Multitolerant Barrier Synchronization

Sandeep S. Kulkarni *Anish Arora*

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

Abstract

We design a multitolerant program for synchronizing the phases of concurrent processes. The tolerances of the program enable processes to (i) execute all phases correctly in the presence of faults that corrupt process state in a detectable manner, and (ii) execute 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: fault-tolerance, detectable and undetectable faults, parallel processing, concurrency

1 Motivation

Barrier synchronization in its general form requires that a set of processes execute a cyclic sequence of phases so that a phase is executed by each process only after all processes have completed the previous phase. This form of synchronization generalizes a variety of others, such as clock unison [1], phase synchronization [2] and atomic commitment [3], and appears frequently in parallel, distributed, and scientific computation applications.

Often the design of barrier synchronization has to accommodate the occurrence of faults. Commonly considered examples of faults include incorrect initializations; corruption, loss, reordering, and duplication of messages; processor restarts; and performance and timing violations. While some of these fault-classes can be masked (i.e., even in their presence each phase is executed correctly), others cannot.

To accommodate multiple fault-classes, not all of which can be masked, it is convenient to view the effect of faults in each fault-class as a "corruption" of the state of some process, i.e., the state of that process prior to the fault is lost and replaced with some other value. The state corruption view suggests that one way to accommodate all of the fault-classes is to design the set of processes to be stabilizing [4], i.e., to recover from an arbitrarily corrupted state to one from where the specification of barrier synchronization is (re)satisfied. Unfortunately, a stabilizing design allows incorrect execution of (a finite number of) phases in the presence of each fault-class before the recovery is complete, and it is thus not ideal for the fault-classes that can be masked.

We therefore present in this paper barrier synchronization designs that offer multiple levels of tolerance corresponding to multiple fault-classes, a notion which we refer to as multitolerance [5]. Specifically, our designs are able to tolerate all state corruptions, while keeping for each fault-class the number of phases executed incorrectly to be the minimum that is possible for that fault-class.

[1] Email: {kulkarni,anish}@cis.ohio-state.edu; Web: http://www.cis.ohio-state.edu/{~kulkarni,~anish }.
Research supported in part by OSU Grant 221506, NSF CCR-93-08640, and NSA MDA904-96-1-1011.

2 Problem Statement

Specification of barrier synchronization. Given is an undirected, connected network of processes, each of which consists of a cyclic sequence of n terminating phases, [$phase.0$; $phase.1$; $\ldots$; $phase.(n{-}1)$]. The following two properties are required for each i, $0 \leq i < n$:

(*Safety*) No process executes $phase.(i+1 \;\; mod \;\; n)$ until all processes have successfully executed $phase.i$, and

(*Progress*) After all processes have successfully executed $phase.i$, they each eventually execute $phase.(i+1 \;\; mod \;\; n)$.

Initially, each process has successfully executed $phase.(n-1)$ and is thus ready to execute $phase.0$.

Fault-classes. We classify the faults that the barrier synchronization is subject to into two classes: "detectable" and "undetectable". Detectable faults are those that corrupt the state of a process in such a way that the corrupted state can be "reset" before it is accessed by any process, whereas undetectable faults are those that corrupt the state of a process in such a way that the corrupted state need not be reset before it is accessed by any process. For both classes, the state of a process before its corruption by a fault in that class is lost; hence, a reset after a fault may yield a state that differs from that before the fault.

We assume that the state of each process can be reset such that re-execution of a phase by all processes starting from their reset states has the same effect as a successful execution of that phase. It follows that detectable faults can be masked by requiring all processes to re-execute the current phase when a detectable fault occurs. Care has to taken, however, so that the failed execution and the re-execution of the current phase do not overlap, because execution of the processes in any phase may depend on each other. Recalling that phases are terminating, we may avoid any overlap by starting the re-execution only after no process is executing in the current phase. To illustrate this by way of example, let us consider a network of two processes, j and k, both in $phase.0$. If j is subject to a detectable fault, then k is unaware whether j has executed $phase.0$. So $phase.0$ has to be re-executed. But if k is currently executing $phase.0$ then the re-execution would start only after k finishes executing $phase.0$.

Undetectable faults cannot be masked since they allow processes to access corrupted state. We may however ensure that after undetectable faults occur, the system eventually reaches a state from where the specification of barrier synchronization is satisfied. Until such a state is reached, execution of some phases may violate Safety or Progress of barrier synchronization.

The problem. Our goal is to design a multitolerant barrier synchronization that tolerates both detectable and undetectable faults in a manner best suited to each fault-class. Thus, the problem is:

Design for the process network a barrier synchronization program that, when subject to a finite number of detectable or undetectable faults, (re)satisfies the barrier synchronization specification with minimum number of phases executed incorrectly.

To solve this problem, we proceed as follows: In Section 3, we design the program for the case where the number of phases in the cyclic sequence is 1. The resulting program 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 the number of phases is greater than

1. The resulting program recovers from detectable faults with no phase executed incorrectly, and from undetectable faults that perturb processes into, say, m distinct phases with at most m phases executed incorrectly. (Note that if all processes are corrupted detectably, the resulting state cannot be reset without violating Safety and Progress and, hence, this fault scenario is considered to be undetectable.)

Notation. We write programs in a guarded command notation: 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. For convenience, a unique name is associated with each action. Thus, each action has 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., if its guard evaluates to true. To execute the action, its statement is executed atomically. Also for convenience, we sometimes specify an action, say $\langle name' \rangle$, in terms of another action, say $\langle name \rangle$, using the form:

$$\langle name' \rangle :: \langle name \rangle \ \| \ \langle statement' \rangle$$

where the guard of $\langle name' \rangle$ is the same as that of $\langle name \rangle$, and the statement of $\langle name' \rangle$ is the statement of $\langle name \rangle$ in parallel with $\langle statement' \rangle$.

We represent each fault by an action. A detectable fault assigns to variables of a process "reset" values from their domains (note that this reset would in practice be implemented in the process, but since the corrupted state of the process cannot be accessed by any process until the reset is complete, it is convenient to specify the reset as part of the fault action). An undetectable fault assigns to variables of a process nondeterministically chosen values from their domains.

3 Step 1: Single Phase Barrier Synchronization

In this section, we consider the case where the cyclic sequence of phases consists of a single phase. To synchronize the underlying computation, each process j maintains some 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 *success*, which denotes that j has completed its phase. We consider two additional control position values: *ready*, which denotes that j is ready to execute 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*, in which case a state is reached where the control position of j is *success* and that of k is *ready*. The same state is also reached from the initial state in case 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, it can subsequently start executing the next phase, which would violate Safety in the first case (although not in the second).

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 *error* to *ready* occurs only when no process is in the control position *execute* and no process is in control position *success*. This restriction ensures that if any detectable corruption occurs during execution of a phase, re-execution of that phase does not begin as long as some process is executing that phase.

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

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* is invariantly true in program SB, where

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

Safety. In the absence of faults, as noted above, when a process changes its control position from *execute* to *success*, no process is in control position *ready*, i.e., all processes have started execution of their current phase. 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 that phase. Thus, when a process starts executing the next phase, by changing its control position from *ready* to *execute*, all processes have successfully completed the current phase.

In the presence of detectable faults, a process that is subject to state corruption changes its control position to *error*. Hence, *inv* 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 current execution.

Progress. In the absence of faults, once all processes have completed execution of a phase, they each execute $SB3$ to change their control position to *ready*. Once all processes change their control position to *ready*, they each eventually execute the next phase.

In the presence of detectable faults, *inv* continues to hold. In *inv* states, at least one action of program SB is enabled. Now consider the variant function: $3 * |\{j : cp.j = ready\}| + 2 * |\{j : cp.j = execute\}| + |\{j : cp.j = success\}|$. Execution of actions $SB1$ and $SB2$ decreases the value of the function whereas execution of actions $SB3$ and $SB4$ increases the value of the function. Since the value of the function is non-negative, it follows that eventually $SB3$ or $SB4$ 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*. From this state, the current phase is re-executed.

In the presence of undetectable faults, SB may reach a state where *inv* is violated. In such a state, $SB1$ and $SB2$ are always enabled at some process whereas $SB3$ and $SB4$ 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 $SB3$ or $SB4$ is executed. Since $SB3$ and $SB4$ execute only in a state where *inv* holds, it follows that eventually SB reaches a state where *inv* is satisfied and correct computation resumes.

Since a process cannot execute the transition from *success* to *ready* in a state where *inv* 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* is satisfied. $\square$

Remark. In presenting program SB, we have assumed that when a process changes its control position from *ready* to *execute*, it can decide whether it has to execute the next phase (in case the current phase was executed successfully) or to re-execute the current (in case the current phase may not executed successfully). This decision can be programmed by maintaining a variable that distinguishes the current phase from the next one. One strategy for maintaining this variable, despite detectable and undetectable faults, is presented in the next section.

4 Step 2: Multiphase Barrier Synchronization

In this section, we design the synchronization for the case where the cyclic sequence consists of multiple phases. To distinguish between the phases, each process j maintains a variable $ph.j$, whose value denotes the number of the phase that j is currently in.

As shown in Figure 3, the state transitions of the multiphase barrier synchronization are obtained by a simple repetition of the transitions of the single phase case. The transitions from *ready* to *execute* and from *execute* to *success* are respectively identical to $SB1$ and $SB2$.

The transition from *success* to *ready*, in addition to $SB3$, also chooses the phase in which the process will execute in. For the first process that executes this transition in the current phase, if the control position of all processes is *success* (i.e., all processes have completed the current phase successfully), its phase is incremented, thereby leading to the execution of the next phase; if, however, the control position of some process is *error* (i.e., some processes have not completed the current phase successfully), the phase is unchanged, thereby leading to a re-execution of the current phase. After the first process executes transition to *ready*, other processes obtain their phase from any process whose control position is *ready*.

The transition from *error* to *ready*, in addition to $SB4$, also obtains the phase in which the process will execute in from any process whose control position is *ready*. If, however, there is no process in control position *ready*, i.e., the phase of all processes is corrupted, the phase is chosen arbitrarily.

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

Proof of correctness: *Safety.* In the absence of faults, when a process increments its phase from i to i+1, all processes have successfully completed execution of *phase.i*, i.e., a process executes *phase.*(i+1) only after all processes have successfully executed *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 re-execute *phase.i* 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 re-execution of *phase.i* does not interfere with its current execution.

Progress. In the absence of faults, once all processes have successfully completed *phase.i*, i.e., the phase of all processes is i and their control position is *success*, action $MB3$ is enabled at all processes. Each process executes $MB3$ and changes its phase to $(i+1)$. Thus, the program reaches a state where all processes are in *phase.*$(i+1)$ and in control position *ready*. Starting from this state, each process executes *phase.*$(i+1)$.

In the presence of detectable faults, if some process is in control position *error* and *phase.i* is the current phase, the first process to execute $MB3$ remains in *phase.i*. After all processes in control position *success* execute $MB3$ and change their control position to *ready*, the processes in control position *error* execute action $MB4$ to change their control position to *ready* and set their phase to i. Starting from this state, each process re-executes *phase.i*.

In the presence of undetectable faults, MB eventually reaches a state where the control position of all processes is *success*, exactly as SB did. Starting from this state, all processes execute $MB3$ to enter the same phase and change their control position to *ready*. From this latter state, the specification of barrier synchronization is satisfied for all phases.

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 if a process executes $MB3$ to enter a new phase, that phase is executed correctly. Thus, at most m phases are executed incorrectly. □

An alternative program, MB'. In program MB, before a process increments its phase (cf. action $MB3$), it must check that no process has already executed $MB3$ in the current phase. This check can be avoided if the number of phases in a cycle is at least three, as follows: Let all processes be in *phase.i* and their control position be *success* or *error*. When they change their control position to *ready*, processes in control position *success* set their phase to $i+1$ and those in *error* set their phase to i. Thus, when all processes are in control position *ready*, they may be in two successive phases, i and i+1 of which the current phase can be defined to be i (provided the number of phases is at least three). Hence, when a process later changes its control position from *ready* to *execute*, it executes this current phase.

This simpler program, MB', is designed by modifying SB as follows: (1) whenever a process executes $SB1$, it sets its phase to the current phase that is to be executed, (2) whenever it executes $SB3$, it increments its sequence number, and (3) whenever it executes $SB4$, it sets its phase to the maximum phase number (of processes in control position *ready*) minus 1. The resulting program is shown in Figure 4.

In sum, the main distinctions between MB and $MB^{'}$ are that MB requires that the number of phases be at least two whereas $MB^{'}$ requires that the number of phases be at least three, but $MB^{'}$ permits more concurrency than MB in the presence of detectable faults.

5 Related Work

Instantiations of our multitolerant solution satisfy the specification and typical tolerance requirements of synchronization problems such as asynchronous clock unison, phase synchronization, and atomic commitment. A significant amount of work has already addressed fault-tolerant solutions to these problems, but to the best of our knowledge, the existence and systematic design of multitolerant solutions for these problems —and for barrier synchronization– has not received attention. We briefly recall these problems below and discuss how our barrier synchronization program can be used to design fault-tolerant programs for these problems.

Asynchronous unison. In the asynchronous unison problem [1], every process maintains a bounded-value clock such that, at all times, the difference between the clock values of two processes is at most 1, and each clock is incremented infinitely often. Traditionally, the tolerance requirement in asynchronous unison is stabilizing tolerance for the fault that corrupts the clocks of processes undetectably.

The barrier synchronization problem generalizes asynchronous unison problem in the sense that phase i of the computation may be mapped onto the i-th value of the clock. Since our solution is stabilizing tolerant to undetectable state corruption of phase, it meets the tolerance requirements of asynchronous unison.

Phase Synchronization. In the phase synchronization problem [2], 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 tolerance requirement in phase synchronization is to mask the fault that corrupts the phase of processes initially in (and not during) the computation.

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

Atomic commitment. In the atomic commit problem [3], 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. The decision reached by all processes must be identical. Traditionally, the tolerance requirement for atomic commitment is to mask the faults that force processes to crash or exhibit Byzantine behavior. Although these faults have not explicitly been considered in this paper, as they seem to corrupt actions —as opposed to state variables— in processes, it is possible to represent the corruption of actions by faults that corrupt "auxiliary" variables of processes and then design multitolerance accordingly [6].

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 let a process complete a phase successfully if it executes its current phase with the vote Yes; else (if it votes No) we "fail" the current phase by allowing a detectable fault at that process.

6 Concluding Remarks

In this paper, we presented multitolerant programs SB, MB, and MB' for barrier synchronization. SB handled the case where the cyclic sequence of phases consisted of a single phase whereas MB and MB' respectively handled the cases where the sequence consisted of at least two and at least three phases. Our programs had low space-complexity: four values per process for the control position and n values per processes for the phase number.

Two refinements of these programs are worthy of note. Since each process in these programs directly accesses the state of all processes, one refinement is therefore to distribute the processes so that each process directly accesses only its state and the state of its neighbors in the network. This distribution is achieved by using a "multitolerant diffusing computation". Another refinement is to reduce the atomicity of process actions so that in each atomic step a process either accesses the state of at most one neighboring process or updates its own state (but not both). This reduction of atomicity is achieved by using a "multitolerant snapshot". We refer the interested reader to [6] for the refined versions of our programs.

Acknowledgments. We thank the referees for their helpful comments on an earlier version of this paper. We are especially grateful to Referee 1 for suggesting the substantially simpler program MB' for the case where the sequence of phases consists of multiple processes.

References

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

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

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

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

[5] A. Arora and S. S. Kulkarni. Component based design of multitolerance. Revised for *IEEE Transactions on Software Engineering*, 1996.

[6] S. S. Kulkarni and A. Arora. Fine-grain multitolerant barrier synchronization. Technical Report OSU-CISRC TR34, Ohio State University, 1997.

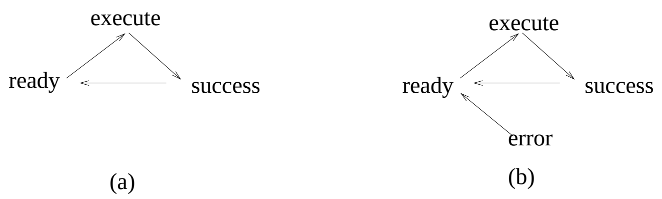

Figure 1: **State transitions for program SB**

$SB1 ::$	$cp.j = ready \;\wedge\; ((\forall k :: cp.k = ready) \;\vee\; (\exists k :: cp.k = execute))$	$\longrightarrow$	$cp.j := execute$
$SB2 ::$	$cp.j = execute \;\wedge\; ((\forall k :: cp.k \neq ready) \;\vee\; (\exists k :: cp.k = success))$	$\longrightarrow$	$cp.j := success$
$SB3 ::$	$cp.j = success \;\wedge\; (\forall k :: cp.k \neq execute)$	$\longrightarrow$	$cp.j := ready$
$SB4 ::$	$cp.j = error \;\wedge\; (\forall k :: cp.k \neq execute \;\wedge\; cp.k \neq success)$	$\longrightarrow$	$cp.j := ready$

Program SB

Detectable corruption of control position

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

Undetectable corruption of control position

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

Figure 2: **Program SB and its faults**

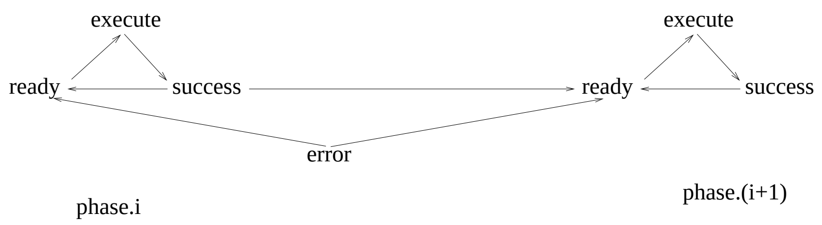

Figure 3: **State transitions for program MB**

$MB1$ ::	$SB1$		
$MB2$ ::	$SB2$		
$MB3$ ::	$SB3$	$\|\|$	if $(\forall k :: cp.k = success)$ then $ph.j := ph.j + 1$;
			if $(\exists k :: cp.k = ready)$ then $ph.j := (\underline{any}\ k : cp.k = ready : ph.k)$
$MB4$ ::	$SB4$	$\|\|$	$ph.j := (\underline{any}\ k : cp.k = ready : ph.k)$

where	$(\underline{any}\ k : k \in X : k)$	$=$	an arbitrary element from the set X	if $X \neq \{\}$
		$=$	an arbitrary element from the set $\{0..n-1\}$	otherwise

Program MB

$MB1'$ ::	$SB1$	$\|\|$	$ph.j := (\underline{curr}\ k : cp.k \neq error : ph.k)$
$MB2'$ ::	$SB2$		
$MB3'$ ::	$SB3$	$\|\|$	$ph.j := ph.j + 1$
$MB4'$ ::	$SB4$	$\|\|$	$ph.j := (\underline{next}\ k : cp.k = ready : ph.k) - 1$

where	$(\underline{curr}\ k : k \in X : k)$	$=$	$(minimum\ k : k \in X : k)$	if $X \neq \{0, n-1\}$
		$=$	$n-1$	otherwise
	$(\underline{next}\ k : k \in X : k)$	$=$	$(maximum\ k : k \in X : k)$	if $X \neq \{0, n-1\}$
		$=$	0	otherwise

Alternative Program MB'

Detectable corruption of phase and control position

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

Undetectable corruption of phase and control position

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

Figure 4: Programs MB and MB', and their faults