# Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution *

Ashis Tarafdar[1] and Vijay K. Garg[2]

[1] Dept. of Computer Sciences, The Univ. of Texas at Austin, Austin, TX 78712
ashis@cs.utexas.edu
[2] Dept. of Electr. and Comp. Engg., The Univ. of Texas at Austin, Austin, TX 78712
garg@ece.utexas.edu

**Abstract.** Concurrent programs often encounter failures, such as races, owing to the presence of synchronization faults (bugs). One existing technique to tolerate synchronization faults is to roll back the program to a previous state and re-execute, in the hope that the failure does not recur. Instead of relying on chance, our approach is to control the re-execution in order to avoid a recurrence of the synchronization failure. The control is achieved by tracing information during an execution and using this information to add synchronizations during the re-execution.
The approach gives rise to a general problem, called the *off-line predicate control problem*, which takes a computation and a property specified on the computation, and outputs a "controlled" computation that maintains the property. We solve the predicate control problem for the mutual exclusion property, which is especially important in synchronization fault tolerance.

## 1 Introduction

Concurrent programs are difficult to write. The programmer is presented with the task of balancing two competing forces: safety and liveness [8]. Frequently, the programmer leans too much in one of the two directions, causing either safety failures (e.g. races) or liveness failures (e.g. deadlocks). Such failures arise from a particular kind of software fault (bug), known as a *synchronization fault*. Studies have shown that synchronization faults account for a sizeable fraction of observed software faults in concurrent programs [6]. Locating synchronization faults and eliminating them by reprogramming is always the best strategy. However, many systems must maintain availability in spite of software failures. It is, therefore, desirable to be able to bypass a synchronization fault and recover from the resulting failure. This problem of software fault tolerance for synchronization faults in concurrent programs [1] is the primary motivation for this paper.

* supported in part by the NSF ECS-9414780, CCR-9520540, a General Motors Fellowship, Texas Education Board ARP-320 and an IBM grant

[1] By concurrent programs, we include all parallel programming paradigms such as: multi-threaded programs, shared-memory parallel programs, message-passing distributed programs, distributed shared-memory programs, etc. We will refer to a parallel entity as a process, although in practice it may also be a thread.

Traditionally, it was believed that software failures are permanent in nature and, therefore, they would recur in every execution of the program with the same inputs. This belief led to the use of design diversity to recover from software failures. In approaches based on design diversity [1, 13], redundant modules with different designs are used, ensuring that there is no single point-of-failure. Contrary to this belief, it was observed that many software failures are, in fact, *transient* - they may not recur when the program is re-executed with the same inputs [3]. In particular, the failures caused by synchronization faults are usually transient in nature.

The existence of transient software failures motivated a new approach to software fault tolerance based on rolling back the processes to a previous state and then restarting them (possibly with message reordering), in the hope that the transient failure will not recur in the new execution [5, 17]. Methods based on this approach have relied on chance in order to recover from a transient software failure. In the special case of synchronization faults, however, it is possible to do better. Instead of leaving recovery to chance, our approach ensures that the transient synchronization failure does not recur. It does so by controlling the re-execution, based on information traced during the failed execution.

Our mechanism involves (i) tracing an execution, (ii) detecting a synchronization failure, (iii) determining a control strategy, and (iv) re-executing under control. Each of these problems is also of independent interest. Our requirements for tracing an execution and re-execution under control are very similar to trace-and-replay techniques in concurrent debugging. Trace-and-replay techniques have been studied in various concurrent paradigms such as message-passing parallel programs [12], shared-memory parallel programs [11], distributed shared memory programs [15], and multi-threaded programs [2]. Among sychronization failures, this paper will focus on races. Race detection has been previously studied [4, 10]. We will discuss tracing, failure detection, and re-execution under control in greater depth in Section 4. This paper addresses the remaining problem of determining a control strategy.

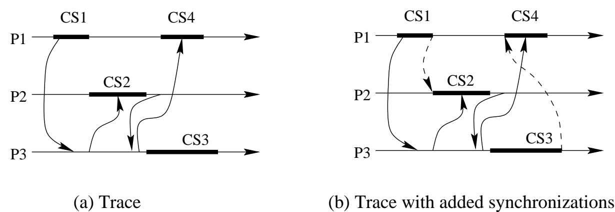


**Fig. 1.** Example: Tracing and Controlling During Rollback Recovery

To illustrate what determining a control strategy involves, consider the execution shown in Figure 1(a). CS1 - CS4 are the critical sections of the execution. The synchronizations between processes are shown as arrows from one process execution to another. A synchronization ensures that the execution after the head

of the arrow can proceed only after the execution before the tail has completed. A race occurs when two critical sections execute simultaneously. For example, CS1 and CS2 may have a race, since the synchronizations do not prevent them from executing simultaneously. A control strategy is a set of *added synchronizations* that would ensure that a race does not occur. The race can be avoided by adding synchronizations, shown as broken arrows in Figure 1(b).

The focus of this paper is the problem of determining which synchronizations to add to an execution trace in order to tolerate a synchronization fault in the re-execution. This proves to be an important problem in its own right and can be applied in areas other than software fault tolerance, such as concurrent debugging. We generalize the problem using a framework known as the *off-line predicate control* problem. This problem was introduced in [16], where it was applied to concurrent debugging. Informally, off-line predicate control specifies that, given a computation and a property on the computation, one must determine a controlled computation (one with more synchronizations) that maintains the property. (We will use the term *computation* for a formal model of an *execution*.) The previous work [16] solved the predicate control problem for a class of properties called *disjunctive predicates*. Applying the results of that study to software fault tolerance would mean avoiding synchronization failures of the form: $l_1 \wedge l_2 \wedge l_3$, where $l_i$ is a local property specified on process $P_i$. For example, if $l_i$ specifies that a server is unavailable, the synchronization failure is that all servers are unavailable at the same time.

In this paper, we address a class of off-line predicate control problems, characterized by the mutual exclusion property, that is especially useful in tolerating races. We consider four classes of mutual exclusion properties: *off-line mutual exclusion*, *off-line readers writers*, *off-line independent mutual exclusion*, and *off-line independent read-write mutual exclusion*. For each of these classes of properties, we determine necessary and sufficient conditions under which the problem may be solved. Furthermore, we design an efficient algorithm that solves the most general of the problems, off-line independent read-write mutual exclusion, and thus also solves each of the other three problems. The algorithm takes $O(np)$ time, where $n$ is the number of concurrent processes and $p$ is the number of critical sections.

The problems have been termed *off-line* problems to distinguish them from their more popular *on-line* variants (i.e. the usual mutual exclusion problems [14]). The difference between the on-line and off-line problems is that in the on-line case, the computation is provided on-line, whereas in the off-line case, the computation is known *a priori*. Ignorance of the future makes on-line mutual exclusion a harder problem to solve. In general, in on-line mutual exclusion, one cannot avoid deadlocks without making some assumptions (e.g. critical sections do not block). Thus, on-line mutual exclusion is impossible to solve. To understand why this is true, consider the scenario in Figure 1. Any on-line algorithm, being unaware of the future computation, would have a symmetric choice of entering CS1 or CS2 first. If CS2 is entered first, it would result in a deadlock. An off-line algorithm, being aware of the future computation, could make the

correct decision to enter CS1 first and add a synchronization from CS1 to CS2. A proof of the impossibility of on-line mutual exclusion follows along similar lines as the proof of Theorem 3 in [16]. Thus, there will always be scenarios where on-line mutual exclusion algorithms will fail, resulting in either race conditions or deadlocks. In such scenarios, controlled re-execution based on off-line mutual exclusion becomes vitally important.

## 2 Model and Problem Statement

The model that we present is of a single execution of the concurrent program. The model is not at the programming language level, but at a lower level, at which the execution consists of a sequence of states for each process and the communications that occurred among them (similar to the happened before model[7]).

Let $S$ be a finite set of elementary entities known as *states*. $S$ is partitioned into subsets $S_1, S_2, \cdots, S_n$, where $n > 1$. These partitions correspond to $n$ processes in the system. A subset $G$ of $S$ is called a *global state* iff $\forall i: \ |G \cap S_i| = 1$. Let $G_i$ denote the unique element in $G \cap S_i$. A *global predicate* is a function that maps a global state onto a boolean value.

A *computation* is a partial order $\rightarrow$ on $S$ such that $\forall i: \rightarrow_i$ is a total order on $S_i$, where $\rightarrow_i$ represents $\rightarrow$ restricted to the set $S_i$. Note that the states in a single process are totally ordered while the states across processes are partially ordered. We will use $\rightarrow$, $\rightarrow^k$, $\rightarrow^c$ to denote computations, and $\|$, $\|^k$, $\|^c$ to denote the respective incomparability relations (e.g. $s \parallel t \ \equiv \ (s \not\rightarrow t) \ \wedge \ (t \not\rightarrow s)$). Given a computation $\rightarrow$ and a subset $K$ of $S$, $\rightarrow$*-consistent*$(K) \ \equiv \ \forall s,t \in K: \ s \parallel t$. In particular, a global state may be $\rightarrow$-consistent. The notion of consistency tells us when a set of states could have occurred concurrently in a computation.

A computation $\rightarrow$ is *extensible* in $S$ iff:

$$\forall K \subseteq S : \rightarrow\text{-consistent}(K) \ \Rightarrow \ \exists \text{ global state } G \supseteq K : \rightarrow\text{-consistent}(G)$$

Intuitively, extensibility allows us to extend a consistent set of states to a consistent global state. Any computation in $S$ can be made extensible by adding "dummy" states to $S$. Therefore, we implicitly assume that any computation is extensible.

Given a computation $\rightarrow$, let $\leq$ be a relation on global states defined as: $G \leq H \ \equiv \ \forall i: \ (G_i \rightarrow_i H_i) \ \vee \ (G_i = H_i)$. It is a well-known fact that the set of $\rightarrow$-consistent global states is a lattice with respect to the $\leq$ relation [9]. In particular, we will use $\rightarrow$*-glb*$(G,H)$ for the greatest lower bound of $G$ and $H$ with respect to $\rightarrow$ (so, $\rightarrow\text{-glb}(G,H)_i = \rightarrow_i\text{-min}(G_i,H_i)$). If $G$ and $H$ are $\rightarrow$-consistent, then $\rightarrow\text{-glb}(G,H)$ is also $\rightarrow$-consistent.

Given a computation $\rightarrow$ and a global predicate $B$, a computation $\rightarrow^c$ is called a *controlling computation* of $B$ in $\rightarrow$ iff (1) $\rightarrow \subseteq \rightarrow^c$, and (2) $\forall G: \rightarrow^c$-consistent$(G) \ \Rightarrow \ B(G)$. This tells us that a controlling computation is a stricter partial order (containing more synchronizations). Further, any global state that may occur in the controlling computation must satisfy the specified global predicate. Thus, the problem of finding a controlling computation is to

add synchronizations until all global states that violate the global predicate are made inconsistent. More formally,

**The Off-line Predicate Control Problem:** Given a computation $\rightarrow$ and a global predicate $B$, find a controlling computation of $B$ in $\rightarrow$

## 3 Solving the Off-line Predicate Control Problem

In [16], it was proved that the Off-line Predicate Control is NP-Hard. Therefore, it is important to solve useful restricted forms of the Off-line Predicate Control Problem. Since we are interested in avoiding race conditions, we restrict the general problem by letting $B$ specify the mutual exclusion property.

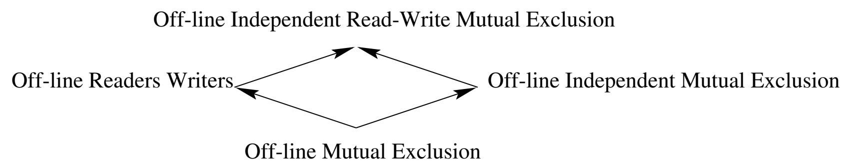


**Fig. 2.** Variants of Off-line Mutual Exclusion

The simplest specification for mutual exclusion is: no two critical sections execute at the same time. This corresponds to the semantics of a single exclusive lock for all the critical sections. We call the corresponding problem the *Off-line Mutual Exclusion Problem*. We can generalize this to the *Off-line Readers Writers Problem* by specifying that only critical sections that "write" must be exclusive, while critical sections that "read" need not be exclusive. This corresponds to the semantics of read-exclusive-write locks. Another way to generalize the Off-line Mutual Exclusion Problem is to allow the semantics of independent locks. In this *Off-line Independent Mutual Exclusion Problem*, no two critical sections of the same lock can execute simultaneously. Finally, we can have critical sections with the semantics of independent read-exclusive-write locks. This is the *Off-line Independent Read-Write Mutual Exclusion Problem*. Figure 2 illustrates the relative generality of the four problems.

In traditional *on-line* mutual exclusion, there has been no "independent" variant, since it trivially involves applying the same algorithm for each lock. However, in off-line mutual exclusion, such an approach will not work, since the synchronizations added by each independent algorithm may cause deadlocks when applied together.

For the practitioner, an algorithm which solves Off-line Independent Read-Write Mutual Exclusion would suffice, since it can be used to solve all other variants. However, for the purpose of presentation we will start with the simplest Off-line Mutual Exclusion Problem and then generalize it in steps. For each problem, we will determine the necessary and sufficient conditions for finding a

solution. Finally, we will make use of the results in the design of an algorithm which solves the most general of the four problems.

### 3.1 Off-line Mutual Exclusion

Off-line Mutual Exclusion is a specialization of Off-line Predicate Control to the following class of global predicates:

$$B_{mutex}(G) \equiv \forall \; distinct \; s,t \in G : \neg(critical(s) \; \wedge \; critical(t))$$

where *critical* is a function that maps a state onto a boolean value. Thus, $B_{mutex}$ specifies that at most one process may be critical in a global state.

Based on the *critical* boolean function on states, we define *critical sections* as maximal intervals of critical states. More precisely: given a *critical* function on $S$ and a computation $\rightarrow$ on $S$, a *critical section*, $CS$, is a non-empty, maximal subset of an $S_i$ such that: (1) $\forall s \in CS : \; critical(s)$, and (2) $\forall s,t \in CS : \; \forall u \in S_i : \; s \rightarrow u \rightarrow t \; \Rightarrow \; u \in CS$.

Let $CS.first$ and $CS.last$ be the minimum and maximum states respectively in $CS$ (w.r.t. $\rightarrow_i$). Let $\mapsto$ be a relation on critical sections defined as: $CS \mapsto CS' \; \equiv \; CS.first \rightarrow CS'.last \; \wedge \; CS \neq CS'$ Thus, $\mapsto$ orders a critical section before another if some state in the first happened before some state in the second. Note that $\mapsto$ may have cycles.

We will be dealing with different computations. All computations will have the same total order $\rightarrow_i$ for each $S_i$. Therefore, the set of critical sections will not change for each computation. However, the $\mapsto$ relation will change, in general. For computations $\rightarrow$, $\rightarrow^k$, and $\rightarrow^c$, the relation on critical sections will be denoted as $\mapsto$, $\mapsto^k$ and $\mapsto^c$ respectively.

**Theorem 1 (Necessary Condition)** *For a computation $\rightarrow$ of $S$, and a global predicate $B_{mutex}$,*

*a controlling computation of $B_{mutex}$ in $\rightarrow$ exists $\quad \Rightarrow \quad \mapsto$ has no cycles*

**Proof:** We prove the contrapositive. Let $\mapsto$ have a cycle, say $CS_1 \mapsto CS_2 \mapsto \cdots CS_m \mapsto CS_1$, $(m \geq 2)$ and let $\rightarrow^c$ be a computation such that $\rightarrow \subseteq \rightarrow^c$. Since $\rightarrow^c$ cannot have a cycle, at least one of:
$CS_1.last \not\rightarrow^c CS_2.first$, $CS_2.last \not\rightarrow^c CS_3.first$, $\cdots$, and $CS_m.last \not\rightarrow^c CS_1.first$ must hold. Without loss of generality, let $CS_1.last \not\rightarrow^c CS_2.first$. We also have $CS_2.last \not\rightarrow^c CS_1.first$ (since $CS_1 \mapsto CS_2$). Since $\rightarrow^c$ is extensible, we can define $s_2$ as the maximum state in $S_2$ such that $CS_1.last \; \|^c \; s_2$, and $s_1$ as the maximum state in $S_1$ such that $CS_2.last \; \|^c \; s_1$. By extensibility of $\rightarrow^c$, we can find $\rightarrow^c$-consistent global states $G_1$ and $G_2$ containing $\{CS_1.last, s_2\}$ and $\{CS_2.last, s_1\}$ respectively. We now have two cases:

*Case 1:* $[s_1 \in CS_1 \; \vee \; s_2 \in CS_2]$ In this case $\neg B_{mutex}(G_2) \; \vee \; \neg B_{mutex}(G_1)$.

*Case 2:* $[s_1 \notin CS_1 \; \wedge \; s_2 \notin CS_2]$ Since $s_1 \notin CS_1$, there are two ways to position $s_1$: (a) $s_1 \rightarrow^c CS_1.first$ or (b) $CS_1.last \rightarrow^c s_1$. In sub-case (a), since $CS_2.last \not\rightarrow^c CS_1.first$, either $CS_2.last \|^c CS_1.first$ or $CS_1.first \rightarrow^c CS_2.last$, which gives us $s_1 \rightarrow^c CS_2.last$. Both possibilities contradict the definition of

$s_1$. This leaves sub-case (b) as the only possibility. Therefore, $CS_1.last \rightarrow^c s_1$. Similarly, we can prove $CS_2.last \rightarrow^c s_2$. Let $H = \rightarrow^c$-glb$(G_1, G_2)$. $H$ contains $CS_1.last$ and $CS_2.last$ and, so, $\neg B_{mutex}(H)$. Further, $H$ is $\rightarrow^c$-consistent (by the lattice property).

So in either case, $\rightarrow^c$ is not a controlling computation of $B_{mutex}$ in $\rightarrow$. □

**Theorem 2 (Sufficient Condition)** *For a computation $\rightarrow$ of $S$, and a global predicate $B_{mutex}$,*

$$\mapsto \textit{ has no cycles} \quad \Rightarrow \quad \textit{a controlling computation of } B_{mutex} \textit{ in} \rightarrow \textit{exists}$$

**Proof:** Since $\mapsto$ has no cycles, we can arrange all of the critical sections in a sequence: $CS_1, CS_2, \cdots CS_m$ such that $CS_i \mapsto CS_j \;\Rightarrow\; i < j$. Let $\rightarrow^c$ be defined as $(\rightarrow \;\cup \{(CS_i.last, CS_{i+1}.first) \mid 1 \leq i \leq m-1\})^+$, where $()^+$ is the transitive closure. Clearly $\rightarrow \subseteq \rightarrow^c$. In the next paragraph, we will prove that $\rightarrow^c$ is a partial order. Assume that there is a global state $G$ such that $\neg B_{mutex}(G)$. Therefore, we can find states $s$ and $t$ such that $critical(s)$ and $critical(t)$. Let $CS_i$ and $CS_j$ be the two critical sections to which $s$ and $t$ belong respectively. w.l.o.g, let $i < j$. Therefore, $s \rightarrow^c t$, and $\neg \rightarrow^c$-consistent$(G)$. Therefore, $\rightarrow^c$-consistent$(G) \;\Rightarrow\; B_{mutex}(G)$. So $\rightarrow^c$ is a controlling computation of $B_{mutex}$ in $\rightarrow$.

Our remaining proof obligation is to prove that $\rightarrow^c$ is a partial order. To this end, let $\rightarrow^k$ be defined as: $(\rightarrow \;\cup \{(CS_i.last, CS_{i+1}.first) \mid 1 \leq i \leq k-1\})^+$. We make the following claim:

*Claim:* $\forall 1 \leq k \leq m$ : (1) $\rightarrow^k$ is a partial order, and (2) $CS_i \mapsto^k CS_j \;\Rightarrow\; i < j$

Clearly $\rightarrow^c = \rightarrow^m$ and so this claim implies that $\rightarrow^c$ is a partial order.

*Proof of Claim:* (by Induction on $k$)

*Base Case:* Immediate from $\rightarrow = \rightarrow^1$.

*Inductive Case:* We make the inductive hypothesis that $\rightarrow^{k-1}$ is a partial order, and that $CS_i \mapsto^{k-1} CS_j \;\Rightarrow\; i < j$. We may rewrite the definition of $\rightarrow^k$ as: $(\rightarrow^{k-1} \cup \{(CS_{k-1}.last,\, CS_k.first)\})^+$. First we demonstrate that $\rightarrow^k$ is irreflexive and transitive (which together imply asymmetry).

*(i) Irreflexivity:* Let $s \rightarrow^k t$. There are two possibilities: either $s \rightarrow^{k-1} t$ or $s \rightarrow^{k-1} CS_{k-1}.last \wedge CS_k.first \rightarrow^{k-1} t$. In the first case, the inductive hypothesis tells us that $\rightarrow^{k-1}$ is irreflexive and so $s \neq t$. In the second case, part (1) of the inductive hypothesis tells us that $\rightarrow^{k-1}$ is transitive, and part (2) of the inductive hypothesis tells us that $CS_k.first \not\rightarrow^{k-1} CS_{k-1}.last$ and so $s \neq t$.

*(ii) Transitivity:* This is immediate from the definition of $\rightarrow^k$.

Therefore, $\rightarrow^k$ is a partial order. We now show the second part of the claim. Suppose $CS_i \mapsto^k CS_j$. This implies that $CS_i.first \rightarrow^k CS_j.last \wedge i \neq j$. There are two cases: either $CS_i.first \rightarrow^{k-1} CS_j.last \;\wedge\; i \neq j$ or $CS_i.first \rightarrow^{k-1} CS_{k-1}.last \;\wedge\; CS_k.first \rightarrow^{k-1} CS_j.last \;\wedge i \neq j$. In the first case, we have $CS_i \mapsto^{k-1} CS_j$ and so by the inductive hypothesis, $i < j$. In a similar manner, the second case would give us $i \leq k-1 \;\wedge\; k \leq j$ and so $i < j$. □

In conclusion, the necessary and sufficient condition for finding a controlling computation for $B_{mutex}$ is that there is no cycle of critical sections with respect to $\mapsto$. Further note that, since the proof of Theorem 2 is constructive, we can

use it to design a naive algorithm to find a controlling computation. (We will see why this algorithm is naive in Section 3.5).

### 3.2 Off-line Readers Writers Problem

Let $read_critical$ and $write_critical$ be functions that map a state onto a boolean value. Further, no state can be both $read_critical$ and $write_critical$ (any read and write locked state is considered to be only write locked). Let $critical(s) \equiv read_critical(s) \lor write_critical(s)$. The Off-line Readers Writers Problem is a specialization of the Off-line Predicate Control Problem to the following class of global predicates:

$$B_{rw}(G) \equiv \forall \, distinct \; s, t \in G : \neg(write_critical(s) \land critical(t))$$

Given a $read_critical$ function and a $write_critical$ function on $S$ and a computation $\rightarrow$ on $S$, we define a *read critical section* and a *write critical section* in an analogous fashion to the critical sections that we defined before. Note that, since no state is both $read_critical$ and $write_critical$, critical sections in a process do not overlap.

Let $\mapsto$ be a relation on both read and write critical sections defined as: $CS \mapsto CS' \equiv CS.first \rightarrow CS'.last \land CS \neq CS'$

**Theorem 3 (Necessary Condition)** *For a computation $\rightarrow$ of $S$, and a global predicate $B_{rw}$,*

| | | |
|---|---|---|
| *a controlling computation of $B_{rw}$ in $\rightarrow$ exists* | $\Rightarrow$ | *all cycles in $\mapsto$ contain only read critical sections* |

**Proof:** The proof is similar to the proof of Theorem 1. We will prove the contrapositive. Let $\mapsto$ have a cycle, say $CS_1 \mapsto CS_2 \mapsto \cdots CS_m \mapsto CS_1$. Without loss of generality, let $CS_1$ be a write critical section. Let $\rightarrow^c$ be a computation such that $\rightarrow \subseteq \rightarrow^c$.

First, we claim that there is at least one critical section in the cycle say $CS_k$ (where $k \neq 1$), such that $CS_1.last \not\rightarrow^c CS_k.first$ and $CS_k.last \not\rightarrow^c CS_1.first$. To prove this, we assume the opposite:

$\forall CS_k \; (k \neq 1) : \; CS_1.last \rightarrow^c CS_k.first \; \lor \; CS_k.last \rightarrow^c CS_1.first$ – (i)

and prove a contradiction as follows. $CS_m \mapsto CS_1$ implies $CS_1.last \not\rightarrow^c CS_m.first$. Therefore, by (i), $CS_m.last \rightarrow^c CS_1.first$. This allows us to define $j$ as the smallest integer such that $CS_j.last \rightarrow^c CS_1.first$. $CS_1 \mapsto CS_2$ implies that $CS_2.last \not\rightarrow^c CS_1.first$. Therefore, $j \neq 2$. In particular, $CS_{j-1}$ and $CS_1$ are distinct. By our choice of $j$, $CS_{j-1}.last \not\rightarrow^c CS_1.first$. So, using (i), $CS_1.last \rightarrow^c CS_{j-1}.first$. We now have a cycle: $CS_1.last \rightarrow^c CS_{j-1}.first$ (as above), $CS_{j-1}.first \rightarrow^c CS_j.last$ (since $CS_{j-1} \mapsto CS_j$), $CS_j.last \rightarrow^c CS_1.first$ (by our choice of $j$), and $CS_1.first \rightarrow^c CS_1.last$ (by the definition of $first$ and $last$). This cycle contradicts the fact that $\rightarrow^c$ is a partial order.

Since we have demonstrated the existence of a $CS_k$ such that $CS_1.last \not\rightarrow^c CS_k.first$ and $CS_k.last \not\rightarrow^c CS_1.first$, we can use a proof similar to the one in Theorem 1 to show that $\rightarrow^c$ is not a controlling computation of $B_{rw}$ in $\rightarrow$. □

**Theorem 4 (Sufficient Condition)** *For a computation $\rightarrow$ of $S$, and a global predicate $B_{rw}$,*

| | | |
|---|---|---|
| *all cycles in $\mapsto$ contain only read critical sections* | $\Rightarrow$ | *a controlling computation of $B_{rw}$ in $\rightarrow$ exists* |

**Proof:** Consider the set of strongly connected components of the set of critical sections with respect to the $\mapsto$ relation. Define the $\hookrightarrow$ relation on strongly connected components as $SCC \hookrightarrow SCC' \equiv \exists CS \in SCC, CS' \in SCC' : \ CS \mapsto CS' \ \wedge \ SCC \neq SCC'$. It is verifiable that $\hookrightarrow$ is a partial order. Therefore, we can linearize it to get a sequence of all strongly connected components, say $SCC_1, SCC_2, \cdots SCC_l$ such that $SCC_i \hookrightarrow SCC_j \ \Rightarrow \ i < j$. Let $\rightarrow^c$ be defined as $(\rightarrow \ \cup \{(CS_i.last, CS_j.first) \mid CS_i \in SCC_k, CS_j \in SCC_{k+1}$ for some $1 \leq k \leq l-1\})^+$. Clearly $\rightarrow \subseteq \rightarrow^c$. We can show that $\rightarrow^c$ is a partial order along similar lines as the proof of Theorem 2.

We now show that $\rightarrow^c$ is a controlling computation of $B_{rw}$ in $\rightarrow$. Suppose $G$ is a global state such that $\neg B_{rw}(G)$. Therefore, we can find states $s$ and $t$ such that $write_critical(s)$ and $critical(t)$. Let $CS$ be a write critical section that contains $s$ and let $CS'$ be a critical section that contains $t$. Let $SCC_i$ and $SCC_j$ be the strongly connected components that contain $CS$ and $CS'$ respectively. $SCC_i$ is distinct from $SCC_j$ since, otherwise, there would be a cycle in $\mapsto$ that contains a write critical section. Without loss of generality, let $i < j$. By the definition of $\rightarrow^c$, we have $s \rightarrow^c t$ and, therefore, $\neg \rightarrow^c$-consistent$(G)$. Therefore, $\rightarrow^c$ is a controlling computation of $B_{rw}$ in $\rightarrow$. $\square$

Note, as before, that the proof of Theorem 4 can be used to design an algorithm to find a controlling computation.

### 3.3 Off-line Independent Mutual Exclusion

Let $critical_1, critical_2, \cdots critical_m$ be functions that map an event onto a boolean value. The Off-line Independent Mutual Exclusion Problem is a specialization of the Off-line Predicate Control Problem to the following class of global predicates:

$$B_{ind}(G) \ \equiv \ \forall \ distinct \ s,t \in G : \ \forall i : \ \neg(critical_i(s) \ \wedge \ critical_i(t))$$

Given a function $critical_i$ on $S$ and a computation $\rightarrow$ on $S$, we define an *$i$-critical section* in an analogous fashion to the critical sections that we defined before. Note that the definition allows independent critical sections on the same process to overlap. In particular the same set of states may correspond to two different critical sections (corresponding to a critical section with multiple locks). Let $\mapsto$ be a relation on all critical sections defined as before.

**Theorem 5 (Necessary Condition)**
*For a computation $\rightarrow$ of $S$, and a global predicate $B_{ind}$,*

| | | |
|---|---|---|
| *a controlling computation of $B_{ind}$ in $\rightarrow$ exists* | $\Rightarrow$ | *$\mapsto$ has no cycles of $i$-critical sections, for some $i$* |

**Proof:** The proof is almost identical to the proof of Theorem 1. □

**Theorem 6 (Sufficient Condition)** *For a computation $\rightarrow$ of $S$, and a global predicate $B_{ind}$,*

| | | |
|---|---|---|
| *$\mapsto$ has no cycles of $i$-critical sections, for some $i$* | $\Rightarrow$ | *a controlling computation of $B_{ind}$ in $\rightarrow$ exists* |

**Proof:** The proof is along similar lines to the proof of Theorem 4. In this case we take strongly connected components as before, but make use of the fact that no two $i$-critical sections may be in the same strongly connected component (otherwise, there would be a cycle of $i$-critical sections). □

### 3.4 Off-line Independent Read-Write Mutual Exclusion

Using similar definitions, the Off-line Independent Read-Write Mutual Exclusion Problem is a specialization of the Off-line Predicate Control Problem to the following class of global predicates:

$$B_{ind-rw}(G) \equiv \forall \; distinct \; s,t \in G : \; \forall i : \; \neg(write_critical_i(s) \; \wedge \; critical_i(t))$$

As before, we define *$i$-read critical sections* and *$i$-write critical section* ($1 \leq i \leq m$). Similarly, let $\mapsto$ be a relation on all critical sections. The necessary and sufficient condition is a combination of that of the previous two sections. Since the proofs are similar to the previous ones, we simply state:

**Theorem 7 (Necessary and Sufficient Condition)**
*For a computation $\rightarrow$ of $S$, and a global predicate $B_{ind-rw}$,*

| | | |
|---|---|---|
| *a controlling computation of $B_{ind-rw}$ in $\rightarrow$ exists* | $\equiv$ | *all cycles of $i$-critical sections in $\mapsto$ contain only read critical sections* |

### 3.5 Algorithm

Figure 3 shows the algorithm to find a controlling computation of $B_{ind-rw}$ in $\rightarrow$. Since the other forms of mutual exclusion are special cases of $B_{ind-rw}$, this algorithm can be applied to any of them.

The input to the algorithm is the computation, represented by $n$ lists of critical sections $C_1, \cdots, C_n$. For now, to simplify presentation, we assume that critical sections are totally ordered on each process. Each critical section is represented as its process id, its first and last states, a type identifier *cs_id* that specifies the $critical_{cs_id}$ function, and a flag indicating if it is a write or read critical section. The partial order is implicitly maintained by vector clocks [9] associated with the first and last states of each critical section. The algorithm outputs the $\rightarrow^c$ relation specified as a list of ordered pairs of states.

The first while loop of the algorithm builds *ordered*, a totally ordered set of strongly connected components of critical sections (called scc's from here on). The second while loop simply uses *ordered* to construct the $\rightarrow^c$ relation.

```
Types:   state:                 (pid: int; v: vector_clock);
         critical_section:      (pid: int; first: state; last: state;
                                 cs_id: integer; write_critical: boolean);
         strongly_conn_component:  set of critical_section;
Input:   C_1, C_2, ..., C_n:    list of critical_section
Output:  O:                     list of (state, state), initially null
Vars:    scc_set, crossable:    set of strongly_conn_component
         crossed, prev, curr:   strongly_conn_component
         cs, cs':               critical_section
         ordered:               list of strongly_conn_component

 while (∀i : C_i ≠ null) do
     scc_set := get_scc(C_1.head, C_2.head, ..., C_n.head)
     crossable := { s ∈ scc_set | ∀s' ∈ scc_set, s' ≠ s : s' ↪̸ s }
     crossed := select(crossable);
     if (not_valid(crossed)) then
         exit("No Controlled Computation Exists");
     for each cs in crossed do
         C_{cs.pid}.delete_head();
     ordered.add_head(crossed);
 prev := ordered.delete_head();
 while (ordered ≠ null) do
     curr := ordered.delete_head();
     for each cs in prev and cs' in curr do
         if (cs.last ↛ cs'.first) then
         O.add_head(cs.last, cs'.first);
```

**Fig. 3.** Algorithm for Off-line Independent Read-Write Mutual Exclusion

The goal of each iteration of the first while loop is to add an scc, which is minimal w.r.t. $\hookrightarrow$, to *ordered* (where $\hookrightarrow$ is the relation on scc's defined in the proof of Theorem 4). To determine this scc, it first computes the set of scc's among the leading critical sections in $C_1, \cdots C_n$. Since no scc can contain two critical sections from the same process, it is sufficient to consider only the leading critical sections. From the set of scc's, it determines the set of minimal scc's, *crossable*. It then randomly selects one of the minimal scc's. Finally, before adding the scc to *ordered*, it must check if the scc is *not_valid*, where $not_valid(crossed) \equiv \exists cs, cs' \in crossed : \ cs.cs_id = cs'.cs_id \ \wedge \ cs.write_critical$. If an invalid scc is found, no controlling computation exists (by Theorem 7).

The main while loop of the algorithm executes $p$ times in the worst case, where $p$ is the number of critical sections in the computation. Each iteration takes $O(n^2)$, since it must compute the scc's. Thus, a simple implementation of the algorithm will have a time complexity of $O(n^2p)$. However, a better implementation of the algorithm would amortize the cost of computing scc's over multiple iterations of the loop. Each iteration would compare each of the critical sections that have newly reached the heads of the lists with the existing scc's, thus forming new scc's. Therefore, each of the $p$ critical section reaches the head

of the list just once, when it is compared with $n-1$ critical sections to determine the new scc's. The time complexity of the algorithm with this improved implementation is, therefore, $O(np)$. Note that a naive algorithm based directly on the constructive proof of the sufficient condition in Theorem 7 would take $O(p^2)$. We have reduced the complexity significantly by using the fact that the critical sections in a process are totally ordered.

The algorithm has implicitly assumed a total ordering of critical sections in each process. However, as noted before, independent critical sections on the same process may overlap, and may even coincide exactly (a critical section with multiple locks is treated as multiple critical sections that completely overlap). The algorithm can be extended to handle such cases by first determining the scc's within a process. These scc's correspond to maximal sets of overlapping critical sections. The input to the algorithm would consist of $n$ lists of such process-local scc's. The remainder of the algorithm remains unchanged.

## 4 Application to Software Fault Tolerance

Our proposed scheme for software fault tolerance consists of four parts: (i) tracing an execution, (ii) detecting a synchronization failure, (iii) determining a control strategy, and (iv) re-executing under control. This paper has focused mainly on the problem of determining a control strategy. We have designed an efficient algorithm that determines which synchronizations to add in order to avoid very general forms of mutual exclusion violation. As mentioned before, the other three parts of our scheme have been addressed as independent problems. We now put all the pieces together for a comprehensive look at how race failures (mutual exclusion violations) can be tolerated.

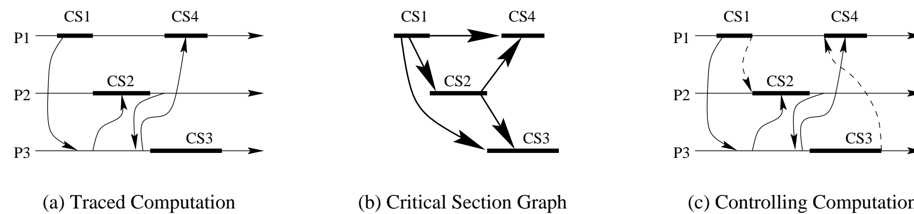


**Fig. 4.** Example: Tolerating Races in a Concurrent Execution

The problem of determining a control strategy was placed in a very general model of concurrent execution. However, tracing, detection, and controlled re-execution depend greatly on the particular concurrent paradigm. We choose a simple example that demonstrates the key issues that will arise in most concurrent paradigms. Consider a distributed system of processes that write to a single shared file. The file system itself does not synchronize accesses and so the processes are responsible for synchronizing their accesses to the file. If they do not

do so, the writes may interleave and the data may get corrupted. Since the file data is very crucial, we must ensure that races can be tolerated. Synchronization occurs through the use of explicit message passing between the processes.

The first part of our mechanism involves tracing the execution. The concern during tracing is to reduce the space and time overhead, so that tolerating a possible fault does not come at too great a cost. Much work has been done in implementing tracing in various paradigms, while keeping the overhead low [2, 11, 12, 15]. In our example, we use a vector clock mechanism [9], updating the vector clock at each send and receive point. This vector clock needs to be logged for each of the writes to the file (for our algorithm). The vector clock values must also be logged for each receive point (for replay). When a write is initiated, and when it returns, the vector clock must be logged. In our example, the writes are typically very long and therefore are performed asynchronously. Thus, execution continues while the write is in progress. In particular, the process may receive a message from another process during its write to the file. Inserting some computation at the send, receive, write initiation, and write completion points can be achieved either by code instrumentation, or by modifying the run-time environment (message-passing interface and the file system interface).

The second part of our mechanism is detecting when a race occurs. Many existing tools have been built to solve exactly this problem [4, 10]. Since we use message passing as our synchronization mechanism, the methods described in [10] are particularly applicable.

Once a race has been detected, we roll-back all processes to a consistent global state prior to the race. We also roll-back the file to a version consistent with the rolled-back state of the processes. (We assume a versioned file system with the ability to roll back.) We then take the section of the traced vector clock values that occur after the rolled-back state. These indicate the critical section entry and exit points required by our algorithm. The algorithm would take $O(np)$ time, where $n$ is the number of processes and $p$ is the number of critical sections that have been rolled back. The output of the algorithm is the set of added synchronizations specified as pairs of critical section boundary points. Figure 4 demonstrates a possible scenario. Here the semantics of mutual exclusion correspond to a single exclusive lock. Therefore, the necessary and sufficient condition is that there are no cycles in the critical section graph shown in Figure 4(b). Applying the algorithm would add synchronizations to give the controlling computation shown in Figure 4(c).

The next step is to replay the processes using the logged vector clock values of the receive points. Each receive point must be blocked until the same message arrives as in the previous execution. This is a standard replay mechanism (e.g. [12]). In addition to this replay, we must impose additional synchronizations. For example, suppose $(s, t)$ is one of the synchronizations output by our algorithm. The state $s$ is a critical section exit point while $t$ is a critical section entry point. Each of these additional synchronizations is implemented by a control message sent from $s$ and received before $t$. Thus, at each critical section exit point, we must check the added synchronizations to decide if a control message must be

sent. At each critical section entry point, we must check the added synchronizations to decide if the process must block waiting for a control message. As in tracing, the points at which computation must be added are the write initiation and completion points, and the send and receive points. Again, we can accomplish this by code instrumentation or run-time environment modification.

We have chosen an example in which the processes only write to the file. If the processes were to read from the file as well, then that would cause causal dependencies between processes. Then we would have to track these causal dependencies as we did for messages. Another option would be to assume that these causal dependencies do not affect the message communications, in which case, we do not have to track them. However, if we take this approach, we would have to check to see that our traced computation is the same as the one being replayed. In case of a divergence, we would leave the execution to proceed uncontrolled from the point of divergence.

## 5 Concluding Remarks

We have presented an approach for tolerating synchronization faults in concurrent programs based on rollback and controlled re-execution. Our focus in this paper has been on races, which form a particular type of synchronization fault. In order to determine a control strategy that avoids races while re-executing, we have solved the off-line predicate control problem for various forms of mutual exclusion properties. We have determined the necessary and sufficient conditions for solving off-line predicate control for simple mutual exlusion, read-write mutual exclusion, independent mutual exclusion, and independent read-write mutual exclusion. We have presented an efficient algorithm that solves for the most general property, independent read-write mutual exclusion. The algorithm takes $O(np)$ time, where $n$ is the number of processes and $p$ is the number of critical sections. Finally, we have demonstrated how races can be tolerated using our algorithm. An implementation of software fault tolerance using controlled re-execution is currently being developed in order to evaluate the performance and effectiveness of the technique in practice.

It may be argued that mutual exclusion could be simply handled at the programming language level using locks (in other words, on-line mutual exclusion, as opposed to off-line mutual exclusion). However, there are good reasons for our approach. Firstly, as noted in Section 1, it is impossible to ensure that there will be no deadlocks with on-line locking unless some assumptions are made, such as non-blocking critical sections. In off-line mutual exclusion, no such assumptions are required. Secondly, programmers make mistakes, being prone to reduce locking for greater efficiency. Thirdly, source code is often unavailable for modification, while requirements change dynamically. In modern component-based systems, different components may come from different vendors and it may be difficult to ensure a consistent locking discipline throughout the code. The best approach is to use both good programming discipline and a sofware fault tolerance technique to make programs more resistant to failures.

## References


1. A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. of the First IEEE-CS International Conference on Computer Software and Applications*, pages 149 – 155, November 1977.
2. J. D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *2nd SIGMETRICS Symp. on Parallel and Distr. Tools*, pages 48 – 59, Aug. 1998.
3. F. Cristian. Understanding fault-tolerant distributed systems. *CACM*, 34(2):56 – 78, Feb 1991.
4. M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proc. of 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 22–25, Newport, USA, June 1997.
5. Y. Huang and C. Kintala. Software implemented fault tolerance: technologies and experience. In *Proc. IEEE Fault-Tolerant Comp. Symp.*, pages 138 – 144, June 1993.
6. R. K. Iyer and I. Lee. Software fault tolerance in computer operating systems. In M. R. Lyu, editor, *Software Fault Tolerance*, Trends in Software Series, chapter 11, pages 249 – 278. John Wiley & Sons, Inc., 1995.
7. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558 – 565, July 1978.
8. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*, chapter 3.1.2. The Java Series. Addison Wesley Longman, Inc., 1997.
9. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215 – 226. Elsevier Science Publishers B. V. (North Holland), 1989.
10. R. H. B. Netzer. *Race condition detection for debugging shared-memory parallel programs.* PhD thesis, University of Wisconsin-Madison, 1991.
11. R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1 – 11, May 1993. Also available as ACM SIGPLAN Notices Vol. 28, No. 12.
12. R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, pages 502 – 511, November 1992.
13. B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 1(2):220 – 232, June 1975.
14. M. Raynal. *Algorithms for mutual exclusion.* MIT Press, 1986.
15. M. Ronnse and W. Zwaenepoel. Execution replay for treadmarks. In *Proc. of the 5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'97)*, pages 343–350, January 1997.
16. A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proc. of the 9th Symposium on Parallel and Distributed Processing*, Orlando, USA, April 1998. IEEE.
17. Y. M. Wang, Y. Huang, W. K. Fuchs, C. Kintala, , and G. Suri. Progressive retry for software failure recovery in message-passing applications. *IEEE Trans. on Computers*, 46(10):1137–1141, October 1997.