Stepwise Design of Tolerances in Barrier Computations

Sandeep S. Kulkarni *Anish Arora*

Department of Computer and Information Science [1]
The Ohio State University
Columbus, Ohio 43210 USA

Abstract

We design in a stepwise manner a multitolerant computation for synchronizing the phases of concurrent processes. The tolerances of the computation enable processes to (i) compute all phases correctly in the presence of faults that corrupt process state in a detectable manner, and (ii) compute only a minimum possible number of phases incorrectly before resuming correct computation in the presence of faults that corrupt process state in an undetectable manner.

Keywords: multitolerance; detectable and undetectable faults; stepwise design; concurrent systems.

[1] Email: {kulkarni,anish}@cis.ohio-state.edu; Web: http://www.cis.ohio-state.edu/{~kulkarni,~anish }.
Research supported in part by NSF Grant CCR-93-08640 and OSU Grant 221506

1 The Problem

Given is an undirected, connected network of processes. Each process consists of a cyclic sequence of $n+1$ terminating phases, [$phase.0$; $phase.1$; ... ; $phase.n$], and is initially ready to execute $phase.0$. A computation of the network of processes is correct iff each $phase.i$, $0 \leq i \leq n$, is executed correctly with respect to the following barrier synchronization specification:

- Safety: No process executes $phase.i$ until all processes are ready to execute $phase.i$, and
- Progress: Each processes is eventually ready to execute $phase.(i+1)$,

where $phase.(n+1) = phase.0$. Note that in a correct computation different processes do not execute concurrently in different phases.

Faults may occur at any time during execution of processes. The effect of a fault is to "corrupt" the state of some process, i.e., the state of that process prior to the fault is lost and replaced with some other value. Corruptions are of two types, "detectable" and "undetectable". If detectable, the corrupted state can be reset before it is accessed by the processes, whereas if undetectable, the corrupted state cannot be reset before it is accessed by the processes.

In the presence of a finite number of detectable faults, correct computation can be eventually resumed without executing any phase incorrectly, by forcing all processes to re-execute the current phase. In the presence of a finite number of undetectable faults, however, correct execution may not be eventually resumed without executing some phases incorrectly, since processes may execute concurrently in these different phases due to corruption of their state. This motivates the following problem:

> *Design for the network of processes a program that, in the presence of any finite number of faults, executes only a minimum possible number of phases incorrectly and eventually resumes correct computation.*

A few comments about the problem are in order. The problem requires not only that each possible corruption of state be tolerated —which would require only the property of stabilizing fault-tolerance [1]— but also that each corruption be tolerated in a manner that is proportional to the extent and type of that corruption. Thus, a solution to this problem offers multiple levels of fault-tolerance, a notion which we abbreviate as multitolerance [2].

The fault model is general, in that it collectively abstracts various classes of concrete faults. For instance, it readily abstracts faults whose effects are transient, such as hazards, corruption, loss, reordering, and duplication of messages; incorrect initializations; processor restarts; and performance and timing violations. It also abstracts faults whose effects are permanent or intermittent, provided auxiliary state variables are considered.

Finally, the process model is also general. In particular, it extends a variety of other process models, including asynchronous clock unison [3], phase synchronization [4], and atomic commitment [5] (see Section 6 for a detailed discussion). A significant amount of work has already addressed the fault-tolerance of each of these models. To the best of our knowledge, however, the existence and systematic design of multitolerant solutions for these models has not received much attention.

2 Structure of the Solution

For purposes of design, we divide the state of each process into three portions: *intraphase* state, *interphase* state, and *interprocess* state. Intraphase state comprises the state of the underlying execution (which we will deliberately leave unspecified) and the superposed state used to control the underlying execution within a phase. Interphase state comprises the superposed state used to control the underlying execution between different phases. Interprocess state comprises the superposed state used to coordinate the underlying execution with that of the remaining processes.

In accordance with this division, our design proceeds in a stepwise fashion: In Section 3, we design the program for the case where n (the highest phase number) is 1 and each process atomically (i.e., instantaneously) accesses the state of all other processes and updates its own state. For this case, only intraphase state is needed. The resulting computation recovers from detectable faults with no phase executed incorrectly and from undetectable faults with at most one phase executed incorrectly.

In Section 4, we extend the design for the case where $n > 1$ and each process atomically accesses the state of all other processes and updates its own state. For this case, only interphase state is needed in addition to the intraphase state introduced earlier. The resulting computation recovers from detectable faults with no phase executed incorrectly and from undetectable faults with the number of incorrectly executed phases no more than the number of phases that processes are left in.

In Section 5, we distribute the program resulting from the second step for the case where each process atomically accesses only the state of the neighboring processes in the network and updates its own state. For this case, interprocess state is needed in addition to the intraphase and interphase state introduced earlier. The resulting distributed computation recovers in the same manner as its non-distributed version.

In Section 6, we briefly discuss the extension for the case where each process atomically accesses either the state of at most one neighboring process or updates its own state (but not both).

Notation. We present our designs in terms of a guarded command notation. More precisely, each process consists of a finite set of variables and a finite set of actions. Each action consists of two parts: a guard and a statement. To simplify discussion about actions, a unique name is associated with each action. Thus, each action is of the following form:

$$\langle name \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$$

The guard is a boolean expression over the variables of that and possibly other processes, and the statement updates zero or more variables of that process. An action is executed only if it is enabled, i.e., its its guard evaluates to true. To execute the action, its statement is executed atomically.

A computation of the network of processes is a fair, infinite sequence of steps; in each step, an enabled action of some process is executed. Fairness of the sequence means that if an action of a process is continuously enabled then that action is eventually executed.

We represent each fault by an action. An undetectable fault assigns to variables of a process nondeterministically chosen values from their domains. A detectable fault assigns to variables of a process deterministically chosen values from their domains, that may depend on the values of the variables of other processes.

3 Step 1: One Phase Barrier Synchronization

In this section, we present the first step of our design, for the case where the sequence of phases is of length one. To synchronize the underlying computation, each process j maintains some intraphase control state, which we represent by the variable $cp.j$ (for control position of j).

Clearly, there exists a $cp.j$ value, say *execute*, which denotes that j is executing its phase, and a value, say *ready*, which denotes that j is ready to execute its phase. We consider two additional control position values: *success*, which denotes that j has completed its phase, and *error*, which denotes that the control position of j is detectably corrupted.

The state transitions between the control position values of j are shown in Figure 1: Figure 1(a) shows the transitions in the absence of faults, and Figure 1(b) shows the transitions in the presence of detectable corruption of control position. We describe the conditions under which j executes these transitions next.

A transition from *ready* to *execute* occurs only after some process checks that all processes are in control position *ready*. After the first process changes its control position thus from *ready* to *execute*, the remaining processes can change their control position from *ready* to *execute* without checking the state of all processes; they need only check that some process is in control position *execute*.

The transition from *execute* to *success* occurs only after all processes start execution of their phases. This restriction is motivated by the following scenario: Consider a network of two processes, say j and k, both in the control position *ready*. Let j change its control position to *execute*, execute its phase, and then change its control position to *success*, thereby yielding a state where the control position of j is *success* and that of k is *ready*. The same state is also reached from the initial state if both j and k execute their respective phases, change their control position to *success*, and then k changes its control position to *ready*. Thus, if j changes its control position to *success* without this restriction, j cannot determine whether k has executed its phase or not.

The transition from *success* to *ready* occurs only when no process is in the control position *execute*. This restriction prevents the network from remaining forever in states where the control positions *ready*, *execute*, and *success* coexist (the network may reach such a state due to undetectable faults). As we prove later in this section, this restriction also enables the network to recover from such a state to a state where the control position of all processes is *ready*. Starting from the latter state, further computation satisfies the specification of barrier synchronization.

The transition from *success* to *error* occurs only when a process in the control position *success* checks that some process is in the control position *error*. This restriction ensures that each process becomes aware of any detectable corruptions.

The transition from *error* to *ready* occurs only when no process is in the control position *execute*. This restriction ensures that if any detectable corruption occurs during execution of a phase, re-execution of that phase does not begin until all processes have completed that phase.

Our one phase barrier synchronization program, OB, consists of five actions, one for each transition discussed above (see Figure 2).

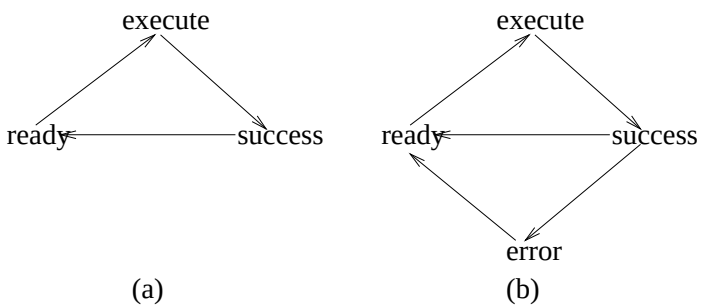

Figure 1: **State transitions for one phase barrier synchronization**

Program

$OB1 ::$	$cp.j = ready \ \wedge \ ((\forall k :: cp.k = ready) \ \vee \ (\exists k :: cp.k = execute))$	$\longrightarrow$	$cp.j := execute$
$OB2 ::$	$cp.j = execute \ \wedge \ ((\forall k :: cp.k \neq ready) \ \vee \ (\exists k :: cp.k = success))$	$\longrightarrow$	$cp.j := success$
$OB3 ::$	$cp.j = success \ \wedge \ (\forall k :: cp.k \neq execute)$	$\longrightarrow$	$cp.j := ready$
$OB4 ::$	$cp.j = success \ \wedge \ (\exists k :: cp.k = error)$	$\longrightarrow$	$cp.j := error$
$OB5 ::$	$cp.j = error \ \wedge \ (\forall k :: cp.k \neq execute)$	$\longrightarrow$	$cp.j := ready$

Detectable corruption of control position

$D-Corr ::$ true $\longrightarrow$ $cp.j := error$

Undetectable corruption of control position

$U-Corr ::$ true $\longrightarrow$ $cp.j := ?$, where ? is any value from the domain (of $cp.j$)

Figure 2: **Program OB and the faults**

Proof of correctness. In the absence of faults, the first process to change its control position from *ready* (*execute*) to *execute* (*success*) checks that all processes are in control position *ready* (*execute*). Also, a process changes its control position from *success* to *ready* only when no process is in control position *execute*. Thus, the control positions *ready*, *execute*, and *success* cannot all coexist, i.e., the state predicate inv_1 is invariantly true in program OB, where

$$inv_1 \;=\; \neg(\exists j,k,l :: cp.j = ready \;\wedge\; cp.k = execute \;\wedge\; cp.l = success)$$

Safety. In the absence of faults, as noted above, when a process changes its control position from *success* to *ready*, no process is in the control position *execute*, i.e., all processes have completed the previous phase. To execute the next phase, a process changes its control position from *ready* to *execute*. Hence, no process starts execution of a phase until all processes are ready to execute the phase.

In the presence of detectable faults, a process that is subject to state corruption changes its control position to *error*. Thus, inv_1 continues to be true. Since the first process to re-execute a phase checks that all processes are in control position *ready* and since a process changes its control position from *error* to *ready* only when no process is in control position *execute*, it follows that a phase re-execution does not overlap with its previous execution.

Progress. In the absence of faults, once the control position of all processes is *ready*, each process eventually changes its control position to *execute*. A process changes its control position to *success* only after completing its phase. As long as some process is in control position *execute*, no process can change its control position from *success* to *ready*. It follows that eventually OB reaches a state where all processes are in control position *success*. By a similar argument, starting from such a state, OB reaches a state where all processes are in control position *ready*.

In the presence of detectable faults, inv_1 continues to hold. In inv_1 states, at least one action of program OB is enabled. Now consider the variant function: $3 * |\{j : cp.j = ready\}| \;+\; 2 * |\{j : cp.j = execute\}| \;+\; |\{j : cp.j = success\}|$. Execution of actions $OB1$, $OB2$, and $OB4$ decreases the value of the function whereas execution of actions $OB3$ and $OB5$ increases the value of the function. Since the value of the function is non-negative, it follows that eventually action $OB3$ or $OB5$ is executed, and in the resulting state no process is in control position *execute*. Starting from such a state, each process changes its control position to *ready*, and a state is reached where the control position of all processes is *ready*.

In the presence of undetectable faults, OB may reach a state where inv_1 is violated. In such a state, actions $OB1$ and $OB2$ are always enabled at some process whereas actions $OB3$ and $OB5$ are disabled at all processes. Consider again the variant function $3 * |\{j : cp.j = ready\}| \;+\; 2 * |\{j : cp.j = execute\}| \;+\; |\{j : cp.j = success\}|$. By the same argument as before, eventually $OB3$ or $OB5$ is executed. Since $OB3$ and $OB5$ execute only in a state where inv_1 holds, it follows that eventually OB reaches a state where inv_1 is satisfied and correct computation resumes.

Since a process cannot execute the transition from *success* to *ready* in a state where inv_1 does not hold, and since all processes change their control position from *success* to *ready* between execution of two phases, it follows that at most one phase may be executed incorrectly before inv_1 is satisfied.

4 Step 2: Multiphase Barrier Synchronization

In this section, we present the second step of our design, for the case where the cyclic sequence consists of multiple phases. To distinguish between the phases, each process maintains some interphase state, which we represent here by the variable *ph*, whose value denotes the number of the phase the process is currently in.

In the absence of undetectable corruption of state, the state transitions of multiphase barrier synchronization are as shown in Figure 3. Figure 3(a) shows the state transition diagram obtained by simple repetition of the state transitions of one phase barrier synchronization. A process increments its phase when it changes its control position from *success* to *ready*. As shown in Figure 3(b), the control position *success* of *phase.i* and the control position *ready* of *phase.*$(i+1)$ can be merged. When a process completes its phase, it increments its phase and changes its control position to *ready*. We describe the conditions under which these transitions occur next.

The transition from *ready* to *execute* is a modified version of action $OB1$: it first occurs in a phase when some process checks that all processes are in the same phase and their control position is *ready*. The processes that follow change their control position to *execute* without checking the state of all processes: they need only find some process in the current phase and in the control position *execute*.

The transition from *execute* to *ready* is a modified version of actions $OB2$ and $OB3$: it occurs when all processes are in the current phase and their control position is not *ready* or in the next phase and their control position is *ready*.

The transition from *ready* to *error* is a modified version of action $OB4$: it occurs when a process in control position *ready* checks that some process is in the control position *error* of the previous phase.

The transition from *error* to *ready* is a modified version of action $OB5$: it occurs when all processes are in one phase and no process is in the control position *execute*.

Detectable corruption. If the phase or control position of j is corrupted detectably, j obtains the number of the current phase from other processes. If there is another process whose control position is either *execute* or *error*, j copies the phase of that process; else, j copies the phase of any process.

Undetectable corruption. If the phase or control position of j is corrupted undetectably, the program recovers to a state where all processes are in the same phase; starting from such a state, correct computation resumes. To this end, a transition across phases is added. This transition occurs at a process j when either the distance between the phases of j and some other process k is more than one or both j and k are executing different phases. In either case, j copies the minimum phase number existing in the network and sets its control position to *error*.

The multiphase barrier synchronization program, MB, consists of five actions, one for each transition discussed above (see Figure 4).

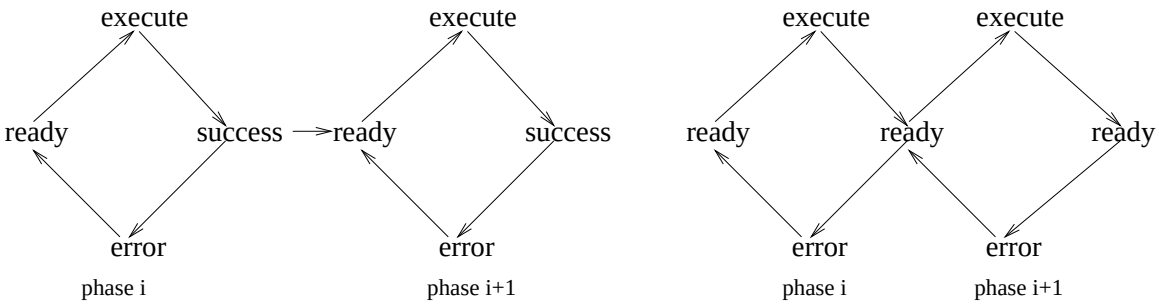

Figure 3: **State transitions for multiphase barrier synchronization**

Program

$MB1 ::$ $cp.j = ready \ \wedge \ ((\forall k :: ph.j = ph.k \ \wedge \ cp.k = ready) \vee (\exists k :: ph.j = ph.k \ \wedge \ cp.k = execute)) \longrightarrow cp.j := execute$

$MB2 ::$ $cp.j = execute \ \wedge \ (\forall k :: (ph.j = ph.k \ \wedge \ cp.k \neq ready) \vee ((ph.j+1) = ph.k \ \wedge \ cp.k = ready)) \longrightarrow ph.j, cp.j := (ph.j+1), ready$

$MB3 ::$ $cp.j = ready \ \wedge \ (\exists k :: (ph.j-1) = ph.k \ \wedge \ cp.k = error) \longrightarrow ph.j, cp.j := (ph.j-1), error$

$MB4 ::$ $cp.j = error \ \wedge \ (\forall k :: ph.j = ph.k \ \wedge \ cp.k \neq execute) \longrightarrow cp.j := ready$

$MB5 ::$ $(\exists k :: ph.j > ph.k \wedge$ $\longrightarrow ph.j, cp.j := (min \ k :: ph.k), error$
$((ph.j+1) \neq ph.k \ \wedge \ (ph.j-1) \neq ph.k)$
$\vee (cp.j = ready \ \equiv \ cp.k = ready)$
$\vee ((ph.k+1) = ph.j \ \wedge \ cp.j \neq ready \ \wedge \ (ph.k-1) \neq ph.j)))$

Detectable corruption

$D-Corr ::$ true $\longrightarrow$ if (there exists k such that $cp.k \neq ready$)
then $ph.j := ph.k$ for this k
else $ph.j := ph.k$ for any process k;
$cp.j := error$

Undetectable corruption

$U-Corr ::$ true $\longrightarrow$ $ph.j, cp.j := ?, ?$

Figure 4: **Program MB and its faults**

Proof of correctness. In the absence of faults, the phases of processes can differ by at most one. Let j and k be processes whose phases are i and $i+1$ respectively. Such a state occurs when k has completed *phase.i* and moved to *phase.*$(i+1)$. It follows that the control positions of k and j are *ready* and not *ready* respectively. Thus, if j and k are in different phases, exactly one of them is in control position *ready*.

Note that if the sequence of phases is of length 2 and the phase of k is one greater than that of j, then the phase of j is also one greater than that of k. In this case, either process could be in the next phase; thus, either process could be in control position *ready*. However, if there are at least three phases and the phase of k is one greater than that of j, then the control position of k is *ready*. Hence, the state predicate inv_2, $inv_2 = (\forall j,k :: S1.j.k \ \wedge \ S2.j.k \ \wedge \ S3.j.k)$, is invariantly true in program MB, where

$$
\begin{array}{lcl}
S1.j.k & = & (ph.j = ph.k \ \vee \ (ph.j+1) = ph.k \ \vee \ (ph.j-1) = ph.k) \\
S2.j.k & = & (ph.j \neq ph.k \ \ \Rightarrow \ \ (cp.j = ready \equiv cp.k = ready)) \\
S3.j.k & = & ((ph.j+1) = ph.k \ \ \Rightarrow \ \ (cp.k = ready \ \ \vee \ \ (ph.j-1) = ph.k))
\end{array}
$$

Safety. In the absence of faults, when a process increments its phase from i to $i+1$, all processes have started execution of *phase.i*, i.e., a process cannot execute *phase.*$(i+1)$ unless all processes start execution of *phase.i*. From $S2.j.k$, execution of *phase.*$(i+1)$ cannot overlap with the execution of *phase.i*. It follows that execution of *phase.*$(i+1)$ begins only after all processes are ready to execute *phase.i*.

In the presence of detectable faults, a process that is subject to state corruption changes its control position to *error*. Since the first process to execute a new instance of a phase checks that all processes are in control position *ready* and since a process changes its control position from *error* to *ready* only when no process is in control position *execute*, it follows that the new instance of the phase does not overlap with the previous instance.

Progress. In the absence of faults, once the control position of all processes is *ready* in a phase, say i, each process eventually changes its control position to *execute*. After completing *phase.i*, each process increments its phase to $i+1$ and changes its control position to *ready*.

In the presence of detectable faults, a process that is subject to state corruption restores its phase, say to i. Since inv_2 continues to be satisfied in the presence of detectable faults, it follows that the phase of the remaining processes is either i or $i+1$ and, if it is $i+1$, the control position is *ready*. In inv_2 states, $MB5$ is disabled and at least one of the remaining actions is enabled. Now consider the variant function $3 * |\{j : ph.j = i \ \wedge \ cp.j = ready\}| \ + \ 2 * |\{j : ph.j = i \ \wedge \ cp.j = execute\}| \ + \ |\{j : ph.j = i+1 \ \wedge \ cp.j = ready\}|$. Actions $MB1$, $MB2$, and $MB3$ decrease the value of this function, while action $MB4$ increases the value. Since the value of the function is non-negative, eventually $MB4$ is executed, i.e., all processes are in phase i and in control position *ready* or *error*. Starting from such a state, each process changes its control position to *ready*.

In the presence of undetectable faults, MB may reach a state where inv_2 is violated. In $\neg inv_2$ states, at least one action ($MB5$) is always enabled. Now, consider the variant function $(n+2) * |\{j : cp.j = ready\}| \ + \ 2 * |\{j : cp.j = execute\}| \ + \ (\sum j :: ph.j)$. Execution of $MB1$, $MB3$, and $MB5$ decreases the value of the function, and execution of $MB2$ and $MB4$ increases the value of the function. Since

the value of the function is non-negative, it follows that eventually $MB2$ or $MB4$ is executed. Since $MB2$ and $MB4$ are executed only when inv_2 holds, it follows that eventually inv_2 holds.

If the network of processes is perturbed to a state where processes are in m distinct phases, each of these m phases may execute incorrectly (with respect to the barrier synchronization specification). But correct execution resumes before any more phases execute incorrectly. To see this, observe that no process can execute action $MB2$ in a state where inv_2 is violated, i.e., no process enters a new phase until inv_2 is satisfied. Thus, at most m phases are executed incorrectly. Also, of the m phases, MB correctly re-executes the one with smallest number.

5 Step 3: Distributed Multiphase Barrier Synchronization

In this section, we present the third step of our design, for the case where the network of processes is connected but not fully and, therefore, a process cannot unilaterally access the state of all processes. Since processes now have to cooperate with each other to detect the state of all processes, we propose to exploit the connectivity of the network as follows: A fixed, rooted spanning tree is embedded in the network (by fixed, we mean that the tree cannot be corrupted by faults). The root process uses its unique privilege to be the first process to begin execution of a phase and the last process to complete execution of a phase.

After the root process completes a phase, it starts execution of the next phase (i.e., changes its control position to *execute*). Each other process starts execution of this next phase only after its parent starts. Leaf processes are the first to complete the phase (i.e., to increment their phase and change their control position to *ready*). Each other process completes the phase only after all its children have completed the phase. It follows that when the root completes the phase, all processes have completed the phase.

We use a diffusing computation to implement the synchronization described in the previous paragraph. Upon starting execution of a phase, the root initiates a diffusing computation. The diffusing computation propagates from the root towards the leaves, and completes from the leaves towards the root.

If the state of a process is corrupted detectably, the process aborts its diffusing computation: to do so, it completes the diffusing computation with the control position *error*. The control position *error* is passed back towards the root during the completion of the diffusing computation. When the root enters the control position *error*, some processes may still be executing their phase. The root then initiates a new diffusing computation to restore the control position of all processes to *ready*, so that the phase can be safely re-executed.

Our distributed barrier synchronization program, DB, consists of three transitions: one each for initiation, propagation, and completion of diffusing computations (see Figure 5). Each process j maintains a diffusion state $st.j$, which is either *prop* or *comp*, denoting whether j is propagating a diffusing computation or has completed a diffusing computation; and a sequence number ($sn.j$), which is either 0 or 1, to distinguish between successive diffusing computations. We describe these transitions next.

The initiation transition occurs when the root has completed a diffusing computation. If the diffusing

Program

$DB1$ :: j is root $\wedge\ st.j = comp$ $\longrightarrow$ $st.j, sn.j = prop, sn.j+1$;
if $cp.j = ready$ then $cp.j := execute$ else $cp.j := ready$

$DB2$ :: $st.(P.j) = prop\ \wedge\ sn.j \neq sn.(P.j)$ $\longrightarrow$ $st.j, sn.j := st.(P.j), sn.(P.j)$;
$ph.j, cp.j := ph.(P.j), cp.(P.j)$

$DB3$ :: $st.j = prop \wedge$
$(\forall l : l \in ch.j : st.l = comp\ \wedge\ sn.j = sn.l)$ $\longrightarrow$ $st.j := comp$;
if $(cp.j = ready)$ then
if $(\exists l : l \in ch.j : cp.l \neq ready\ \vee\ ph.j \neq ph.l)$ then $cp.j := error$
else
if $(\forall l : l \in ch.j : ph.l = (ph.j+1)\ \wedge\ cp.l = ready))$ then
$ph.j, cp.j := (ph.j+1), ready$;
else $cp.j := error$

Detectable corruption

$D-Corr$ :: true $\longrightarrow$ if (there exists a neighbor k such that $cp.k \neq ready$)
then $ph.j = ph.k$ for this k
else $ph.j = ph.k$ for any k;
$cp.j, st.j, sn.j := error, comp, sn.(P.j)$

Undetectable corruption

$U-Corr$ :: true $\longrightarrow$ $ph.j, cp.j, st.j, sn.j := ?, ?, ?, ?$

Figure 5: **Program DB and its faults**

computation was completed successfully (i.e., the root is in the control position *ready*), the root changes its control position to *execute*; else (i.e., the root is in the control position *error*), the root changes its control position to *ready*.

The propagation transition occurs when the parent of a process j (which we denote by $P.j$) is propagating a diffusing computation. Upon propagation, j copies the phase and the control position of its parent. Thus, if a diffusing computation is initiated when the root is in the control position *execute* (*ready*), each process eventually changes its control position to *execute* (*ready*).

The completion transition occurs when all children of a process have completed their diffusing computation. If the control position of j is *ready*, j checkes that all children of j are in the same phase and in control poistion *ready*; if not j changes its control position to *error*. If the control position of j is *execute*, j checks that all children of j are in the next phase and their control position is ready; if so, j copies the phase and the control position of its children. Else, j changes its control position to *error*.

Detectable corruption of the phase, control position and diffusing computation variables. If a detectable state corruption occurs at process j, j completes its diffusing computation, changes its control position to *error*, and copies the sequence number from its parent. If j has a neighbor that is in the control position *execute* or *error*, j copies the phase of that process; otherwise, j copies the phase of any neighbor.

Undetectable corruption of the phase, control position and diffusing computation variables. As we prove later in this section, the transitions described above suffice to ensure recovery from undetectable corruption, i.e., starting from an arbitrary state, the program eventually satisfies the specification of barrier synchronization.

Proof of correctness (sketch). *Safety.* In the absence of faults, when the root increments its phase from i to $i+1$, all processes have completed phase i. Since the root process is the first process to execute phase $i+1$, safety follows.

In the presence of detectable faults, a process that is subject to state corruption completes its diffusing computation in the control position *error*. In the completion action, a process changes its control position to *ready* only when none of its descendents are in control position *error* or *execute*. It follows that when the root completes its diffusing computation in the control position *ready*, no process is in control position *error* or *execute*. Thus, when the root re-executes a phase by changing its control position from *ready* to *execute*, no process is executing the previous instance of that phase.

Progress. In the absence of faults, once the control position of all processes is *ready* in a phase, say i, the root process changes its control position to *execute*. Eventually, all processes change their control position to *execute*. After completing *phase.i*, each leaf process increments its phase and changes its control position to *ready*. Eventually, all processes increment their phase and change their control position to *ready*.

In the presence of detectable faults, a process subject to state corruption completes any diffusing computation that it may be propagating in the control position *error*. Eventually, all ancestors of this process complete the diffusing computation in control position *error*. When the root completes thus, it initiates a new diffusing computation that changes the control position of all processes to

ready. Subsequently, correct computation resumes.

In the presence of undetectable faults, regardless of the state of the processes, at least one action can always execute. Also, if the initiation action is not executed, propagation and completion actions can execute only a finite number of times. Hence, an initiation action executes eventually. In the diffusing computation that follows, the children of the root 'synchronize' with the root since each process copies the phase and control position of its parent in a propagation action. In the next diffusing computation, the children of these processes also synchronize with the root. It follows, by induction on the height h of the spanning tree, that all processes synchronize with the root within h diffusing computations. Subsequently, correct computation resumes.

If the network of processes is perturbed to a state where processes are in m distinct phases, at most these m phases are executed incorrectly before correct computation resumes. To see this, observe that if a new phase number, say i, is introduced into the network it must be at a leaf process that is in control position *ready*. Subsequently, no process executes *phase.i* until the root starts execution of *phase.i*. The root executes *phase.i* only if it is in the control position *ready* and, hence, all its descendents are in the control position *ready* in *phase.i*. Thus, *phase.i* executes correctly.

6 Discussion

In this section, we discuss how instantiations of barrier synchronization satisfy the typical tolerance requirements of asynchronous unison, phase synchronization and atomic commitment. We also discuss how to implement programs OB, MB, and DB so that each process action either atomically reads the state of at most one neighboring process or updates its own state (but not both).

6.1 Related Work

Instantiations of our multitolerant solution satisfy the specification and typical tolerance requirements of a variety of synchronization problems, such as asynchronous clock unison, phase synchronization, and atomic commitment. We briefly recall these problems below.

Asynchronous unison. In the asynchronous unison problem, every process maintains a bounded-value clock such that, at all times, the clock differs from the clock of each other process by no more than one and that, infinitely often, the clock is incremented.

Traditionally, the faults considered in the asynchronous unison problem corrupt the clocks of processes undetectably. The desired tolerance to these faults is that upon starting from a state where the clocks have arbitrary values, the program should eventually reach a state where the clocks resume correct computation.

The barrier synchronization problem generalizes the asynchronous unison problem, in the sense that phase i of the computation may be mapped onto the i-th value of the clock (chosen from its bounded domain). Note that in the absence of undetectable faults, the phases of all processes in the barrier synchronization differ from each other by at most one. Since our solution is stabilizing tolerant in the presence of undetectable faults, it meets the tolerance requirements of asynchronous unison.

Phase Synchronization. In the phase synchronization problem, each process executes a (potentially

infinite) sequence of phases. A process executes a phase only when all processes have completed the previous phase.

Traditionally, the faults considered in the phase synchronization problem corrupt the phase of processes initially in (and not during) the computation. The desired tolerance to these faults is that each phase be executed correctly, without making any assumptions about the speed of other processes. This implies that no process should read any shared variable (such as the phase variable) unless that variable has been previously initialized.

The barrier synchronization problem generalizes the phase synchronization problem, in the sense that each phase in the latter can be uniquely mapped onto an instance of a phase in the former. Our solution tolerates the detectable corruption of variables without executing any phase incorrectly, but it assumes that each process corrects its detectably corrupted shared variables before any process action accesses these variables. Our solution can, however, be extended so that each process corrects the shared variables of all process, thereby meeting the tolerance requirements of phase synchronization.

Atomic commitment. In the atomic commit problem, each process casts one of two votes, Yes or No, and then reaches one of two decisions, Commit or Abort. A process reaches the decision to Commit iff all processes cast Yes votes. Otherwise, it reaches the decision to Abort. The decision reached by all processes must be identical.

The barrier synchronization problem generalizes the atomic commit problem, in the sense that each phase of the computation may be seen as an atomic commitment. We can let a process become *ready* in the next phase only if it executes its current phase with the vote Yes; else (if it votes No), we can "fail" the current phase.

Traditionally, the faults considered in the atomic commit problem make processes crash, restart, or exhibit Byzantine behavior. Although these faults have not explicitly been considered in this paper, as they seem to corrupt actions —as opposed to variables— in processes, it is possible to represent the corruption of actions by faults that corrupt the auxiliary variables by introducing auxiliary variables. For example, the crash of a process can be captured by introducing an auxiliary variable *up* for that process, as follows. Each action of that process is to be executed only if *up* is true. The crash itself is modeled as the occurrence of a fault that corrupts *up*, by setting it to false. Similarly, the Byzantine behavior of a process can be captured by introducing an auxiliary variable *good*, as follows: If the variable *good* is true, then the process executes its normal actions. When a fault action corrupts *good* to false, the process executes actions whose behavior is nondeterministic.

Our multitolerant solution can be readily extended to handle faults that effectively corrupt the actions of the program. Consider, for instance, fail-stop faults. To enable non-faulty processes to continue to satisfy the specification of barrier synchronization in the presence of fail-stop faults, we may ensure that the spanning tree is (re)constructed and that a root process completes a diffusing computation only after all processes participate in that diffusing computation. (The interested reader is referred to [6] for the details of this approach.) Also, using Schneider's approach [7] to implement processors that behave like fail-stop processors with high probability, our program can be extended to tolerate Byzantine faults with high probability.

6.2 Low Atomicity

In program OB, actions have high atomicity, in the sense that they instantaneously access the state of several processes. To reduce atomicity, we let each process j first read the state of all processes in some order, atomically reading only one process at a time. Then, we let j decides whether some action can be executed based on the values read; if there is one, j atomically executes that action, thereby updating its state. Below, we informally discuss (1) why this low atomicity program works correctly in the absence of faults; (2) the modifications required for this low atomicity program to work in the presence of faults; and (3) the corresponding modifications required in program MB and DB.

(1) In the absence of faults, when j reads the state of one process at a time, execution of $OB1$, $OB2$, and $OB3$ is acceptable. To see this, consider action $OB1$: j checks that the control position of each process k is *ready* before executing $OB1$; it is possible that some k may change their control position to *execute* before j executes $OB1$. But it is not possible that any of these k change their control position to *success* before j executes $OB1$, since j is still in the control position *ready*. A similar argument holds for $OB2$ and $OB3$ (we do not consider $OB4$ and $OB5$ as they are disabled in the absence of faults).

(2) In the presence of faults, the low atomicity program admits a scenario where control positions *ready*, *execute* and *success* coexist indefinitely in the network. To prevent this scenario, we make the following modifications: (a) the transition from *ready* to *execute* is restricted to occur only after a process checks that the control position of each process is *ready* or *execute*, and (b) when process j in control position *success* or *error* checks that $cp.k$ is *ready*, it also checks that k has correct value of $cp.j$. It follows that when j changes its control position from *success* or *error* to *ready*, no process is in the control position *execute*.

(3) We make analogous modifications in the low atomicity program for MB: (a) the transition from *ready* to *execute* is restricted to occur only after a process checks that the control position of each process is *ready* or *execute*, and (b) when a process j in control position *execute* or *error* checks that $cp.k$ is *ready*, it also checks that k has correct values of $ph.j$ and $cp.j$.

Program MB has one more complication, namely the high atomicity of the fault action. This raises two concerns: what happens if a program action attempts to read variables of a process that is executing a detectable fault action, and what happens if a detectable fault action attempts to read variables of another process that is executing its detectable fault action. To deal with the first concern, the program action is blocked until the corresponding fault action completes. To deal with the second concern, the detectable fault actions are ordered in such a way that the concurrent execution of the detectable fault actions is equivalent to some serial execution of the detectable fault actions. To this end, the detectable fault action at j blocks if it attempts to read variables of a process k that is executing its detectable fault action and has already checked the state of j. To avoid deadlocks, each process checks all processes in a fixed order. This ensures that the detectable fault actions of j and k are serialized.

In the low atomicity program for DB, if the state of process j is corrupted detectably, we let j check the state of its ancestors. More specifically, if the state of the parent of j is not corrupted, j copies the phase from its parent. Otherwise, j blocks until its parent corrects its state. It follows that if j has an ancestor whose state is not corrupted, j eventually corrects its state. If the state

of all ancestors of j is corrupted (in particular, if the state of the root process is corrupted), j will be blocked forever. To prevent this scenario, we ensure that the root obtains the current phase information from its descendents. To this end, the root initiates a special diffusing computation. A process propagates this diffusing computation to its children only if its state is detectably corrupted. A process completes this diffusing computation when the state of none of its children is corrupted. Finally, when the root completes this diffusing computation, it obtains the current phase number. After the root process completes its diffusing computation, j has an ancestor whose state is not corrupted. Hence, j eventually corrects its state.

Other properties of the solution. Programs OB, MB, and DB are masking tolerant to detectable faults [8], since they continue to satisfy the specification when detectable faults occur. They are nonmasking tolerant to undetectable faults, since they eventually resatisfy the specification when undetectable faults stop. They have low space-complexity. Optimal state is introduced for intraphase control (three values per process) and for interphase control ($n + 1$ values per process), and little state is introduced for interprocess control (four values per process).

In conclusion, our solution illustrates the stepwise design of tolerance properties. We have presented formal methods for the stepwise design of multitolerant programs elsewhere [2].

References

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[2] A. Arora and S. S. Kulkarni. Multitolerance. Technical Report OSU-CISRC TR37, Ohio State University, 1996.

[3] J. Couvreur, N. Francez, and M. Gouda. Asynchronous unison. *Proceedings of the Twelveth International Conference on Distributed Computing Systems, Tokyo*, 1992.

[4] J. Misra. Phase synchronization. *Information Processing Letters*, 38:101–105, April 1991.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 7. Addison Wesley, 1987.

[6] S. S. Kulkarni and A. Arora. Multitolerance in distributed reset. Technical Report OSU-CISRC TR13, Ohio State University, 1996. Submitted to *Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization*.

[7] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[8] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr*, 14:174–185, 1995. Submitted to *IEEE Transactions on Software Engineering*.