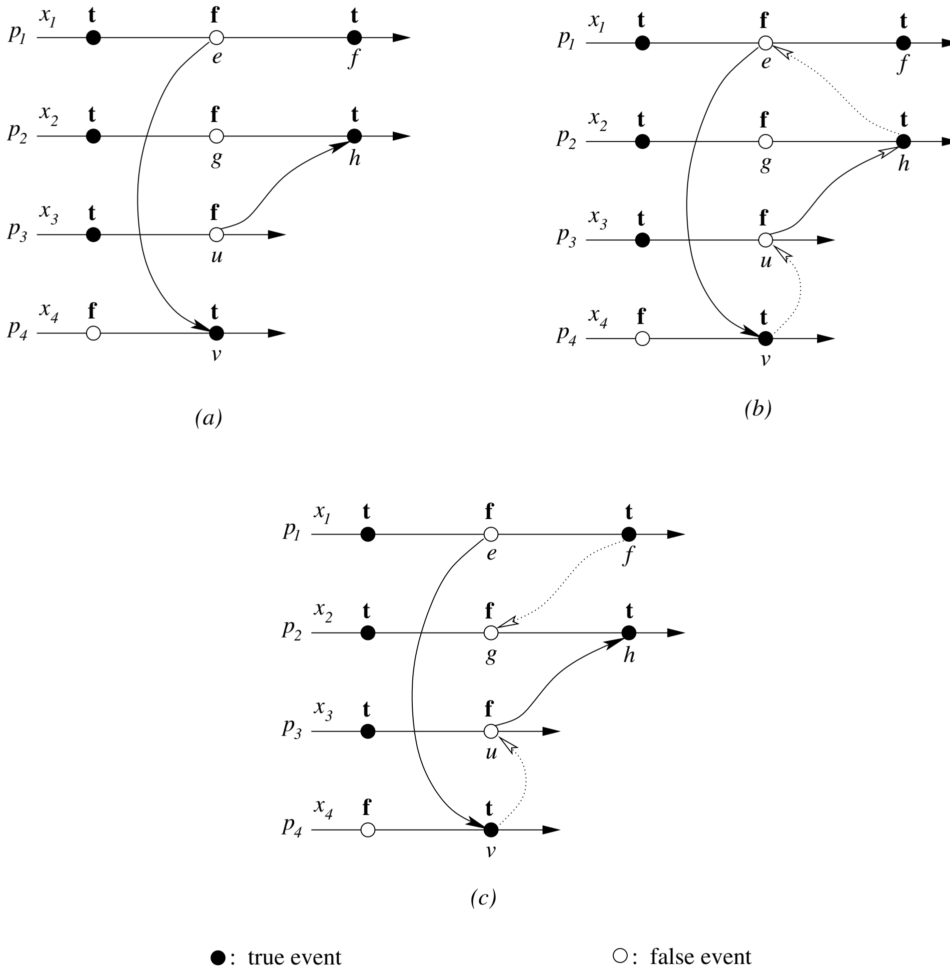


**Fig. 5.** An example to illustrate that a collective synchronization may interfere with the happened-before relation of the computation, even if the predicate can be controlled

4.1.1 Necessary and sufficient condition

To find a synchronization necessary to control a region predicate in a computation, we first compute synchronizations sufficient to control each of its conjunct (recall that the $i^{th}$ conjunct corresponds to a $p_i$-region predicate). If it turns out that one or more of these conjuncts are not controllable then, trivially, the region predicate itself cannot be controlled. Moreover, in case synchronizations for various conjuncts do not interfere with each other and, in addition, the resulting *collective synchronization* does not interfere with the happened-before relation of the computation then, clearly, the collective synchronization constitutes a controlling synchronization for the given region predicate. Such a guarantee, however, cannot be provided in general if controlling synchronizations for various conjuncts are computed independently of each other, even if the predicate is controllable in the computation. This is illustrated by the following example.

*Example 6* Suppose we are interesting in controlling the predicate $(x_1 \vee x_2) \ \wedge \ (x_3 \vee x_4)$ in the computation shown in Fig. 5(a), where each $x_i$ is a boolean variable on process $p_i$. It can be verified that the arrow from event $h$ to event $e$ constitutes a controlling synchronization for the first conjunct $x_1 \vee x_2$. Similarly, the arrow from event $v$ to event $u$ constitutes a controlling synchronization for the second conjunct $x_3 \vee x_4$. However, the collective synchronization given by $\{(h, e), (v, u)\}$ interferes with the happened-before relation of the computation. In other words, it creates a cycle as shown in Fig. 5(b). The first conjunct has another controlling synchronization, namely the arrow from event $f$ to event $g$. In this case, the collective synchronization given by $\{(f, g), (v, u)\}$ neither interferes with itself nor with the happened-before relation of the computation, thereby constituting a controlling synchronization for the predicate $(x_1 \vee x_2) \ \wedge \ (x_3 \vee x_4)$. □

Now, suppose that the synchronization computed for each conjunct is *smallest* in the sense that it is contained in every possible controlling synchronization for the respective conjunct. In that case, it can be shown that if it is possible to control the predicate in the computation then indeed not only do various synchronizations not interfere with each other but the resulting collective synchronization does not interfere with the happened-before relation of the computation as well. Intuitively, this is because a controlling synchronization for a region predicate also acts as a controlling synchronization for each of its conjunct. We formally define the notion of smallest controlling synchronization next.

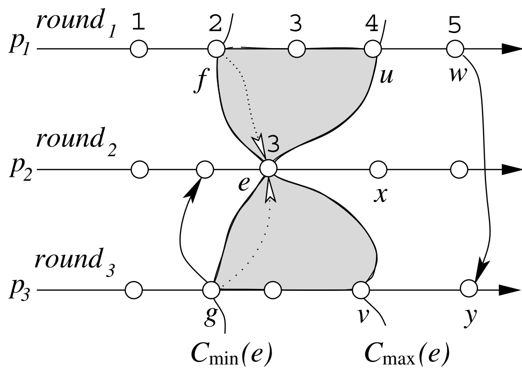


**Fig. 6.** An illustration of the synchronization $\overset{e(1)}{\rightarrow}$ (denoted by dotted arrows)

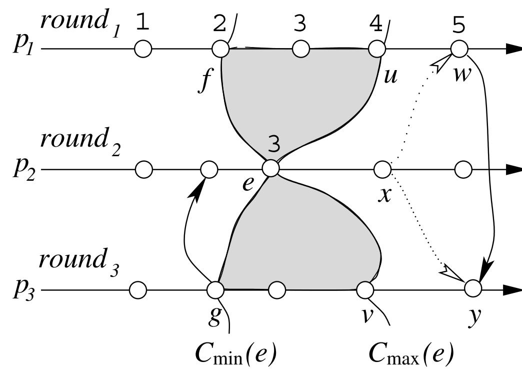


**Fig. 7.** An illustration of the synchronization $\overset{e(2)}{\rightarrow}$ (denoted by dotted arrows)

**Definition 2 (smallest controlling synchronization)** *A controlling synchronization is said to be* smallest *if it is contained in every possible controlling synchronization for the predicate. Formally, given a controlling synchronization $\overset{S}{\rightarrow}$ for a predicate $b$ in a computation $\langle E, \rightarrow \rangle$,*

$$\overset{S}{\rightarrow} \text{ is smallest}$$
$$\triangleq$$
$$\langle \forall \rightsquigarrow : \rightsquigarrow \text{ extends } \rightarrow :$$
$$\langle E, \rightsquigarrow \rangle \models \mathit{invariant} : b \equiv \rightsquigarrow \text{ contains } \overset{S}{\rightarrow} \rangle$$

The smallest controlling synchronization, whenever it exists, is uniquely defined. Suppose $\overset{S_1}{\rightarrow}$ and $\overset{S_2}{\rightarrow}$ are two smallest controlling synchronizations for a given computation and region predicate. Then, from the definition, $\overset{S_1}{\rightarrow} \subseteq \overset{S_2}{\rightarrow}$ and $\overset{S_2}{\rightarrow} \subseteq \overset{S_1}{\rightarrow}$. This implies that $\overset{S_1}{\rightarrow}$ and $\overset{S_2}{\rightarrow}$ are identical. Of course, the smallest controlling synchronization may not always exist. As it happens, the smallest controlling synchronization in fact exists for a $p$-region predicate (and therefore also exists for a region predicate). Thus in order to find a controlling synchronization for a region predicate, from the above discussion, it suffices to devise an algorithm to compute the smallest controlling synchronization for a $p$-region predicate.

Consider a computation $\langle E, \rightarrow \rangle$ and a region predicate $b$. What does it entail to control the $p_i$-region predicate $b^{(i)}$ in $\langle E, \rightarrow \rangle$ where $1 \leqslant i \leqslant n$? Consider an event $e$ on process $p_i$. Assume that there is at least one consistent cut that passes through $e$ and satisfies $b^{(i)}$, which implies that both $C_{\min}(e)$ and $C_{\max}(e)$ exist. As we know, the computation progresses from the initial consistent cut $\perp$ to the final consistent cut $E$ by executing, one-by-one, the events in $E$. For $b^{(i)}$ to hold when it first reaches $e$, it must be the case that no event in the frontier of the computation lies before the frontier of $C_{\min}(e)$. That is, when $e$ is executed, all other events in the frontier of $C_{\min}(e)$ must have already been executed. This entails adding synchronization dependencies from each event in the frontier of $C_{\min}(e)$ that is different from $e$ to $e$. We denote this synchronization by $\overset{e(1)}{\rightarrow}$ and formally define it as follows:

$$\overset{e(1)}{\rightarrow} \triangleq \{ (f, e) \mid f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} \text{ and } e \notin \perp \}$$

For an example refer to Fig. 6. Furthermore, for $b^{(i)}$ to hold as long as the computation stays at $e$ (equivalently, until the successor of $e$, if it exists, is executed), the frontier of the computation cannot advance beyond $C_{\max}(e)$. That is, the successor of any event in the frontier of $C_{\max}(e)$ that is different from $e$, if it exists, cannot be executed until the computation advances beyond $e$. This involves adding synchronization dependencies from the successor of $e$, if it exists, to the successor of every other event in the frontier of $C_{\max}(e)$, if it exists. We denote this synchronization by $\overset{e(2)}{\rightarrow}$ and formally define it as follows:

$$\overset{e(2)}{\rightarrow}$$
$$\triangleq$$
$$\{ (succ(e), succ(f)) \mid$$
$$f \in \mathit{frontier}(C_{\max}(e)) \setminus \{e\} \text{ and } \{e, f\} \cap \top = \emptyset \}$$

For an illustration see Fig. 7. The overall synchronization needed for controlling $b^{(i)}$ in $\langle E, \rightarrow \rangle$ is given by the union of $(\overset{e(1)}{\rightarrow} \cup \overset{e(2)}{\rightarrow})$, where $e$ ranges over the events on process $p_i$. Finally, the synchronization required to control $b$ in $\langle E, \rightarrow \rangle$, denoted by $\overset{S}{\rightarrow}$, is given by:

$$\overset{S}{\rightarrow} \triangleq \bigcup_{e \in E} (\overset{e(1)}{\rightarrow} \cup \overset{e(2)}{\rightarrow}) \tag{1}$$

For convenience, we use $\overset{C}{\rightarrow}$ to denote the transitive closure of the relation obtained by adding $\overset{S}{\rightarrow}$ to $\rightarrow$. Formally,

$$\overset{C}{\rightarrow} \triangleq (\rightarrow \cup \overset{S}{\rightarrow})^{+} \tag{2}$$

Clearly, $\overset{S}{\rightarrow}$ and $\overset{C}{\rightarrow}$ exist only if $C_{\min}(e)$ and $C_{\max}(e)$ exist for all events $e$. The next lemma describes the sufficient condition under which a region predicate is controllable in a computation. Informally, this happens when each of its conjunct is controllable and the collective synchronization neither interferes with itself nor with the happened-before relation of the computation—which can be succinctly represented as: $(\rightarrow \cup \overset{S}{\rightarrow})$ is acyclic.

**Lemma 2 (sufficient condition)** *If (1) the initial and final consistent cuts of a computation $\langle E, \rightarrow \rangle$ satisfy a region predicate $b$, and (2) $\overset{C}{\rightarrow}$ defined in (2) exists and is an irreflexive partial order then $b$ is invariant in $\langle E, \overset{C}{\rightarrow} \rangle$.*

*Proof.* Consider a consistent cut $C$ of $\langle E, \stackrel{C}{\to}\rangle$ and an event $e$ contained in its frontier. We show that $C$ lies between $C_{\min}(e)$ and $C_{\max}(e)$. We first prove that $C_{\min}(e) \subseteq C$. If $e \in \bot$ then $C_{\min}(e) = \bot$ because, trivially, $\bot$ is the least consistent cut of $\langle E, \to\rangle$ that passes through $e$ and $\bot \models b$. Furthermore, by definition of consistent cut, $C \supseteq \bot$. Thus $C_{\min}(e) \subseteq C$. The more interesting case is when $e \notin \bot$. We want to prove that,

$$
\begin{array}{ll}
& C_{\min}(e) \subseteq C \\
\equiv & \{ \text{ definition of consistent cut and its frontier } \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) : f \in C \rangle \\
\equiv & \{ \text{ by definition, } C_{\min}(e) \text{ passes through } e \text{ } \} \\
& (e \in C) \wedge \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : f \in C \rangle \\
\Leftarrow & \{ \text{ } C \text{ is a consistent cut of } \langle E, \stackrel{C}{\to}\rangle \text{ } \} \\
& (e \in C) \wedge \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : f \stackrel{C}{\to} e \rangle \\
\Leftarrow & \{ \text{ } C \text{ passes through } e \text{ } \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : (f \stackrel{C}{\to} e) \rangle \\
\Leftarrow & \{ \stackrel{S}{\to} \subseteq \stackrel{C}{\to} \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : (f \stackrel{S}{\to} e) \rangle \\
\Leftarrow & \{ \stackrel{e(1)}{\to} \subseteq \stackrel{S}{\to} \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : (f \stackrel{e(1)}{\to} e) \rangle \\
& \{ \text{ } e \notin \bot \text{ and definition of } \stackrel{e(1)}{\to} \} 
\end{array}
$$

Likewise, $C \subseteq C_{\max}(e)$. Let $proc(e) = p_i$. By definition, both $C_{\min}(e)$ and $C_{\max}(e)$ satisfy $b^{(i)}$. Thus, from the weak convexity property, $C$ satisfies $b^{(i)}$. Since $e$ was chosen arbitrarily, for each $i$, we can infer that $C$ satisfies $b^{(i)}$. This implies that $C$ satisfies $b$. □

The next lemma proves that the synchronization given by $\stackrel{S}{\to}$ is indeed the smallest controlling synchronization for $b$ in $\langle E, \to\rangle$. In other words, any other controlling synchronization for $b$ in $\langle E, \to\rangle$, if it exists, must contain $\stackrel{S}{\to}$.

**Theorem 3** *If a region predicate $b$ is controllable in a computation $\langle E, \to\rangle$ then the synchronization $\stackrel{S}{\to}$ defined in (1) is the smallest controlling synchronization.*

*Proof.* Since $b$ is controllable in $\langle E, \to\rangle$, there exists an irreflexive partial order $\leadsto$ that extends $\to$ such that $b$ is invariant in $\langle E, \leadsto\rangle$. We need to prove that $\stackrel{S}{\to}$ is contained in $\leadsto$. It is sufficient to prove that, for each event $e$, both $\stackrel{e(1)}{\to}$ and $\stackrel{e(2)}{\to}$ are contained in $\leadsto$.

We first show that, for each event $e$, $\leadsto$ includes $\stackrel{e(1)}{\to}$. Consider an event $e$, $e \notin \bot$, on process $p_i$. Note that if $e \in \bot$ then $\stackrel{e(1)}{\to}$ is an empty set. In the proof we use the notion of the *least* consistent cut of $\langle E, \leadsto\rangle$ that contains $e$, denoted by $C_{least}(e)$. By definition, $C_{least}(e)$ passes through $e$ and an event other than $e$ belongs to $C_{least}(e)$ if and only if it happened-before $e$ in $\langle E, \leadsto\rangle$. Formally,

$$
\begin{array}{l}
(e \in \mathit{frontier}(C_{least}(e))) \wedge \\
\langle \forall f : f \neq e : f \in C_{least}(e) \equiv f \leadsto e \rangle
\end{array} \tag{3}
$$

We want to prove that,

$$
\begin{array}{ll}
& \stackrel{e(1)}{\to} \subseteq \leadsto \\
\equiv & \{ \text{ definition of } \stackrel{e(1)}{\to} \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : f \leadsto e \rangle \\
\equiv & \{ \text{ using (3) } \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}(e)) \setminus \{e\} : f \in C_{least}(e) \rangle \\
\Leftarrow & \{ \text{ definition of consistent cut and its frontier } \} \\
& C_{\min}(e) \subseteq C_{least}(e) \\
\Leftarrow & \left\{ \begin{array}{l} C_{least}(e) \text{ is a consistent cut of } \langle E, \to\rangle \\ \text{that passes through } e \text{ and satisfies } b^{(i)} \text{ and} \\ C_{\min}(e) \text{ is the } \textit{least} \text{ such cut} \end{array} \right\} \\
& (C_{least}(e) \text{ is a consistent cut of } \langle E, \to\rangle) \wedge \\
& (e \in \mathit{frontier}(C_{least}(e))) \wedge (C_{least}(e) \models b^{(i)}) \\
\Leftarrow & \{ \text{ } C_{least}(e) \text{ is a consistent cut of } \\
& \langle E, \leadsto\rangle \text{ and } \to \subseteq \leadsto \} \\
& (e \in \mathit{frontier}(C_{least}(e))) \wedge (C_{least}(e) \models b^{(i)}) \\
\Leftarrow & \{ \text{ using (3) } \} \\
& C_{least}(e) \models b^{(i)} \\
\Leftarrow & \{ \text{ } b^{(i)} \text{ is a conjunct of } b \text{ } \} \\
& C_{least}(e) \models b \\
& \{ \text{ since } b \text{ is invariant in } \langle E, \leadsto\rangle, C_{least}(e) \text{ satisfies } b \text{ } \}
\end{array}
$$

Similarly, it can be proved that, for each event $e$, $\leadsto$ includes $\stackrel{e(2)}{\to}$. □

The necessary condition for a region predicate to be controllable in a computation can now be easily derived.

**Lemma 4 (necessary condition)** *If a region predicate $b$ is controllable in a computation $\langle E, \to\rangle$ then (1) the initial and final consistent cuts of $\langle E, \to\rangle$ satisfy b, and (2) $\stackrel{C}{\to}$ defined in (2) exists and is an irreflexive partial order.*

*Proof.* Since $b$ is controllable in $\langle E, \to\rangle$, $C_{\min}(e)$ and $C_{\max}(e)$ exist for all events $e$, which implies that $\stackrel{C}{\to}$ exists as well. Furthermore, there exists an irreflexive partial order $\leadsto$ that extends $\to$ such that $b$ is invariant in $\langle E, \leadsto\rangle$. Since $\bot$ and $E$ are also the consistent cuts of $\langle E, \leadsto\rangle$, they satisfy $b$. Furthermore, from Theorem 3, $\stackrel{S}{\to}$ is the smallest controlling synchronization implying that $\leadsto$ contains $\stackrel{S}{\to}$. Thus $\leadsto$ contains $(\to \cup \stackrel{S}{\to})$. Since $\leadsto$ is an irreflexive partial order, $(\to \cup \stackrel{S}{\to})^{+}$ $(= \stackrel{C}{\to})$ is also an irreflexive partial order. □

Finally, the next theorem combines the previous two lemmas and furnishes the necessary and sufficient condition for a region predicate to be controllable in a computation.

**Theorem 5 (necessary and sufficient condition)** *A region predicate $b$ is controllable in a computation $\langle E, \to\rangle$ if and*

*only if (1) the initial and final consistent cuts of* $\langle E, \rightarrow \rangle$ *satisfy b, and (2)* $\stackrel{C}{\rightarrow}$ *defined in (2) exists and is an irreflexive partial order.*

It turns out that the controlling synchronization $\stackrel{S}{\rightarrow}$ defined in (1) is minimal in another sense. It not only eliminates all unsafe runs of the computation but also does not suppress any safe run. We call such a synchronization *optimal*.

**Definition 3 (optimal controlling synchronization)** *We call a controlling synchronization* optimal *if it does not suppress any safe run of the computation. Formally, given a controlling synchronization* $\stackrel{S}{\rightarrow}$ *for a predicate b in a computation* $\langle E, \rightarrow \rangle$,

$$\stackrel{S}{\rightarrow} \text{ is optimal}$$
$$\triangleq$$
$$\langle \forall \leadsto : \langle E, \leadsto \rangle \text{ is a run of } \langle E, \rightarrow \rangle : \langle E, \leadsto \rangle \models invariant : b \equiv \langle E, \leadsto \rangle \text{ is a run of } \langle E, \stackrel{C}{\rightarrow} \rangle \rangle$$

*where* $\stackrel{C}{\rightarrow} = (\rightarrow \cup \stackrel{S}{\rightarrow})^{+}$.

In fact, the two aforementioned notions of minimality, namely the smallest and the optimal controlling synchronization, turn out to be identical. We establish their equivalence in the next theorem.

**Theorem 6 (smallest versus optimal)** *The smallest controlling synchronization is also optimal and vice versa.*

The proof is given in the appendix. From Theorem 3 and Theorem 6, we obtain,

**Theorem 7** *If a region predicate b is controllable in a computation* $\langle E, \rightarrow \rangle$ *then the synchronization* $\stackrel{S}{\rightarrow}$ *defined in (1) is the optimal controlling synchronization.*

Theorem 7 implies that the controlling synchronization $\stackrel{S}{\rightarrow}$ defined in (1) is not too restrictive and, in fact, admits the maximum possible concurrency in the controlled computation.

#### 4.1.2 Computing $\mathbf{C_{min}(e)}$ and $\mathbf{C_{max}(e)}$

From the earlier discussion, it follows that a controlling synchronization for a region predicate can be easily computed provided, for each event $e$, we can efficiently compute $C_{\min}(e)$ and $C_{\max}(e)$, if they exist. To that end, given a $p$-region predicate $b$ and an event $e$ on process $p$, we define a predicate $b_e$ to be true for a consistent cut if it passes through $e$ and satisfies $b$. Formally,

$$C \models b_e \triangleq (e \in frontier(C)) \wedge (C \models b)$$

It can be verified easily, using the weak lattice property, that if two consistent cuts satisfy $b_e$ then so does the consistent cut given by their set intersection. Chase and Garg [2] call such predicates *linear*. Likewise, if two consistent cuts satisfy $b_e$ then the consistent cut given by their set union also satisfies $b_e$. Such predicates are called *post-linear* [2].

**Observation 1** *The predicate* $b_e$ *is linear and post-linear.*

The consistent cuts $C_{\min}(e)$ and $C_{\max}(e)$ can be reinterpreted as the least and greatest consistent cut, respectively, that satisfy $b_e$. Chase and Garg [2] also provide algorithms to find the least consistent cut that satisfies a linear predicate and the greatest consistent cut that satisfies a post-linear predicate. Here, we focus on the former and give the basic idea behind the algorithm. The correctness proof and other details can be found elsewhere [2]. The algorithm is based on the *linearity property* which is defined as follows:

**Definition 4 (linearity property [2])** *A predicate satisfies the* linearity property *if, given a consistent cut that does not satisfy the predicate, there exists an event in its frontier, called the* forbidden *event, such that there does not exist a consistent cut containing the given consistent cut that satisfies the predicate and also passes through the forbidden event. Formally, given a computation* $\langle E, \rightarrow \rangle$*, a linear predicate b and a consistent cut C,*

$$C \not\models b$$
$$\Rightarrow$$
$$\langle \exists f : f \in frontier(C) : \langle \forall D : D \supseteq C : D \models b \Rightarrow succ(f) \in D \rangle \rangle$$

It is assumed that, given a linear predicate $b$, there is an efficient partial function $forbidden_b : \mathcal{C}(\langle E, \rightarrow \rangle) \longrightarrow E$, where $\mathcal{C}(\langle E, \rightarrow \rangle)$ denote the set of consistent cuts of $\langle E, \rightarrow \rangle$, that can be used to compute the event $f$ mentioned in the definition of the linearity property. It is hard to provide a general algorithm to compute the function that works for any linear predicate. Nevertheless, for the linear predicates encountered in practice, an efficient algorithm can indeed be given. For example, for a conjunctive predicate—a conjunction of local predicates—the forbidden event corresponds to that event in the cut's frontier for which the local predicate evaluates to false. Throughout this paper, we assume that a linear predicate also satisfies the *advancing property* which guarantees the existence of an efficient function to compute the forbidden event.

We now informally describe the algorithm to determine $C_{\min}(e)$ for an event $e$. Starting from the least consistent cut that passes through $e$—which basically corresponds to the Fidge/Mattern's vector timestamp for $e$ [13,5], the algorithm scans the computation from left to right adding events to the cut constructed so far one-by-one, until either the desired consistent cut is reached or all events have been exhausted. At each step, an event is added to the cut, if at all, because of one of the following two reasons. First, the cut constructed so far is not consistent. In that case, its frontier contains events $f$ and $g$ such that $succ(f) \rightarrow g$. Clearly, as long as the computation does not advance beyond $f$ on $proc(f)$, the cut stays inconsistent. Therefore the next event to be added is given by $succ(f)$. Second, the cut is consistent but does not satisfy the region predicate. In that case, the next event to be added is determined using the linearity property. The time-complexity of the algorithm is $O(n|E|)$ [2]. The algorithm to compute $C_{\max}(e)$, based on the *post-linearity property* [2], is similar and has been omitted.

```
     Input: a computation ⟨E, →⟩ and a region predicate b
     Output: a controlling synchronization for b in ⟨E, →⟩, if it exists

1    if either ⊥ or E does not satisfy b then
2        exit("b cannot be controlled in ⟨E, →⟩");
     endif;
3    for each event e do
4        compute C_min(e) and C_max(e);
5        if either C_min(e) or C_max(e) does not exist then
6            exit("b cannot be controlled in ⟨E, →⟩");
         endfor;
     endfor;
7    compute the synchronization -S-> defined in (1);
8    if (→ ∪ -S->) is acyclic then
9        exit(-S->);
     else
10       exit("b cannot be controlled in ⟨E, →⟩");
     endif;
```

**Fig. 8.** The algorithm ControlRegionPredicate to compute a controlling synchronization for a region predicate

Figure 8 depicts the algorithm ControlRegionPredicate for computing a synchronization to control a region predicate in a computation. The algorithm first tests whether the initial and final consistent cuts satisfy the region predicate (at line 1). It then computes $C_{\min}(e)$ and $C_{\max}(e)$ for all events $e$ (at line 4). Finally, it computes $\xrightarrow{S}$ defined in (1) (at line 7) and checks whether it interferes with $\rightarrow$ (at line 8). The correctness of the algorithm follows from Theorem 5. Its time-complexity analysis is as follows. The time-complexity of executing the if statement at line 1 is $O(n)$. Each iteration of the for loop at line 3 has $O(n|E|)$ time-complexity giving the for loop an overall time-complexity of $O(n|E|^2)$. The synchronization at line 7 can be computed in $O(n|E|)$ time. Finally, the if statement at line 8 can be executed in $O(|E|^2)$ time. Thus the overall time-complexity of the algorithm ControlRegionPredicate is $O(n|E|^2)$. It is easy to see that the message-complexity of the algorithm is $O(n|E|)$.

## 5 Controlling disjunctive predicates

In this section, we first introduce the notion of admissible sequence of events and establish that existence of such a sequence is a necessary and sufficient condition for controllability of a predicate in a computation. Using the notion of admissible sequence, we then derive an efficient control algorithm for a disjunctive predicate. We further modify the algorithm to generate a controlling synchronization with the *least* number of synchronization dependencies, that is, with the *optimal* message-complexity.

A predicate is said to be *disjunctive* if it can be expressed as disjunction of local predicates. Some examples of disjunctive predicates are:

- at least one server is available: $avail_1 \vee avail_2 \vee \cdots \vee avail_n$
- at least one philosopher has no fork: $\neg fork_1 \vee \neg fork_2 \vee \cdots \vee \neg fork_n$

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. Our algorithm for computing a controlling synchronization for a disjunctive predicate utilizes the notion of admissible sequence defined next.

### *5.1 Admissible sequences*

In this section, we establish that the notion of controllability is actually identical to the notion of admissible sequence whose motivation in turn lies in the control algorithm for a disjunctive predicate. We make the following observation:

**Observation 2** *A consistent cut satisfies a disjunctive predicate if and only if it contains at least one true event in its frontier.*

Suppose we wish to control a disjunctive predicate in a computation. As the computation proceeds from the initial consistent cut to the final consistent cut, from the above observation it follows that it is both necessary and sufficient to ensure that throughout there exists at least one true event in the frontier of the computation. Thus at least one initial event must be a true event. To start with, one such initial event bears the responsibility for ensuring that the predicate stays true—by acting as an *anchor*—until the burden can be passed on to some other true event. This transference of burden continues until the computation reaches the final consistent cut. This is illustrated by the following example.

*Example 7* We want to control the disjunctive predicate $x_1 \vee x_2$ in the computation depicted in Fig. 9. The initial event $e$ is a true event. Hence, using $e$ as an anchor, the computation advances from the initial consistent cut $C_1$, shown in Fig. 9(a), to the consistent cut $C_2$, portrayed in Fig. 9(b). Next, using the true event $f$ as an anchor, it advances to the consistent cut $C_3$ as shown in Fig. 9(c). Finally, using the true event $g$ as an anchor—which is also a final event, it reaches the final consistent cut $C_4$ as depicted in Fig. 9(d). Since, throughout, the frontier of the computation passes through at least one true event, the predicate is never falsified. □

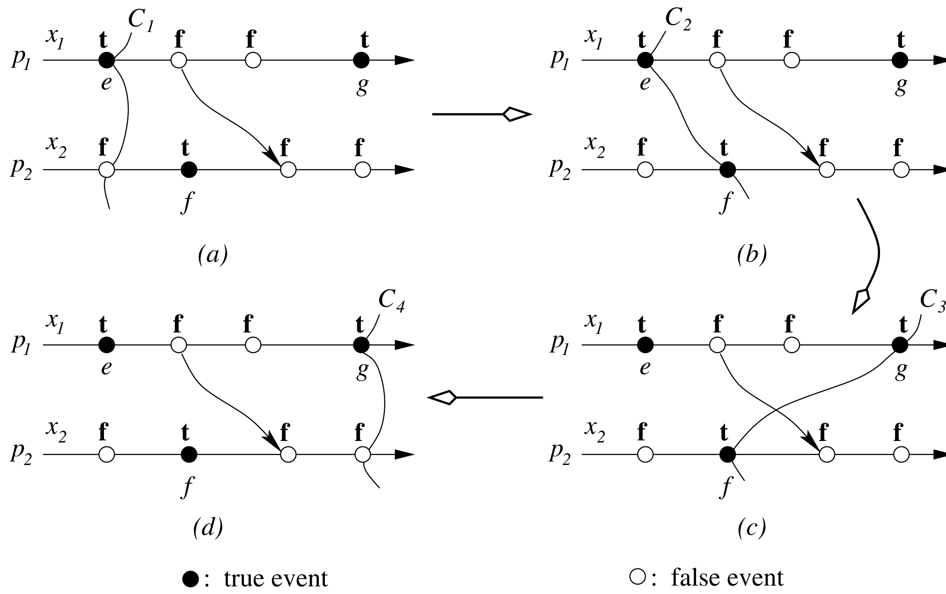


**Fig. 9.** A strategy for controlling a disjunctive predicate

A natural question to ask is: *"If there are more than one possible candidates for the next anchor event, which one should we choose?"*. The answer is non-trivial as illustrated by the following example.

*Example 8* Consider the computation shown in Fig. 10. It has four true events, namely $e$, $f$, $g$ and $h$. After using $e$ as an anchor, the computation has two possible choices of events for the next anchor. They are the events $f$ and $g$. The event $h$ is unavailable because the computation has to advance beyond $e$ before it can execute $h$. Clearly, $f$ is a bad choice for anchor because once the computation reaches the consistent cut $C$, using $f$ as an anchor, neither $g$ nor $h$ can be used as the next anchor without falsifying the predicate. □

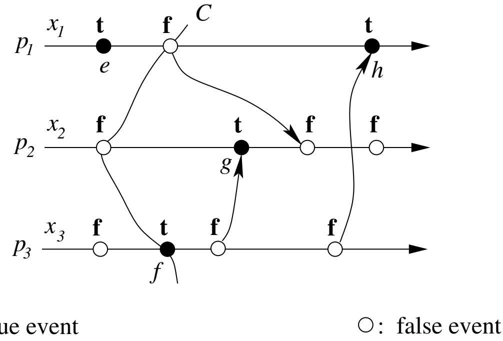


**Fig. 10.** An example to illustrate the difficulty in choosing the next anchor event

The notion of admissible sequence attempts to answer the above question in a more general setting. In the next section, we formalize the aforementioned algorithm for controlling a disjunctive predicate using the notion of admissible sequence. We first define a legal cut as follows:

**Definition 5 (legal cut)** *A consistent cut is* legal *with respect to a sequence of events if it contains an event from the sequence only if it contains all its preceding events from the sequence as well. Formally, given a consistent cut $C$ and an event $s_i$ from a sequence of events $s$,*

$$s_i \in C \;\Rightarrow\; \langle \forall\, j : j \leqslant i : s_j \in C \rangle$$

Roughly speaking, the notion of legal cut helps to capture those runs of a computation that respect the order of the events in a sequence. More precisely, given a sequence of events, if every consistent cut of a run is legal then the run and the sequence do not disagree on relative order of any pair of events and vice versa. We next define the notion of admissible sequence. Informally, every event in an admissible sequence acts as an anchor in the order given by the sequence. To be able to do so, the sequence must respect the happened-before order between events. This constraint is captured by the *agreement property*. The *continuity property* ensures that the transfer of burden from one event in the sequence to the next occurs "smoothly" in a single step. In other words, the computation does not advance beyond the current anchor event until it reaches the next anchor event. The *weak safety property* ascertains that, on reaching an anchor event, at least as long as the computation does not advance beyond the event the predicate is not falsified. Finally, the *boundary condition* captures the fact that the initial and final consistent cuts satisfy the predicate. Formally,

**Definition 6 (admissible sequence)** *A sequence of events $s = s_1 s_2 \cdots s_{l-1} s_l$ is* admissible *with respect to a predicate $b$ and a computation $\langle E, \rightarrow \rangle$ if it satisfies the following properties:*

- **(boundary condition)** *The sequence starts with an initial event and ends with a final event of the computation. Formally,*

$$(s_1 \in \bot) \;\wedge\; (s_l \in \top)$$

- **(agreement)** *The sequence respects the partial order (that is, happened-before relation) of the computation. Formally,*

$$\langle \forall\, i, j : i < j : s_j \not\rightarrow s_i \rangle$$

- **(continuity)** *The successor of each event in the sequence, if it exists, did not happen-before the next event in the sequence. Formally,*

$$\langle \forall\, i : s_i \notin \top : succ(s_i) \not\rightarrow s_{i+1} \rangle$$

- **(weak safety)** *Any consistent cut of the computation that is legal with respect to the sequence and contains at least one event from the sequence in its frontier satisfies the predicate. Formally,*

$$\langle \forall C : C \textit{ is legal with respect to } s : (s \cap \mathit{frontier}(C)) \neq \emptyset \;\Rightarrow\; C \models b \rangle$$

The next example illustrates the notions of legal cut and admissible sequence (along with the associated properties).

*Example 9* Consider the computation depicted in Fig. 11. The consistent cut $C$ is not legal with respect to the sequence of events $e f u\, v\, h$ because it contains $u$ but does not contain $f$ which occurs before $u$ in the sequence. On the other hand, the consistent cut $D$ is legal with respect to the same sequence. The sequence $f u\, v\, h$ does not satisfy the boundary condition because the first event in the sequence, in this case $f$, is not an initial event. The sequence $egfh$ does not satisfy the agreement property because although $f$ happened-before $g$ in the computation, it occurs after $g$ in the sequence. Finally, the sequence $e\, g\, h$ does not satisfy the continuity property as the successor of $e$, namely $f$, happened-before $g$, the next event in the sequence after $e$. □

The following theorem proves that existence of an admissible sequence is necessary for a predicate to be controllable in a computation. Specifically, we prove that any safe run of a computation constitutes an admissible sequence.

**Theorem 8 (necessary condition)** *If a predicate $b$ can be controlled in a computation $\langle E, \rightarrow \rangle$ then there exists an admissible sequence with respect to $b$ and $\langle E, \rightarrow \rangle$.*

*Proof.* Since $b$ is controllable in $\langle E, \rightarrow \rangle$, there exists a total order $\rightsquigarrow$ that extends $\rightarrow$ such that $b$ is invariant in $\langle E, \rightsquigarrow \rangle$. Let $s$ be the sequence of events corresponding to $\langle E, \rightsquigarrow \rangle$. We prove that $s$ is admissible with respect to $b$ and $\langle E, \rightarrow \rangle$. Clearly, $s$ satisfies the boundary condition and the agreement property. We next prove that $s$ satisfies the continuity property. Assume the contrary. Then,

$$\begin{aligned}
& \langle \exists\, i :: succ(s_i) \rightarrow s_{i+1} \rangle \\
\equiv\;\; & \{\, s_i \rightarrow succ(s_i) \,\} \\
& \langle \exists\, i :: s_i \rightarrow succ(s_i) \rightarrow s_{i+1} \rangle \\
\Rightarrow\;\; & \left\{ \begin{array}{l} succ(s_i) \in s \text{ because } s \text{ corresponds to } \langle E, \rightsquigarrow \rangle \\ \text{which is a run of } \langle E, \rightarrow \rangle \end{array} \right\} \\
& \langle \exists\, i, j :: s_i \rightarrow s_j \rightarrow s_{i+1} \rangle \\
\Rightarrow\;\; & \{\, s \text{ satisfies the agreement property} \,\} \\
& \langle \exists\, i, j :: i < j < i+1 \rangle \\
\Rightarrow\;\; & \{\, i \text{ and } j \text{ are integers} \,\} \\
& \text{a contradiction}
\end{aligned}$$

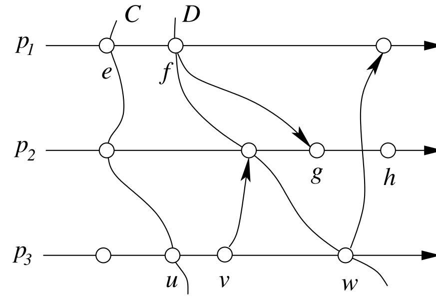


**Fig. 11.** An example to illustrate the notions of legal cut and admissible sequence

Finally, we show that $s$ satisfies the weak safety property. Consider a consistent cut $C$ of $\langle E, \rightarrow \rangle$ that is legal with respect to $s$. We prove that $C$ is also a consistent cut of $\langle E, \rightsquigarrow \rangle$. Consider events $e$ and $f$. We have,

$$\begin{aligned}
& \{\, \text{assumption} \,\} \\
& (e \rightsquigarrow f) \;\wedge\; (f \in C) \\
\equiv\;\; & \{\, \text{let } e = s_i \text{ and } f = s_j \,\} \\
& (s_i \rightsquigarrow s_j) \;\wedge\; (s_j \in C) \\
\Rightarrow\;\; & \{\, \text{definition of } s \,\} \\
& (i < j) \;\wedge\; (s_j \in C) \\
\Rightarrow\;\; & \{\, C \text{ is legal with respect to } s \,\} \\
& s_i \in C \\
\equiv\;\; & \{\, s_i = e \,\} \\
& e \in C
\end{aligned}$$

Thus $C$ is a consistent cut of $\langle E, \rightsquigarrow \rangle$. Since $b$ is invariant in $\langle E, \rightsquigarrow \rangle$, $C$ satisfies $b$. This establishes that $s$ satisfies the weak safety property. □

Our next step is to prove that the existence of an admissible sequence is also a sufficient condition for a predicate to be controllable in a computation. To achieve that it suffices to give the synchronization necessary to control the predicate. Of course the synchronization will depend on the particular sequence. Observe that not all events in the sequence may be ordered by the happened-before relation. Thus, to ensure that they are executed in the order they occur in the sequence, we need to add synchronization dependencies from an event in the sequence to all other events that occur later in the sequence. This synchronization is denoted by $\overset{S(1)}{\rightarrow}$ and is formally defined as follows:

$$\overset{S(1)}{\rightarrow} \;\triangleq\; \{\, (s_i, s_j) \mid 1 \leqslant i < j \leqslant n \,\} \tag{4}$$

For an example please refer to Fig. 12. In the following lemma we show that if the sequence is admissible, in particular if it satisfies the agreement property, the above synchronization does not interfere with the happened-before relation of the computation. For convenience, we define $\overset{C(1)}{\rightarrow}$ as the transitive closure of $\rightarrow \cup \overset{S(1)}{\rightarrow}$. Formally,

$$\overset{C(1)}{\rightarrow} \;\triangleq\; \left(\rightarrow \cup \overset{S(1)}{\rightarrow}\right)^{+}$$

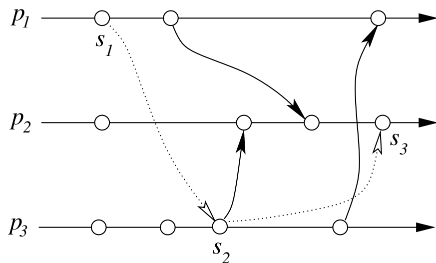


**Fig. 12.** An illustration of the synchronization $\stackrel{S^{(1)}}{\to}$ (denoted by dotted arrows)

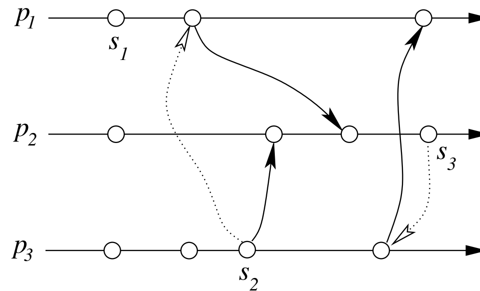


**Fig. 13.** An illustration of the synchronization $\stackrel{S^{(2)}}{\to}$ (denoted by dotted arrows)

**Lemma 9** $\stackrel{C^{(1)}}{\to}$ *is an irreflexive partial order.*

The proof is given in the appendix. After adding the synchronization $\stackrel{S^{(1)}}{\to}$ to the computation $\langle E, \to \rangle$, the resulting computation $\langle E, \stackrel{C^{(1)}}{\to} \rangle$ retains only those consistent cuts—not necessarily all—that are legal. From the weak safety property, a sufficient condition for a legal cut to satisfy the predicate is that it should contain at least one event from the sequence in its frontier. To ensure this, given an event in the sequence, we add a synchronization arrow from the event next to it in the sequence, if it exists and is on a different process, to its succeeding event on the process, if it exists. This synchronization, denoted by $\stackrel{S^{(2)}}{\to}$, ascertains that the computation does not advance beyond an event in the sequence until it reaches the next event in the sequence.

$$\stackrel{S^{(2)}}{\to} \triangleq \{ (s_{i+1}, succ(s_i)) \mid 1 \leqslant i < n,\ s_i \notin \top \text{ and } proc(s_{i+1}) \neq proc(s_i) \} \quad (5)$$

For an illustration please refer to Fig. 13. In the next lemma we prove that if the sequence is admissible, in particular if it satisfies the agreement and continuity properties, then the above synchronization $\stackrel{S^{(2)}}{\to}$ does not interfere with $\stackrel{C^{(1)}}{\to}$. For convenience, we define $\stackrel{C^{(2)}}{\to}$ as the transitive closure of $\stackrel{C^{(1)}}{\to} \cup \stackrel{S^{(3)}}{\to}$. Formally,

$$\stackrel{C^{(2)}}{\to} \triangleq \left(\stackrel{C^{(1)}}{\to} \cup \stackrel{S^{(2)}}{\to}\right)^{+}$$

**Lemma 10** $\stackrel{C^{(2)}}{\to}$ *is an irreflexive partial order.*

Again, the proof can be found in the appendix. The final step is to prove that the combined synchronization, given by $\stackrel{S^{(1)}}{\to} \cup \stackrel{S^{(2)}}{\to}$, indeed ensures that the predicate is invariant in the resulting computation. Specifically, we show that if the sequence is admissible then every consistent cut of the resultant computation satisfies the antecedent of the weak safety property. We denote the controlled computation by $\langle E, \stackrel{C}{\to} \rangle$, where $\stackrel{C}{\to}$ is same as $\stackrel{C^{(2)}}{\to}$.

**Lemma 11** *Every consistent cut of* $\langle E, \stackrel{C}{\to} \rangle$ *satisfies* $b$.

*Proof.* Consider a consistent cut $C$ of $\langle E, \stackrel{C}{\to} \rangle$. We first prove that $C$ is legal with respect to $s$. Consider events $s_i$ and $s_j$. We have,

$$\begin{aligned}
& \{ \text{ assumption } \} \\
& (s_j \in C) \wedge (i < j) \\
\equiv\ & \{ \text{ definition of } \stackrel{S^{(1)}}{\to} \} \\
& (s_j \in C) \wedge (s_i \stackrel{S^{(1)}}{\to} s_j) \\
\Rightarrow\ & \{ \stackrel{S^{(1)}}{\to} \subseteq \stackrel{C}{\to} \} \\
& (s_j \in C) \wedge (s_i \stackrel{C}{\to} s_j) \\
\Rightarrow\ & \{ C \text{ is a consistent cut of } \langle E, \stackrel{C}{\to} \rangle \} \\
& s_i \in C
\end{aligned}$$

This establishes that $C$ is legal with respect to $s$. We now prove that the frontier of $C$ contains at least one event from $s$. To that end, we first prove that, for each $i$, $s_i \notin \top$ implies $s_{i+1} \stackrel{C}{\twoheadrightarrow} succ(s_i)$. Clearly, if $proc(s_{i+1}) \neq proc(s_i)$ then, by definition of $\stackrel{S^{(2)}}{\to}$, $s_{i+1} \stackrel{S^{(2)}}{\to} succ(s_i)$. Since $\stackrel{S^{(2)}}{\to} \subseteq \stackrel{C}{\to}$, $s_{i+1} \stackrel{C}{\to} succ(s_i)$. The more interesting case is when $proc(s_{i+1}) = proc(s_i)$. Since $proc(s_i) = proc(succ(s_i))$, $proc(s_{i+1}) = proc(succ(s_i))$. Then,

$$\begin{aligned}
& \{ \text{ events on a process are totally ordered by } \stackrel{P}{\to} \} \\
& (s_{i+1} \stackrel{P}{\twoheadrightarrow} succ(s_i)) \vee (succ(s_i) \stackrel{P}{\twoheadrightarrow} s_{i+1}) \\
\Rightarrow\ & \{ \stackrel{P}{\to} \subseteq \to \} \\
& (s_{i+1} \twoheadrightarrow succ(s_i)) \vee (succ(s_i) \twoheadrightarrow s_{i+1}) \\
\Rightarrow\ & \left\{ \begin{array}{l} \text{since } s \text{ satisfies the continuity property,} \\ succ(s_i) \not\twoheadrightarrow s_{i+1} \end{array} \right\} \\
& s_{i+1} \twoheadrightarrow succ(s_i) \\
\Rightarrow\ & \{ \to \subseteq \stackrel{C}{\to} \} \\
& s_{i+1} \stackrel{C}{\twoheadrightarrow} succ(s_i)
\end{aligned}$$

Assume, on the contrary, that the frontier of $C$ does not contain any event from $s$. We prove by induction on $i$ that, for each $i$, $s_i \in C$. Clearly, since $s$ satisfies the boundary condition and $\perp \subseteq C$, $s_1 \in C$. We have,

$$\begin{aligned}
& \{ \text{ induction hypothesis } \} \\
& s_i \in C \\
\equiv\ & \left\{ \begin{array}{l} \text{since } s_i \notin frontier(C),\ succ(s_i) \text{ exists} \\ \text{and it belongs to } C \end{array} \right\} \\
& succ(s_i) \in C
\end{aligned}$$

**Fig. 14.** An algorithm to compute a controlling synchronization for a disjunctive predicate (edges to initial events and from final events have been omitted)

$$\Rightarrow \{ s_{i+1} \stackrel{C}{\rightarrow} succ(s_i) \}$$
$$(s_{i+1} \stackrel{C}{\rightarrow} succ(s_i)) \wedge (succ(s_i) \in C)$$
$$\Rightarrow \{ C \text{ is a consistent cut of } \langle E, \stackrel{C}{\rightarrow} \rangle \}$$
$$s_{i+1} \in C$$

This establishes that $s_l \in C$. Since, since $s$ satisfies the boundary condition, $s_l \in \top$. Thus, trivially, $s_l \in frontier(C)$—a contradiction. This implies that the frontier of $C$ contains at least one event from $s$. Finally, since $s$ satisfies the weak safety property, $C$ satisfies $b$. □

Combining Lemma 9, Lemma 10 and Lemma 11, we obtain,

**Theorem 12 (sufficient condition)** *If there exists an admissible sequence with respect to a predicate b and a computation* $\langle E, \rightarrow \rangle$ *then b is controllable in* $\langle E, \rightarrow \rangle$.

Finally, from Theorem 8 and Theorem 12, it follows that,

**Theorem 13 (necessary and sufficient condition)** *It is possible to control a predicate b in a computation* $\langle E, \rightarrow \rangle$ *if and only if there exists an admissible sequence with respect to b and* $\langle E, \rightarrow \rangle$.

Although the motivation for defining the notion of admissible sequence was to devise a control strategy for a disjunctive predicate, nevertheless the preceding theorem holds for *any* global predicate.

### *5.2 Finding a controlling synchronization*

In this section, we derive an efficient algorithm for controlling a disjunctive predicate in a computation based on the notion of admissible sequence discussed in the previous section. The main idea is to construct a directed graph consisting of only the true events of the computation and then search for an "appropriate" path in the graph.

Note that, since false is a local predicate of any process, a disjunctive predicate $b$ can be written as disjunction of $n$ predicates such that the $i^{th}$ disjunct, denoted by $b^{(i)}$, is a local predicate of process $p_i$. The algorithm involves constructing a directed graph $G$, called the *true event graph*, as follows:

$$\mathsf{V}(G) \triangleq \{ e \mid e \models b^{(i)}, \text{ where } p_i = proc(e) \}$$
$$\mathsf{E}(G) \triangleq \{ (e, f) \mid e, f \in \mathsf{V}(G),\ e \neq f \text{ and } e \notin \top \Rightarrow succ(e) \not\rightarrow f \}$$

Here, $\mathsf{V}(G)$ and $\mathsf{E}(G)$ refer to the set of vertices and edges, respectively, of the graph $G$. We now define the notion of permissible path which is almost identical to the notion of admissible sequence except that a permissible path consists of true events only and may not satisfy the agreement property.

**Definition 7 (permissible path)** *A path in a true event graph (TEG) is* permissible *if it starts with an initial event and ends with a final event of the computation.*

Clearly, a permissible path satisfies the boundary condition as well as the continuity property. Furthermore, any consistent cut that contains a true event in its frontier, due to the semantics of disjunction, satisfies the predicate. Thus, a permissible path satisfies the weak safety property also. However, a permissible may not satisfy the agreement property in general. But if a path besides being permissible is also the shortest one then it satisfies the agreement property too. This is illustrated by the following example.

*Example 10* The true event graph for the computation shown in Fig. 14(a) and the disjunctive predicate $x_1 \vee x_2$ is depicted in Fig. 14(b). The path $e\,g\,h\,f\,u$ is permissible but does not satisfy the agreement property because although $f$ happened-before $g$ in the computation, it occurs after $g$ in the path. The path $e\,g\,u$ is a shortest permissible path. It can be verified that it indeed satisfies the agreement property. □

**Lemma 14** *A shortest permissible path in the true event graph, if it exists, satisfies the agreement property.*

*Proof.* Assume that the true event graph does contain a permissible path. Consider the shortest permissible path $s = s_1 s_2 \cdots s_l$. Assume, on the contrary, that $s$ does not satisfy the agreement property. Then,

$$\langle \exists\, i, j : i < j : s_j \rightarrow s_i \rangle$$
$$\Rightarrow \left\{ \begin{array}{l} s_j \notin \bot, \text{ otherwise } s_j s_{j+1} \cdots s_l \text{ is a shorter} \\ \text{permissible path than } s \end{array} \right\}$$
$$\langle \exists\, i, j : i < j : (s_j \rightarrow s_i) \wedge (s_j \notin \bot) \rangle$$
$$\Rightarrow \left\{ \begin{array}{l} i \geqslant 2, \text{ otherwise } s_i \in \bot \text{ implying } s_i \rightarrow s_j, \\ \text{thereby creating a cycle in } \rightarrow \end{array} \right\}$$
$$\langle \exists\, i, j : 2 \leqslant i < j : (s_j \rightarrow s_i) \wedge (s_j \notin \bot) \rangle$$

$$\Rightarrow \begin{Bmatrix} \text{since } s \text{ is a shortest permissible path,} \\ (s_{i-1}, s_j) \notin \mathsf{E}(G) \end{Bmatrix}$$
$$\langle \exists\, i, j : 2 \leqslant i < j : (succ(s_{i-1}) \rightarrow s_j) \wedge (s_j \rightarrow s_i) \rangle$$
$$\Rightarrow \{ \rightarrow \text{is transitive} \}$$
$$\langle \exists\, i : i \geqslant 2 : succ(s_{i-1}) \rightarrow s_i \rangle$$
$$\equiv \{ \text{definition of an edge} \}$$
$$\langle \exists\, i : i \geqslant 2 : (s_{i-1}, s_i) \notin \mathsf{E}(G) \rangle$$
$$\Rightarrow \begin{Bmatrix} s \text{ is a path implying} \\ \langle \forall\, i : i \geqslant 2 : (s_{i-1}, s_i) \in \mathsf{E}(G) \rangle \end{Bmatrix}$$
a contradiction

This establishes that $s$ satisfies the agreement property. □

The sufficient condition for a disjunctive predicate to be controllable in a computation can now be given as follows.

**Theorem 15 (sufficient condition)** *Given a disjunctive predicate $b$ and a computation $\langle E, \rightarrow \rangle$, if there exists a permissible path in the corresponding true event graph $G$ then $b$ is controllable in $\langle E, \rightarrow \rangle$.*

*Proof.* Assume that $G$ contains a permissible path. Clearly, each permissible path satisfies the boundary condition, the continuity property and the weak safety property. From Lemma 14, a shortest path among all permissible paths—not necessarily unique—also satisfies the agreement property. Thus a shortest permissible path in $G$ constitutes an admissible sequence with respect to $b$ and $\langle E, \rightarrow \rangle$. Using Theorem 13, $b$ is controllable in $\langle E, \rightarrow \rangle$. □

We next prove that the existence of a permissible path in the true event graph is also a necessary condition for a disjunctive predicate to be controllable in a computation.

**Theorem 16 (necessary condition)** *If a disjunctive predicate $b$ is controllable in a computation $\langle E, \rightarrow \rangle$ then there exists a permissible path in the corresponding true event graph $G$.*

*Proof.* Assume that $b$ is controllable in $\langle E, \rightarrow \rangle$. We inductively construct a path in the graph $G$ that is permissible. Since $b$ is controllable in $\langle E, \rightarrow \rangle$, there exists a total order $\rightsquigarrow$ that extends the partial order $\rightarrow$ such that $b$ is invariant in $\langle E, \rightsquigarrow \rangle$. The initial consistent cut of the computation $\langle E, \rightsquigarrow \rangle$, given by $\perp$, satisfies $b$. Thus there exists a true initial event. We call it $s_1$. Starting from $s_1$, we construct a path $s$ by adding events to the path constructed as yet until we reach a final event.

Let $s_i$ denote the last event added to the path so far. If $s_i$ is a final event then the path we have assembled so far is permissible. The more interesting case is when $s_i$ is not a final event. Consider the *least* consistent cut of $\langle E, \rightsquigarrow \rangle$ that contains $succ(s_i)$, say $C_i$. Note that $C_i$ is uniquely defined because the set of consistent cuts of a computation that contain a given event forms a lattice [8,13]. Since $b$ is invariant in $\langle E, \rightsquigarrow \rangle$, $C_i$ satisfies $b$. Thus the frontier of $C_i$ contains a true event. We call it $s_{i+1}$. We still have to show that there is an edge from $s_i$ to $s_{i+1}$ in the graph $G$, that is, $succ(s_i) \not\rightarrow s_{i+1}$. By definition of $C_i$, for each $e \in C_i$, $e \underline{\rightsquigarrow} succ(s_i)$. Since $s_{i+1} \in C_i$, $s_{i+1} \underline{\rightsquigarrow} succ(s_i)$. Since $\rightsquigarrow$ is an irreflexive partial order, $succ(s_i) \not\rightsquigarrow s_{i+1}$. Thus $succ(s_i) \not\rightarrow s_{i+1}$ because $\rightarrow \subseteq \rightsquigarrow$.

Finally, we prove that a final event is eventually added to the path. Assume that $s_{i+1} \notin \top$. Since $s_{i+1} \in frontier(C)$, $succ(s_{i+1}) \notin C_i$. By definition of $C_i$, $succ(s_{i+1}) \not\underline{\rightsquigarrow} succ(s_i)$. Since $\rightsquigarrow$ is a total order, $succ(s_i) \rightsquigarrow succ(s_{i+1})$. This implies that $C_i \subsetneq C_{i+1}$, that is, $s_{i+1}$ is different from every event already in the path. Thus no event is added to the path being built more than once, thereby establishing that a final event is eventually added to the path. □

From Theorem 15 and Theorem 16, it follows that,

**Theorem 17 (necessary and sufficient condition)** *A disjunctive predicate $b$ is controllable in a computation $\langle E, \rightarrow \rangle$ if and only if there exists a permissible path in the corresponding true event graph $G$.*

The true event graph has $O(|E|)$ vertices and $O(|E|^2)$ edges. A shortest permissible path in the graph can be determined using breadth first search in $O(|E|^2)$ time. Thus the algorithm has an overall time-complexity of $O(|E|^2)$. To improve the time-complexity, we attempt to reduce the number of edges in the graph. To that end, the following observation proves to be helpful.

**Observation 3** *If there is an edge from a true event $e$ to a true event $f$ then there is an edge from every true event that occurs after $e$ on $proc(e)$ to every true event that occurs before $f$ on $proc(f)$. Formally,*

$$(e, f) \in \mathsf{E}(G)$$
$$\Rightarrow$$
$$\langle \forall\, g, h \in \mathsf{V}(G) : (e \overset{P}{\rightarrow} g) \wedge (h \overset{P}{\rightarrow} f) : (g, h) \in \mathsf{E}(G) \rangle$$

It can be verified that, given a true event $e$ and a process $p$, if we only put an edge from $e$ to the *last* true event $f$ on $p$ such that $succ(e) \not\rightarrow f$, in case $succ(e)$ exists, then Theorem 17 still holds. In particular, it can be proved that existence of a permissible path of length $l$ in the true event graph implies existence of a permissible path in the "reduced" true event graph (RTEG) of length at most $l$. The reduced true event graph has at most $O(n|E|)$ edges, thereby reducing the time-complexity to $O(n|E|)$. It can be verified that the message-complexity of the algorithm is $O(|E|)$.

In order to reduce the time-complexity further, we define the notion of *true-interval*—a maximal contiguous sequence of true event on a process. Rather than find a sequence of true event that satisfy certain properties, we can find a sequence of true-intervals satisfying "similar" properties. The details are left to the reader. This algorithm for computing a controlling synchronization for a disjunctive predicate—based on true-intervals—has time-complexity of $O(n|T| + |E|)$ and message-complexity of $O(|T|)$, where $T$ is the set of true-intervals of the computation, which is same as that of Tarafdar and Garg's algorithm [20].

### *5.3 Finding a minimum controlling synchronization*

Now, we modify the above algorithm to compute a *minimum controlling synchronization*, that is, a synchronization with the

least number of dependencies that are not subsumed by the happened-before relation. Such a synchronization is optimal in terms of the number of control messages required to realize it, and has applications when the channel bandwidth is somewhat limited. The main idea is to look for a *specific* shortest permissible path in the true event graph instead of *any* shortest permissible path. This is achieved by assigning a weight to each edge and finding a shortest weighted permissible path. Observe that unlike the smallest controlling synchronization which is unique a minimum controlling synchronization is not.

To find a minimum controlling synchronization, we take advantage of the fact that the predicate to be controlled is disjunctive. As a result, a sequence of true events satisfies a stronger property than the weak safety property: "a consistent cut that contains at least one event from the sequence in its frontier satisfies the predicate". In particular, the cut is not required to be legal. Therefore the following holds:

**Observation 4** *Let $s$ be an admissible sequence with respect to $b$ and $\langle E, \rightarrow\rangle$. If $b$ is a disjunctive predicate then the synchronization given by $\overset{s^{(2)}}{\rightarrow}$ defined in (5) in Sect. 5.1 is sufficient to control $b$ in $\langle E, \rightarrow\rangle$.*

Although the synchronization dependencies given by $\overset{s^{(1)}}{\rightarrow}$ can be omitted, the sequence is still required to satisfy the agreement property. This is to ensure that the synchronization $\overset{s^{(2)}}{\rightarrow}$ does not interfere with the happened-before relation of the computation. To count the number of synchronization dependencies in $\overset{s^{(2)}}{\rightarrow}$ that are not covered by $\rightarrow$, we assign weight to each edge as follows:

$$w(e,f) \triangleq \begin{cases} (0,1) & : \text{ if } f \underline{\rightarrow} succ(e) \\ (1,1) & : \text{ otherwise} \end{cases}$$

Two weights are added by summing their respective components and are compared using lexicographic comparison. As before in the case of true event graph, a shortest permissible path in the weighted true event graph not only satisfies the boundary condition, the continuity property and the weak safety property but also satisfies the agreement property.

**Lemma 18** *A shortest permissible path in the weighted true event graph, if it exists, satisfies the agreement property.*

The proof is similar to the proof of Lemma 14 and can be found in the appendix. For a path $s$ with weight $w(s)$, let $w_f(s)$ denote the first entry of the tuple $w(s)$. The rank of a weighted true event graph $G$, denoted by $rank(G)$, is given by,

$$rank(G) \triangleq \begin{cases} \bot & : \text{if there is no permissible path in G} \\ w_f(s) & : s \text{ is a shortest weighted} \\ & \quad \text{permissible path in } G \end{cases}$$

Intuitively, the rank gives the cardinality of minimum controlling synchronization. We now show that rank behaves in a "continuous" fashion by proving that adding a single synchronization dependency to a computation cannot reduce the rank of its weighted true event graph substantially. Consider a computation $\langle E, \rightsquigarrow\rangle$ such that (1) $\rightsquigarrow$ extends $\rightarrow$, and (2) the two computations $\langle E, \rightarrow\rangle$ and $\langle E, \rightsquigarrow\rangle$ differ by at most one message. Formally,

$$\langle \exists\, e, f :: \rightsquigarrow = (\rightarrow \cup (e,f)^{+})\rangle$$

Let $H$ be the weighted true event graph corresponding to $b$ and $\langle E, \rightsquigarrow\rangle$.

**Lemma 19 (bounded reduction)** *If $b$ is controllable in $\langle E, \rightsquigarrow\rangle$ then $rank(G)$ is at most one more than $rank(H)$.*

*Proof.* Since $\langle E, \rightsquigarrow\rangle \models controllable: b$, by virtue of Theorem 16, there exists a permissible path in $H$. Consider a shortest permissible path in $H$, say $s = s_1 s_2 \cdots s_l$. For convenience, let $w^G$ and $w^H$ be the weight functions for the graphs $G$ and $H$, respectively. Since $\rightarrow \subseteq \rightsquigarrow$, $succ(e) \not\rightsquigarrow f$ implies $succ(e) \not\rightarrow f$. Thus each edge of $H$ is also an edge of $G$ which implies that $s$ is a path in $G$. The following can be easily verified.

$$rank(G) \leqslant w_f^G(s) \tag{6}$$

$$rank(H) = w_f^H(s) \tag{7}$$

$$\langle \forall\, e, f : (e,f) \in \mathsf{E}(H) : w^G(e,f) = (0,1)\rangle \Rightarrow w^H(e,f) = (0,1) \tag{8}$$

We first prove that $w_f^G(s) - w_f^H(s) \leqslant 1$. Assume the contrary. Thus, from (8), there exist at least two distinct edges in the path $s$ such that their weight in $G$ is $(1,1)$ but in $H$ is $(0,1)$. Let the edges be $(s_i, s_{i+1})$ and $(s_j, s_{j+1})$, where $i \neq j$. Equivalently,

$$s_{i+1} \not\underline{\rightarrow}\, succ(s_i) \text{ and } s_{j+1} \not\underline{\rightarrow}\, succ(s_j) \tag{9}$$

$$s_{i+1} \underline{\rightsquigarrow}\, succ(s_i) \text{ and } s_{j+1} \underline{\rightsquigarrow}\, succ(s_j) \tag{10}$$

Let the additional message in $\langle E, \rightsquigarrow\rangle$ be from $e$ to $f$. From (9) and (10), we can deduce that there exists a path from $s_{i+1}$ to $succ(s_i)$ in $\langle E, \rightsquigarrow\rangle$ that involves the message from $e$ to $f$. Likewise, there exists a path from $s_{j+1}$ to $succ(s_j)$ in $\langle E, \rightsquigarrow\rangle$ that involves the message from $e$ to $f$. Then,

$$s_{i+1} \underline{\rightsquigarrow}\, e \text{ and } f \underline{\rightsquigarrow}\, succ(s_i) \tag{11}$$

$$s_{j+1} \underline{\rightsquigarrow}\, e \text{ and } f \underline{\rightsquigarrow}\, succ(s_j) \tag{12}$$

Without loss of generality, assume that $i < j$. Two possible cases arise depending on whether there is an edge from $s_i$ to $s_{j+1}$ in $H$. We have,

Case 1: $(s_i, s_{j+1}) \notin \mathsf{E}(H)$

$\{$ definition of an edge $\}$

$succ(s_i) \rightsquigarrow s_{j+1}$

$\Rightarrow$ $\{$ using (12) $\}$

$succ(s_i) \rightsquigarrow e$

$\Rightarrow$ $\{$ using (11) $\}$

$f \rightsquigarrow e$

$\Rightarrow$ $\{$ definition of $\rightsquigarrow$ implies $e \rightsquigarrow f$ $\}$

a contradiction

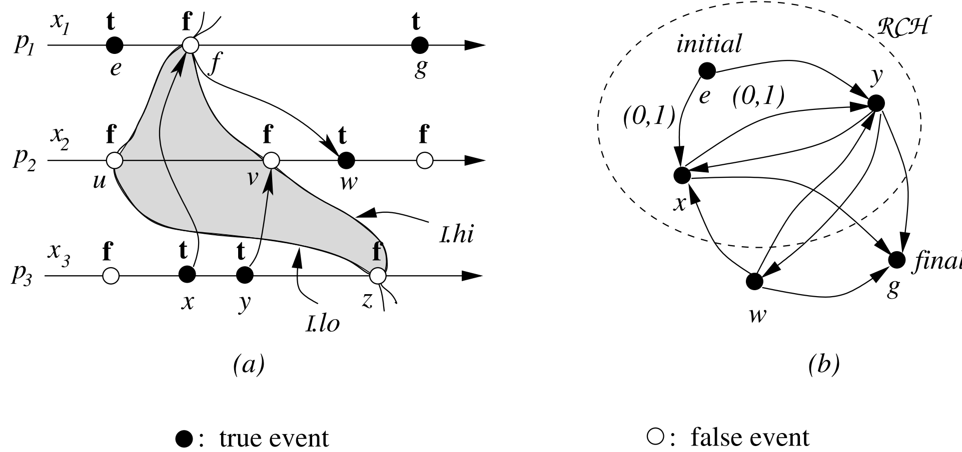


**Fig. 15.** An example to illustrate $\mathcal{I}$

In the second case, when there is an edge from $s_i$ to $s_{j+1}$, from (11) and (12), $s_{j+1} \rightsquigarrow succ(s_i)$. Thus $w^H(s_i, s_{j+1}) = (0,1)$ implying that the path $s_1 s_2 \cdots s_i s_{j+1} \cdots s_l$ is permissible in $H$ and has smaller weight than $s$—a contradiction. Thus,

$$w_f^G(s) - w_f^H(s) \leqslant 1 \tag{13}$$

Finally,

$$\begin{aligned}
& \{ \text{ using (6) } \} \\
& rank(G) \leqslant w_f^G(s) \\
\equiv\ & \{ \text{ using (13) } \} \\
& rank(G) \leqslant w_f^H(s) + 1 \\
\equiv\ & \{ \text{ using (7) } \} \\
& rank(G) \leqslant rank(H) + 1
\end{aligned}$$

This establishes the lemma. □

Now, assume that $rank(G) \neq 0$. Let $\mathcal{RCH}$ denote the subset of true events that are reachable from some initial true event in the weighted true event graph $G$ via edges with weight $(0,1)$ only. Since $rank(G) \neq 0$, $\mathcal{RCH}$ does not contain any final event; if otherwise, there is a path from an initial event to a final event via edges with weight $(0,1)$ only, thereby forcing $rank(G)$ to be zero. For each process $p_i$, we identify an interval of contiguous events on $p_i$ that we denote by $\mathcal{I}_i$. The first event of $\mathcal{I}_i$, denoted by $\mathcal{I}_i.lo$, is given by the successor of the last event on $p_i$ that belongs to $\mathcal{RCH}$. In case there is no such event, $\mathcal{I}_i.lo$ is set to $\bot_i$, the initial event on $p_i$. The last event of $\mathcal{I}_i$, denoted by $\mathcal{I}_i.hi$, is given by the earliest event on $p_i$ that did not occur before $\mathcal{I}_i.lo$ such that its successor, if it exists, is a true event. Clearly, $\mathcal{I}_i$ is non-empty and all events in $\mathcal{I}_i$ are false events. For convenience,

$$\begin{aligned}
\mathcal{I} &\triangleq \bigcup_{1 \leqslant i \leqslant n} \mathcal{I}_i \\
\mathcal{I}.lo &\triangleq \{\, \mathcal{I}_i.lo \mid 1 \leqslant i \leqslant n \,\} \\
\mathcal{I}.hi &\triangleq \{\, \mathcal{I}_i.hi \mid 1 \leqslant i \leqslant n \,\} \\
succ(\mathcal{I}.hi) &\triangleq \{\, succ(e) \mid e \in \mathcal{I}.hi \text{ and } e \notin \top \,\}
\end{aligned}$$

The following example illustrates the aforementioned concepts.

*Example 11* Consider the computation portrayed in Fig. 15(a) and the disjunctive predicate $x_1 \vee x_2 \vee x_3$. The corresponding weighted true event graph is depicted in Fig. 15(b). The incoming edges to the initial event $e$ and the outgoing edges from the final event $g$ have been omitted for obvious reasons. All edges except the edges $(e,x)$ and $(x,y)$ have weight $(1,1)$. For clarity, we have only labeled those edges that have weight $(0,1)$ because they are fewer in number. Thus the set $\mathcal{RCH}$ is given by $\{e, x, y\}$. Further, $\mathcal{I}_1.lo = succ(e) = f$, $\mathcal{I}_2.lo = \bot_2 = u$ and $\mathcal{I}_3.lo = succ(y) = z$. Also, $\mathcal{I}_1.hi = f$, $\mathcal{I}_2.hi = v$ and $\mathcal{I}_3.hi = \top_3 = z$. Finally, $succ(\mathcal{I}) = \{succ(f), succ(v)\} = \{g, w\}$. The shaded region in Fig. 15(a) corresponds to the space spanned by the events of $\mathcal{I}$. □

Observe that if all events in the frontier of a consistent cut belong to $\mathcal{I}$ then the cut will not satisfy the given disjunctive predicate. We make two observations about the set $succ(\mathcal{I}.hi)$. First, all events in the set are true events. Second, no event in the set belongs to $\mathcal{RCH}$. The following lemma proves that the computation must contain a consistent cut that does not satisfy the disjunctive predicate.

**Lemma 20** *If the rank of a weighted true event graph is not zero then there exists a consistent cut of the computation that does not satisfy the disjunctive predicate.*

*Proof.* Our approach is to add enough synchronization dependencies to the computation $\langle E, \rightarrow \rangle$, without creating any deadlock (or cycle), to obtain another computation, say $\langle E, \rightsquigarrow \rangle$, that satisfies the required property. Specifically, we show that the computation $\langle E, \rightsquigarrow \rangle$ contains a consistent cut whose frontier is completely contained in $\mathcal{I}$. Since all events in $\mathcal{I}$ are false events, we obtain the desired result. The required set of dependencies, denoted by $\stackrel{\mathcal{I}}{\rightarrow}$, is given by,

$$\stackrel{\mathcal{I}}{\rightarrow} \;\triangleq\; \{\, (e,f) \mid e \in \mathcal{I}.lo \text{ and } f \in succ(\mathcal{I}.hi) \,\}$$

We first prove that adding dependencies from $\stackrel{\mathcal{I}}{\rightarrow}$ to $\rightarrow$ does not create any cycle. Consider a path $e \stackrel{\mathcal{I}}{\rightarrow} f \underline{\rightarrow} g \stackrel{\mathcal{I}}{\rightarrow} h$ (events $e$, $f$, $g$ and $h$ need not all be distinct, that is, an event or a sequence of events may be repeated in the path). By definition of $\stackrel{\mathcal{I}}{\rightarrow}$, $f \in succ(\mathcal{I}.hi)$ and $g \in \mathcal{I}.lo$. Clearly, $f \notin \bot$. This implies that $g \notin \bot$; if otherwise, $g \rightarrow f$, thereby creating a cycle in $\rightarrow$. Thus $pred(g)$ exists. Furthermore, both $f$ and $pred(g)$ are true events such that $pred(g) \in \mathcal{RCH}$ but $f \notin \mathcal{RCH}$. Note,

however, that $f \underline{\rightarrow} succ(pred(g))(= g)$ implying that there is an edge from $pred(g)$ to $f$ with weight $(0, 1)$. Thus $f$ is reachable from an initial event via edges with weight $(0, 1)$ only because $pred(g) \in \mathcal{RCH}$ and $w(pred(g), f) = (0, 1)$. This implies that $f$ belongs to $\mathcal{RCH}$—a contradiction. Thus there is no path in $\rightarrow \cup \stackrel{\mathcal{I}}{\rightarrow}$ of the form $e \stackrel{\mathcal{I}}{\rightarrow} f \underline{\rightarrow} g \stackrel{\mathcal{I}}{\rightarrow} h$, thereby ensuring that $\rightarrow \cup \stackrel{\mathcal{I}}{\rightarrow}$ is acyclic.

Now, $\rightsquigarrow = (\rightarrow \cup \stackrel{\mathcal{I}}{\rightarrow})^{+}$. Consider the *least* consistent cut of $\langle E, \rightsquigarrow \rangle$, say $C_{least}((\mathcal{I}.lo))$, that contains $\mathcal{I}.lo$. By definition of $C_{least}((\mathcal{I}.lo))$, we have,

$$\langle \forall\, e :: e \in C_{least}((\mathcal{I}.lo)) \;\Rightarrow\; \langle \exists\, f : f \in \mathcal{I}.lo : e \underline{\rightsquigarrow} f \rangle \rangle \quad (14)$$

We prove that the frontier of $C_{least}((\mathcal{I}.lo))$ lies wholly within $\mathcal{I}$. To that end, it suffices to show that $C_{least}((\mathcal{I}.lo))$ does not contain any event from $succ(\mathcal{I}.hi)$. Assume the contrary. Then,

$$\begin{aligned}
& \langle \exists\, e : e \in succ(\mathcal{I}.hi) : e \in C_{least}((\mathcal{I}.lo)) \rangle \\
\Rightarrow\ & \{ \text{ using (14) } \} \\
& \langle \exists\, e, f : (e \in succ(\mathcal{I}.hi)) \wedge (f \in \mathcal{I}.lo) : e \underline{\rightsquigarrow} f \rangle \\
\Rightarrow\ & \{ \text{ by definition of } \stackrel{\mathcal{I}}{\rightarrow}, f \stackrel{\mathcal{I}}{\rightarrow} e \text{ and } \stackrel{\mathcal{I}}{\rightarrow} \subseteq \rightsquigarrow \} \\
& \langle \exists\, e, f : (e \in succ(\mathcal{I}.hi)) \wedge (f \in \mathcal{I}.lo) : \\
& \qquad (e \underline{\rightsquigarrow} f) \wedge (f \rightsquigarrow e) \rangle \\
\Rightarrow\ & \{ \rightsquigarrow \text{ is an irreflexive partial order } \} \\
& \text{a contradiction}
\end{aligned}$$

This establishes the lemma. □

The necessary and sufficient condition for the rank of a weighted true event graph to be zero can now be furnished easily.

**Theorem 21** *The rank of a weighted true event graph is zero if and only if the disjunctive predicate is invariant in the computation. Formally,*

$$\langle E, \rightarrow \rangle \models invariant : b \iff rank(G) = 0$$

*Proof.* We use the ping-pong argument to prove the theorem.

($\Rightarrow$) Follows from Lemma 20.

($\Leftarrow$) From Lemma 18, a shortest permissible path, say $s$—which exists because $rank(G) \neq \bot$—corresponds to an admissible sequence of events with respect to $b$ and $\langle E, \rightarrow \rangle$. Since $b$ is a disjunctive predicate, by Observation 4, $\stackrel{S^{(2)}}{\rightarrow}$ is sufficient to control $b$ in $\langle E, \rightarrow \rangle$. Let $\stackrel{C}{\rightarrow} = (\rightarrow \cup \stackrel{S^{(2)}}{\rightarrow})^{+}$. By definition of controllability, $b$ is invariant in $\langle E, \stackrel{C}{\rightarrow} \rangle$. Furthermore, by definition of the weight function, $\stackrel{S^{(2)}}{\rightarrow} \subseteq \rightarrow$ which implies that $\stackrel{C}{\rightarrow} = \rightarrow$. □

We now present the main result of this section.

**Theorem 22 (minimum controlling synchronization)** *A shortest permissible path in the weighted true event graph, if it exists, corresponds to a minimum controlling synchronization for the disjunctive predicate in the given computation.*

The proof is straightforward and can be found in the appendix. The algorithm to compute a minimum controlling synchronization has $O(|E|^2)$ time-complexity because the weighted true event graph has $O(|E|)$ vertices, $O(|E|^2)$ edges, and a shortest permissible path in the graph can be determined using Dijkstra's shortest path algorithm [4] in $O(|E|^2)$ time.

## 6 Controlling general predicates

In the previous sections, we present efficient algorithms to find controlling synchronizations for region predicates and disjunctive predicates. For the former, we give an algorithm to generate the optimal controlling synchronization. For the latter, we give an algorithm to generate a minimum controlling synchronization. In this section, we provide a necessary and sufficient condition under which it is possible to efficiently compute a *minimal* controlling synchronization for a *general* predicate, and also give an algorithm to compute such a synchronization. First, we show that if $controllable : b$ can be evaluated efficiently (to "yes" or "no") then there is an efficient algorithm to compute a controlling synchronization for $b$ and vice versa.

**Theorem 23** *There exists a polynomial-time algorithm for computing a controlling synchronization for a predicate b, if it exists, if and only if there exists a polynomial-time algorithm for evaluating $controllable : b$.*

*Proof.* We use the ping-pong argument to prove the theorem.

*(if)* Suppose there is a polynomial-time algorithm for evaluating $controllable : b$. Evidently, if $controllable : b$ is false then no controlling synchronization exists for $b$. However, if $controllable : b$ is true, then a controlling synchronization for $b$ can be computed as follows. Add synchronization arrows to the computation repeatedly until there are no pair of concurrent events left (that is, all events are totally ordered). Of course, at each step, an arrow is added in such a manner that $b$ remains controllable in the resulting computation. The algorithm is described in Fig. 16. The correctness of the algorithm follows from the following observations. Suppose $\langle E, \rightarrow \rangle$ is a computation such that $b$ is controllable in $\langle E, \rightarrow \rangle$. For a pair of concurrent events $(e, f)$, let $\stackrel{(e,f)}{\rightarrow}$ denote the irreflexive partial order of the computation obtained by adding a synchronization arrow from $e$ to $f$. Then, $b$ is controllable in either $\langle E, \stackrel{(e,f)}{\rightarrow} \rangle$ or $\langle E, \stackrel{(f,e)}{\rightarrow} \rangle$ (lines 5-10). Formally,

$$\begin{aligned}
& \langle E, \rightarrow \rangle \models controllable : b \\
& \qquad \equiv \\
& (\langle E, \stackrel{(e,f)}{\rightarrow} \rangle \models controllable : b) \vee \\
& \quad (\langle E, \stackrel{(f,e)}{\rightarrow} \rangle \models controllable : b)
\end{aligned}$$

Also, if $\langle E, \rightarrow \rangle$ does not contain any pair of concurrent events—terminating condition for the while loop—then $b$ is invariant in $\langle E, \rightarrow \rangle$. Formally,

$$\langle E, \rightarrow \rangle \models controllable : b \;\equiv\; \langle E, \rightarrow \rangle \models invariant : b$$

```
    Input: (1) a computation ⟨E, →⟩, (2) a predicate b, and
           (3) an efficient algorithm to evaluate controllable: b

    Output: a controlling synchronization for b, if it exists

1   if ⟨E, →⟩ ⊭ controllable: b then
2       exit("no controlling synchronization exists for b");
    else
3       →S := ∅;
4       →C := →;
5       while there exist events e and f such that e ∥ f in →C do
6           ⇝ := (→C ∪ {(e, f)})+;
7           if ⟨E, ⇝⟩ ⊨ controllable: b then
                // add a synchronization arrow from e to f
8               →S := →S ∪ {(e, f)};
            else
                // add a synchronization arrow from f to e
9               →S := →S ∪ {(f, e)};
            endif;
            // add the transitive dependencies and compute the resulting computation
10          →C := (→ ∪ →S)+;
        endwhile;
11      exit(→S);
    endif;
```

**Fig. 16.** The algorithm FindContSync to compute a controlling synchronization

This implies that, when the while loop terminates, $b$ is invariant in the resultant computation.

*(only if)* Suppose there is a polynomial-time algorithm for finding a controlling synchronization for $b$, whenever it exists. Clearly, $controllable: b$ is false if the algorithm reports that no controlling synchronization exists for $b$. On the other hand, if the algorithm manages to find at least one controlling synchronization, then $controllable: b$ is true. □

The algorithm FindContSync produces a controlling synchronization that is too restrictive in the sense that it inhibits any concurrency whatsoever in the controlled computation whereas we want to retain as much concurrency as possible. To determine a controlling synchronization that permits greater concurrency in the controlled computation, it turns out that we should be able to evaluate a predicate not only under $controllable$ modality efficiently but also under $invariant$ modality efficiently. To that end, we first define the notion of *minimal controlling synchronization.*

**Definition 8 (minimal controlling synchronization)** *A controlling synchronization is said to be* minimal *if it is not possible to remove any synchronization dependency from the corresponding controlled computation while still maintaining the predicate (as invariant). Formally, given a controlling synchronization* $\overset{S}{\rightarrow}$ *for a predicate* $b$ *in a computation* $\langle E, \rightarrow \rangle$,

$$\overset{S}{\rightarrow} \text{ is minimal} \triangleq \langle \forall \leadsto : \rightarrow \subseteq \leadsto \subseteq \overset{C}{\rightarrow} : \langle E, \leadsto \rangle \models invariant: b \equiv \leadsto = \overset{C}{\rightarrow} \rangle$$

*where* $\overset{C}{\rightarrow} = (\rightarrow \cup \overset{S}{\rightarrow})^{+}$.

Observe that minimal controlling synchronization exists for a predicate whenever the predicate is controllable, but it may not be uniquely defined. Now, we show that if both $controllable: b$ and $invariant: b$ can be evaluated efficiently (to "yes" or "no") then there is an efficient algorithm to compute a minimal controlling synchronization for $b$ and vice versa.

**Theorem 24** *There exists a polynomial-time algorithm for computing a minimal controlling synchronization for a predicate* $b$*, if it exists, if and only if there exist polynomial-time algorithms for evaluating* $controllable: b$ *and* $invariant: b$.

*Proof.* We use the ping-pong argument to establish the theorem.

*(if)* Suppose there are polynomial-time algorithms for evaluating $b$ under both $controllable$ and $invariant$ modalities. Clearly, if $controllable: b$ is false then no controlling synchronization exists for $b$. On the other hand, if $controllable: b$ holds then a minimal controlling synchronization can be determined as follows. First, determine a controlling synchronization for $b$ using the algorithm FindContSync in Fig. 16 and compute the corresponding controlled computation. The resultant computation may contain unnecessary synchronization dependencies and therefore may be too restrictive. Next, repeatedly remove synchronization dependencies from the controlled computation in such a way that $b$ remains invariant in the resulting computation. To test for the invariance of $b$, the algorithm for evaluating $invariant: b$ efficiently can be used. The algorithm is described in detail in Fig. 17.

*(only if)* Suppose there is a polynomial-time algorithm for finding a minimal controlling synchronization for $b$, when-

```
Input: (1) a computation ⟨E, →⟩, (2) a predicate b, and
       (3) efficient algorithms to evaluate controllable: b and invariant: b

Output: a minimal controlling synchronization for b, if it exists

if ⟨E, →⟩ ⊭ controllable: b then
    exit("no controlling synchronization exists for b");
else
    →S := compute a controlling synchronization for b;
    →C := (→ ∪ →S)+;
    done := false;
    // remove unnecessary synchronization arrows from →C
    while not(done) do
        found := false;
        // test whether it is possible to remove a synchronization arrow from →C while
        // guaranteeing that b remains invariant in the resulting computation
        for each pair of events (e, f) such that e ∥ f in → but e →C f do
            // remove the synchronization arrow from e to f
            ⇝ := (→C \ {(e, f)})+;
            if (⇝ ≠ →C) and                       // is the resulting computation different?
               ⟨E, ⇝⟩ ⊨ invariant: b then          // is b is still invariant?
                →C := ⇝;                          // remove the synchronization arrow
                found := true;
                break;                            // quit the for loop
            endif;
        endfor;
        if not(found) then                        // it is not possible to remove any
            done := true;                         // synchronization arrow
        endif;
    endwhile;
    →S := →C \ →;
    exit(→S);
endif;
```

**Fig. 17.** The algorithm FindMinlContSync to compute a minimal controlling synchronization

ever it exists. Clearly, $controllable: b$ holds if the algorithm is able to find a minimal controlling synchronization and vice versa. Moreover, $invariant: b$ holds if and only if the synchronization produced by the algorithm is empty. □

For a linear predicate $b$, Sen and Garg [16] give $O(n^2|E|)$ algorithms for evaluating $controllable: b$ and $invariant: b$ in a computation, where $n$ is the number of processes and $E$ is the set of events. Using Theorem 24, it is, therefore, possible to efficiently compute a minimal controlling synchronization for a linear predicate.

## 7 Conclusion and future work

A distributed debugger equipped with the mechanism to re-execute a traced computation under control, with added synchronization, can greatly facilitate the detection and localization of bugs. For software-fault tolerance, in the case of synchronization faults, instead of relying on chance controlled re-execution can be used to avoid a fault in a deterministic manner. In this paper, we provide control algorithms for two useful classes of predicates, namely region predicates and disjunctive predicates. For the former, we demonstrate that the control algorithm is *optimal* in the sense that it guarantees maximum concurrency possible in the controlled computation. For the latter, we give a control algorithm with optimal message-complexity that generates the *least* number of synchronization dependencies. Also, for a general predicate satisfying certain condition, we provide an efficient algorithm to compute a *minimal* controlling synchronization.

It is possible to generalize the notion of admissible sequence of events to the notion of admissible sequence of *sub-frontiers*; a sub-frontier is a subset of mutually consistent events or, in other words, there is at least one consistent cut that passes through all the events in the sub-frontier. An interesting question is: *"Can this generalized notion be used to derive an efficient control algorithm for the class of* $k$-local disjunctive predicates *with* $k > 1$*?"* A $k$-local disjunctive predicate is a disjunction of $k$-local predicates, where a $k$-local predicate depends on variables of at most $k$ processes.

The control algorithms presented is this paper are centralized in nature. They assume that every process sends information about its events, as they are generated, to a central daemon. The daemon then collects the information from all processes, builds the trace and, when needed, computes the synchronization. We are currently working on developing distributed control algorithms for region predicates and disjunctive predicates.

## References

1. Chandy KM, Lamport L: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems 3(1):63–75 (1985)
2. Chase C, Garg VK: Detection of Global Predicates: Techniques and their Limitations. Distributed Computing (DC) 11(4):191–201 (1998)
3. Cooper R, Marzullo K: Consistent Detection of Global Predicates. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp 163–173, Santa Cruz, California, 1991
4. Cormen TH, Leiserson CE, Rivest RL: *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1991
5. Fidge C: Logical Time in Distributed Computing Systems. IEEE Computer 24(8):28–33 (1991)
6. Huang Y, Kintala C: Software Implemented Fault Tolerance: Technologies and Experience. In: Proceedings of the IEEE Fault-Tolerant Computing Symposium (FTCS), pp 138–144, June 1993
7. Hurfin M, Mizuno M, Raynal M, Singhal M: Efficient Distributed Detection of Conjunctions of Local Predicates in Asynchronous Computations. In: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP), pp 588–594, New Orleans, October 1996
8. Johnson DB, Zwaenepoel W: Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In: Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC), pp 171–181, August 1988
9. Kilgore R, Chase C: Testing Distributed Programs Containing Racing Messages. The Computer Journal 40(8):489–498 (1997)
10. Lamport L: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (CACM) 21(7):558–565 (1978)
11. LeBlanc TJ, Mellor-Crummey JM: Debugging Programs with Instant Replay. IEEE Transactions on Computers C-36(4):471–482 (1987)
12. Maggiolo-Schettini A, Welde H, Winkowski J: Modeling a Solution for a Control Problem in Distributed Systems by Restrictions. Theoretical Computer Science 13(1):61–83 (1981)
13. Mattern F: Virtual Time and Global States of Distributed Systems. In: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG), pp 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989
14. Miller BP, Choi J: Breakpoints and Halting in Distributed Programs. In: Proceedings of the 8th IEEE International Conference on Distributed Computing Systems (ICDCS), pp 316–323, 1988
15. Mittal N, Garg VK: Debugging Distributed Programs Using Controlled Re-execution. In: Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC), pp 239–248, Portland, Oregon, July 2000
16. Sen A, Garg VK: Detecting Temporal Logic Predicates in the Happened-Before Model. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Florida, April 2002
17. Stoller SD, Liu YA: Efficient Symbolic Detection of Global Properties in Distributed Systems. In Hu AJ, Vardi MY (eds) Proceedings of the 10th International Conference on Computer-Aided Verification (CAV), volume 1427 of Lecture Notes in Computer Science (LNCS), pp 357–368. Springer-Verlag, 1998
18. Stoller SD, Unnikrishnan L, Liu YA: Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In: Proceedings of the 12th International Conference on Computer-Aided Verification (CAV), volume 1855 of Lecture Notes in Computer Science (LNCS), pp 264–279. Springer-Verlag, July 2000
19. Tarafdar A: Software Fault Tolerance in Distributed Systems Using Controlled Re-execution. PhD thesis, The University of Texas at Austin, August 2000
20. Tarafdar A, Garg VK: Predicate Control for Active Debugging of Distributed Programs. In: Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP), pp 763–769, Orlando, 1998
21. Tarafdar A, Garg VK: Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In: Proceedings of the 13th Symposium on Distributed Computing (DISC), pp 210–224, Bratislava, Slovak Republic, September 1999
22. Torres-Pomales W: Software Fault Tolerance: A Tutorial, 2000. NASA Langley Research Center
23. Wang Y-M, Huang Y, Fuchs WK, Kintala C, Suri G: Progressive Retry for Software Failure Recovery in Message-Passing Applications. IEEE Transactions on Computers 46(10):1137–1141 (1997)

## A Omitted proofs

*Proof for Theorem 1.* We have to prove that if $b_1$ and $b_2$ are $p$-region predicates then so is $b_1 \wedge b_2$. We first prove that $b_1 \wedge b_2$ satisfies the weak lattice property. Consider consistent cuts $C_1$ and $C_2$ passing through an event $e$ on process $p$ that satisfy $b_1 \wedge b_2$. By semantics of conjunction, both $C_1$ and $C_2$ satisfy $b_1$ as well as $b_2$. Applying the weak lattice property twice, we obtain $C_1 \cap C_2$ satisfies $b_1$ and $b_2$. Again, by semantics of conjunction, $C_1 \cap C_2$ satisfies $b_1 \wedge b_2$. Likewise, $C_1 \cup C_2$ satisfies $b_1 \wedge b_2$. Thus $b_1 \wedge b_2$ satisfies the weak lattice property.

We now prove that $b_1 \wedge b_2$ satisfies the weak convexity property. Consider consistent cuts $C_1$ and $C_2$ passing through $e$ that satisfy $b_1 \wedge b_2$ and let $C$ be any consistent cut that lies between the two. By semantics of conjunction, both $C_1$ and $C_2$ satisfy $b_1$ as well as $b_2$. Applying the weak convexity property twice, we obtain $C$ satisfies $b_1$ and $b_2$. This implies that $C$ satisfies $b_1 \wedge b_2$. Therefore $b_1 \wedge b_2$ satisfies the weak convexity property. $\square$

*Proof for Theorem 6.* Consider a controlling synchronization $\stackrel{S}{\rightarrow}$ for a predicate $b$ in a computation $\langle E, \rightarrow \rangle$ and let $\stackrel{C}{\rightarrow}$ be $(\rightarrow \cup \stackrel{S}{\rightarrow})^+$.

(*optimal* $\Rightarrow$ *smallest*) Assume that $\stackrel{S}{\rightarrow}$ is the optimal controlling synchronization. Consider an irreflexive partial order $\leadsto$ that extends $\rightarrow$. Our obligation is to establish that $b$ is invariant in $\langle E, \leadsto \rangle$ if and only if $\leadsto$ contains $\stackrel{S}{\rightarrow}$. We have,

$$
\begin{aligned}
& \langle E, \leadsto \rangle \models \mathit{invariant}: b \\
\equiv \; & \{ \text{ definition of } \mathit{invariant}: b \text{ } \} \\
& \langle \forall \mapsto : \langle E, \mapsto \rangle \text{ is a run of } \langle E, \leadsto \rangle : \\
& \qquad \langle E, \mapsto \rangle \models \mathit{invariant}: b \rangle \\
\equiv \; & \{ \leadsto \text{ extends } \rightarrow \} \\
& \Big\langle \forall \mapsto : \langle E, \mapsto \rangle \text{ is a run of } \langle E, \leadsto \rangle : \\
& \qquad (\langle E, \mapsto \rangle \text{ is a run of } \langle E, \rightarrow \rangle) \wedge \\
& \qquad (\langle E, \rightarrow \rangle \models \mathit{invariant} : b \Big\rangle
\end{aligned}
$$

$$\equiv \left\{\begin{array}{l}\text{definition of } \overset{S}{\rightarrow}\text{, which is the optimal}\\ \text{controlling synchronization}\end{array}\right\}$$
$$\Big\langle \forall \mapsto : \langle E,\mapsto\rangle \text{ is a run of } \langle E,\rightsquigarrow\rangle : \\ (\langle E,\mapsto\rangle \text{ is a run of } \langle E,\rightarrow\rangle \text{ and } \langle E,\overset{C}{\rightarrow}\rangle)\Big\rangle$$
$$\equiv \{ \overset{C}{\rightarrow} \text{ extends } \rightarrow \}$$
$$\Big\langle \forall \mapsto : \langle E,\mapsto\rangle \text{ is a run of } \langle E,\rightsquigarrow\rangle : \\ (\langle E,\mapsto\rangle \text{ is a run of } \langle E,\overset{C}{\rightarrow}\rangle)\Big\rangle$$
$$\equiv \{ \text{definition of a run} \}$$
$$\langle \forall \mapsto : \langle E,\mapsto\rangle \text{is a run of } \langle E,\rightsquigarrow\rangle : \mapsto \text{ contains } \overset{C}{\rightarrow}\rangle$$
$$\equiv \{ \text{partial-order algebra} \}$$
$$\rightsquigarrow \text{ contains } \overset{C}{\rightarrow}$$
$$\equiv \{ \rightsquigarrow \text{ is an irreflexive partial order that extends } \rightarrow \}$$
$$\rightsquigarrow \text{ contains } \overset{S}{\rightarrow}$$

(*smallest* $\Rightarrow$ *optimal*) Assume that $\overset{S}{\rightarrow}$ is the smallest controlling synchronization. Consider a run $\langle E,\rightsquigarrow\rangle$ of $\langle E,\rightarrow\rangle$. Our obligation is to establish that $b$ is invariant in $\langle E,\rightsquigarrow\rangle$ if and only if $\langle E,\rightsquigarrow\rangle$ is a run of $\langle E,\overset{C}{\rightarrow}\rangle$. We have,

$$\langle E,\rightsquigarrow\rangle \models \mathit{invariant}: b$$
$$\equiv \{ \text{definition of } \rightsquigarrow \}$$
$$(\rightsquigarrow \text{ extends } \rightarrow) \wedge (\langle E,\rightsquigarrow\rangle \models \mathit{invariant}: b)$$
$$\equiv \left\{\begin{array}{l}\text{definition of } \overset{S}{\rightarrow}\text{, which is the smallest}\\ \text{controlling synchronization}\end{array}\right\}$$
$$(\rightsquigarrow \text{ extends } \rightarrow) \wedge (\rightsquigarrow \text{ contains } \overset{S}{\rightarrow})$$
$$\equiv \{ \rightsquigarrow \text{ is an irreflexive partial order} \}$$
$$\rightsquigarrow \text{ contains } \overset{C}{\rightarrow}$$
$$\equiv \{ \text{definition of a run} \}$$
$$\langle E,\rightsquigarrow\rangle \text{ is a run of } \langle E,\overset{C}{\rightarrow}\rangle$$

This establishes the equivalence. □

*Proof for Lemma 9.* It suffices to prove that $\rightarrow \cup \overset{S^{(1)}}{\rightarrow}$ does not contain any cycle. Since $\rightarrow$ is an irreflexive partial order, a cycle, if it exists, must contain at least one pair of events ordered by $\overset{S^{(1)}}{\rightarrow}$. Moreover, since both $\rightarrow$ and $\overset{S^{(1)}}{\rightarrow}$ are transitive, the pairs of events in the cycle must be alternately ordered by $\rightarrow$ and $\overset{S^{(1)}}{\rightarrow}$. We first prove that there is no cycle containing exactly one pair of events ordered by $\overset{S^{(1)}}{\rightarrow}$. Assume the contrary. Then,

$$\langle \exists\, i,j :: s_i \overset{S^{(1)}}{\rightarrow} s_j \trianglerighteq s_i\rangle$$
$$\Rightarrow \{ \text{definition of } \overset{S^{(1)}}{\rightarrow} \}$$
$$\langle \exists\, i,j :: (i<j) \wedge (s_j \trianglerighteq s_i)\rangle$$
$$\Rightarrow \{ s \text{ satisfies the agreement property} \}$$
$$\langle \exists\, i,j :: (s_j \not\trianglerighteq s_i) \wedge (s_j \trianglerighteq s_i)\rangle$$
$$\Rightarrow \{ \text{predicate calculus} \}$$
$$\text{a contradiction}$$

We now prove that if there is a cycle that contains $m$, $m \geqslant 2$, pairs of events ordered by $\overset{S^{(1)}}{\rightarrow}$ then there is a cycle that contains strictly fewer than $m$ pairs of events ordered by $\overset{S^{(1)}}{\rightarrow}$. Let the cycle be $s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{C^{(1)}}{\trianglerighteq} s_i$, where the path from $s_v$ to $s_i$ contains exactly $m-2$ pair(s) of events ordered by $\overset{S^{(1)}}{\rightarrow}$. Since $\overset{S^{(1)}}{\rightarrow}$ is a total order, either $s_i \overset{S^{(1)}}{\trianglerighteq} s_v$ or $s_v \overset{S^{(1)}}{\trianglerighteq} s_i$. We have,

Case 1: $s_i \overset{S^{(1)}}{\trianglerighteq} s_v$

$$(s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{C^{(1)}}{\trianglerighteq} s_i) \wedge (s_i \overset{S^{(1)}}{\trianglerighteq} s_v)$$
$$\Rightarrow \{ \text{simplifying} \}$$
$$s_i \overset{S^{(1)}}{\trianglerighteq} s_v \overset{C^{(1)}}{\trianglerighteq} s_i$$
$$\Rightarrow \{ \text{simplifying} \}$$
a cycle with at most $m-1$ pair(s) of events ordered by $\overset{S^{(1)}}{\rightarrow}$

Case 2: $s_v \overset{S^{(1)}}{\trianglerighteq} s_i$

$$(s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{C^{(1)}}{\trianglerighteq} s_i) \wedge (s_v \overset{S^{(1)}}{\trianglerighteq} s_i)$$
$$\Rightarrow \{ \text{simplifying} \}$$
$$s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{S^{(1)}}{\trianglerighteq} s_i$$
$$\equiv \{ \text{rewriting} \}$$
$$s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{S^{(1)}}{\trianglerighteq} s_i \overset{S^{(1)}}{\rightarrow} s_j$$
$$\Rightarrow \{ \overset{S^{(1)}}{\rightarrow} \text{ is transitive} \}$$
$$s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_j$$
$$\Rightarrow \{ \text{simplifying} \}$$
a cycle with at most one pair of events ordered by $\overset{S^{(1)}}{\rightarrow}$

This establishes that there is no cycle in $\rightarrow \cup \overset{S^{(1)}}{\rightarrow}$ and therefore $\overset{C^{(1)}}{\rightarrow}$ is an irreflexive partial order. □

*Proof for Lemma 10.* It suffices to prove that $\overset{C^{(1)}}{\rightarrow} \cup \overset{S^{(2)}}{\rightarrow}$ does not contain any cycle. Since, from Lemma 9, $\overset{C^{(1)}}{\rightarrow}$ is an irreflexive partial order, a cycle, if it exists, must contain at least one pair of events ordered by $\overset{S^{(2)}}{\rightarrow}$. We first prove that there is no cycle containing exactly one pair of events ordered by $\overset{S^{(2)}}{\rightarrow}$. Assume the contrary. We have,

$$\langle \exists\, i :: s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\trianglerighteq} s_{i+1}\rangle$$
$$\Rightarrow \left\{\begin{array}{l}\text{by definition of } \overset{S^{(2)}}{\rightarrow}, proc(s_{i+1}) \neq proc(s_i)\\ \text{implying } s_{i+1} \neq succ(s_i)\end{array}\right\}$$
$$\langle \exists\, i :: s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\rightarrow} s_{i+1}\rangle$$
$$\Rightarrow \left\{\begin{array}{l}\text{since } s \text{ satisfies the continuity property,}\\ succ(s_i) \not\rightarrow s_{i+1}\end{array}\right\}$$
$$\langle \exists\, i,j,k :: s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \trianglerighteq s_j \overset{S^{(1)}}{\rightarrow} s_k \overset{C^{(1)}}{\trianglerighteq} s_{i+1}\rangle$$
$$\Rightarrow \{ \overset{S^{(1)}}{\rightarrow} \text{ is a total order on } s \}$$

$\langle \exists\, i,j :: (s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \underline{\rightarrow}\, s_j \overset{C^{(1)}}{\rightarrow} s_{i+1}) \wedge$
$\quad ((s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_j) \vee (s_j \overset{S^{(1)}}{\rightarrow} s_{i+1}))\rangle$

$\Rightarrow$ { $s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_j$ implies $s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_j \overset{C^{(1)}}{\rightarrow} s_{i+1}$, which contradicts Lemma 9 }

$\langle \exists\, i,j :: (s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \underline{\rightarrow}\, s_j \overset{C^{(1)}}{\rightarrow} s_{i+1}) \wedge$
$\quad (s_j \overset{S^{(1)}}{\rightarrow} s_{i+1})\rangle$

$\Rightarrow$ { $s_i \overset{P}{\rightarrow} succ(s_i)$ and $\overset{P}{\rightarrow} \subseteq \rightarrow$ }

$\langle \exists\, i,j :: (s_i \rightarrow s_j) \wedge (s_j \overset{S^{(1)}}{\rightarrow} s_{i+1})\rangle$

$\Rightarrow$ { $\overset{S^{(1)}}{\rightarrow}$ is a total order on $s$ and $s$ satisfies the agreement property }

$\langle \exists\, i,j :: (s_i \overset{S^{(1)}}{\rightarrow} s_j) \wedge (s_j \overset{S^{(1)}}{\rightarrow} s_{i+1})\rangle$

$\Rightarrow$ { $s$ satisfies the agreement property }

$\langle \exists\, i,j :: i < j < i+1\rangle$

$\Rightarrow$ { $i$ and $j$ are integers }

a contradiction

We now prove that if there is a cycle that contains $m$, $m \geqslant 2$, pairs of events ordered by $\overset{S^{(2)}}{\rightarrow}$ then there is a cycle that contains strictly fewer than $m$ pairs of events ordered by $\overset{S^{(2)}}{\rightarrow}$. Let the cycle be $s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\underline{\rightarrow}} s_{j+1} \overset{S^{(2)}}{\rightarrow} succ(s_j) \overset{C^{(2)}}{\underline{\rightarrow}} s_{i+1}$, where the path from $succ(s_j)$ to $s_{i+1}$ contains exactly $m-2$ pair(s) of events ordered by $\overset{S^{(2)}}{\rightarrow}$. Since $\overset{S^{(1)}}{\rightarrow}$ is a total order, either $s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{j+1}$ or $s_{j+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{i+1}$. We have,

Case 1: $s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{j+1}$

$(s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\underline{\rightarrow}} s_{j+1} \overset{S^{(2)}}{\rightarrow} succ(s_j) \overset{C^{(2)}}{\underline{\rightarrow}} s_{i+1})$
$\wedge\, (s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{j+1})$

$\Rightarrow$ { simplifying }

$s_{i+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{j+1} \overset{S^{(2)}}{\rightarrow} succ(s_j) \overset{C^{(2)}}{\underline{\rightarrow}} s_{i+1}$

$\Rightarrow$ { simplifying }

a cycle with at most $m-1$ pair(s) of events ordered by $\overset{S^{(2)}}{\rightarrow}$

Case 2: $s_{j+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{i+1}$

$(s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\underline{\rightarrow}} s_{j+1} \overset{S^{(2)}}{\rightarrow} succ(s_j) \overset{C^{(2)}}{\underline{\rightarrow}} s_{i+1})$
$\wedge\, (s_{j+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{i+1})$

$\Rightarrow$ { simplifying }

$s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\underline{\rightarrow}} s_{j+1} \overset{S^{(1)}}{\underline{\rightarrow}} s_{i+1}$

$\Rightarrow$ { simplifying }

a cycle with at most one pair of events ordered by $\overset{S^{(2)}}{\rightarrow}$

This establishes that there is no cycle in $\overset{C^{(1)}}{\rightarrow} \cup \overset{S^{(2)}}{\rightarrow}$ and therefore $\overset{C^{(2)}}{\rightarrow}$ is an irreflexive partial order. □

*Proof for Lemma 18.* Assume that the weighted true event graph does contain a permissible path. Consider the shortest permissible path $s = s_1 s_2 \cdots s_l$. Assume, on the contrary, that $s$ does not satisfy the agreement property. Then there exist integers $i$ and $j$, where $i < j$, such that $s_j \rightarrow s_i$. Since $s$ is a shortest permissible path, $s_j \notin \bot$; if otherwise, the path $s_j s_{j+1} \cdots s_l$ is a shorter permissible path than $s$—a contradiction. Furthermore, $i \geqslant 2$; if otherwise, $s_i \in \bot$ which implies that $s_i \rightarrow s_j$, thereby creating a cycle in $\rightarrow$. Two possible cases arise depending on whether there is an edge from $s_{i-1}$ to $s_j$.

Case 1: $(s_{i-1}, s_j) \notin \mathsf{E}(G)$

{ definition of an edge }

$(succ(s_{i-1}) \rightarrow s_j) \wedge (s_j \rightarrow s_i)$

$\Rightarrow$ { $\rightarrow$ is transitive }

$succ(s_{i-1}) \rightarrow s_i$

$\equiv$ { definition of an edge }

$(s_{i-1}, s_i) \notin \mathsf{E}(G)$

$\Rightarrow$ { $s$ is a path implying $(s_{i-1}, s_i) \in \mathsf{E}(G)$ }

a contradiction

In the second case, two possible sub-cases arise depending on the weight of the edge from $s_{i-1}$ to $s_j$. If $w(s_{i-1}, s_j) = (0,1)$ then the path $s_1 s_2 \cdots s_{i-1} s_j \cdots s_l$ is permissible and has lesser weight than $s$—a contradiction. The more interesting case is when $w(s_{i-1}, s_j) = (1,1)$. Then,

Case 2.2: $w(s_{i-1}, s_j) = (1,1)$

{ definition of the weight function }

$s_j \not\underline{\rightarrow}\, succ(s_{i-1})$

$\Rightarrow$ { $s_j \rightarrow s_i$ implying $s_i \underline{\rightarrow}\, succ(s_{i-1}) \Rightarrow\ s_j \underline{\rightarrow}\, succ(s_{i-1})$ }

$s_i \not\underline{\rightarrow}\, succ(s_{i-1})$

$\equiv$ { $(s_{i-1}, s_i) \in \mathsf{E}(G)$ and definition of the weight function }

$w(s_{i-1}, s_i) = (1,1)$

Thus the path $s_1 s_2 \cdots s_{i-1} s_j \cdots s_l$ is permissible and has lesser weight than $s$—a contradiction. This establishes that $s$ satisfies the agreement property. □

*Proof for Theorem 22.* Assume that the weighted true event graph $G$ does contain a permissible path. From Theorem 17, $b$ is controllable in $\langle E, \rightarrow\rangle$. Let $\overset{\min}{\rightarrow}$ denote a minimum controlling synchronization for $b$ in $\langle E, \rightarrow\rangle$. Further, let $\{G^{(k)}\}$ represent the sequence of weighted true event graphs generated by adding synchronization dependencies from $\overset{\min}{\rightarrow}$ one-by-one, where $G^{(0)} = G$. Note that $b$ is invariant in the computation obtained by adding all synchronization dependencies from $\overset{\min}{\rightarrow}$. From the bounded reduction lemma,

$$rank(G^{(i)}) - rank(G^{(i+1)}) \leqslant 1, \quad 0 \leqslant i < |\overset{\min}{\rightarrow}|$$

Adding the above inequality for all values of $i$, we obtain,

$$rank(G^{(0)}) - rank(G^{|\overset{\min}{\rightarrow}|}) \leqslant |\overset{\min}{\rightarrow}|$$

$\equiv$ { using Theorem 21 }

$$rank(G) - 0 \leqslant |\overset{\min}{\rightarrow}|$$

$\equiv$ { simplifying }

$$rank(G) \leqslant |\overset{\min}{\rightarrow}|$$

$\equiv$ { $\overset{\min}{\rightarrow}$ corresponds to a minimum controlling synchronization }

$$rank(G) = |\overset{\min}{\rightarrow}|$$

This establishes the theorem. □

**Neeraj Mittal** received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a co-director of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking and databases.

**Vijay K. Garg** received his B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur in 1984 and M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1985 and 1988, respectively. He is currently a full professor in the Department of Electrical and Computer Engineering and the director of the Parallel and Distributed Systems Laboratory at the University of Texas, Austin. His research interests are in the areas of distributed systems and discrete event systems. He is the author of the books Elements of Distributed Computing (Wiley & Sons, 2002), Principles of Distributed Systems (Kluwer, 1996) and a co-author of the book Modeling and Control of Logical Discrete Event Systems (Kluwer, 1995).