Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1998, Article 4
? December

ISSN 1073–0486. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493 USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://mitpress.mit.edu/CJTCS/*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *ftp://mitpress.mit.edu/pub/CJTCS*
- *ftp://cs.uchicago.edu/pub/publications/cjtcs*

The *Chicago Journal of Theoretical Computer Science* is abstracted or indexed in *Research Alert,*® *SciSearch,*® *Current Contents*®*/Engineering Computing & Technology*, and *CompuMath Citation Index.*®

The *Chicago Journal of Theoretical Computer Science* is a peer-reviewed scholarly journal in theoretical computer science. The journal is committed to providing a forum for significant results on theoretical aspects of all topics in computer science.

Multitolerance in Distributed Reset

Sandeep S. Kulkarni Anish Arora

? December, 1998

Abstract

Abstract-1 A reset of a distributed system is safe if it does not complete "prematurely," i.e., without having reset some process in the system. Safe resets are possible in the presence of certain faults, such as process fail-stops and repairs, but are not always possible in the presence of more general faults, such as arbitrary transients. In this paper, we design a bounded-memory distributed-reset program that possesses two tolerances: (1) in the presence of fail-stops and repairs, it always executes resets safely, and (2) in the presence of a finite number of transient faults, it eventually executes resets safely. Designing this multitolerance in the reset program introduces the novel concern of designing a safety detector that is itself multitolerant. A broad application of our multitolerant safety detector is to make any total program likewise multitolerant.

1 Introduction

1-1 A principal requirement of modern distributed systems, such as the evolving national and global information infrastructures, is robustness. *Robustness* implies that systems should be able to deliver their services in an appropriate manner in spite of variations in environmental behavior and occurrences of faults. Designing robustness in modern distributed systems is a challenging problem: not only is their scale unprecedented, but in a more fundamental sense, their characteristics are atypical in at least two ways: faults of various types are not only possible, but are likely to occur at some component or the other; and transient-system states are the norm and not the exception, since these systems are subject to constant change and evolution.

1-2 In systems that are subject to multiple fault classes, designing the same tolerance to all fault classes is often inappropriate, inefficient, or impossible. For example, in a system that is subject to timing glitches, configuration changes, and safety hazards, it may be desirable that the system recover from the first fault class, mask the second, and fail safely in the presence of the third. In such a system, it would be inappropriate to fail safely in the presence of all three fault classes, or it would be inefficient or impossible to mask all three fault classes. Thus, it is desirable that modern distributed systems be designed to be "multitolerant" [AK98a], that is, tolerant to each of a number of fault classes, in a manner that is best suited to each fault class.

1-3 In this paper, we present a bounded-memory multitolerant, distributed program that enables the processes in a distributed system to reset the state of the system. We focus our attention on the so-called distributed-reset program [AG94], owing to its wide applicability in designing fault tolerance. We consider two fault classes, one of which effectively fail-stops or repairs processes, and the other of which transiently and arbitrarily corrupts the state of processes. Our program is masking tolerant to the first fault class, and stabilizing tolerant to the second.

1-4 Below, we first define the reset problem and what it means for a reset program to be masking tolerant and stabilizing tolerant. We then describe the main difficulties involved in a bounded-memory reset program that is both masking tolerant and stabilizing tolerant.

1-5 **Distributed Reset.** A distributed-reset program consists of two modules at each process: an application module, and a reset module. The application module can initiate a reset operation to reset the program to any given global state. Upon initiation of the reset operation, the reset module resets the state of the application to a state that is reachable from the given global state, and then informs the application module that the reset operation is complete. In other words, each reset operation satisfies the following two properties:

- every reset operation is *nonpremature*, i.e., if the reset operation completes, the program state is reachable from the given global state; and
- every reset operation *completes eventually*, i.e., if an application module at a process initiates a reset operation, eventually the reset module at that process informs the application module that the reset operation is complete.

1-6 The definition captures the intuition that resetting the distributed program to exactly the given global state is not necessarily practical nor desirable, since that would require freezing the distributed program while the processes are individually reset. The definition therefore allows the program computation to proceed concurrently with the reset, to any extent that does not interfere with the correctness of the reset.

1-7 Observe that to reset the program state to the given global state, the application module at every process needs to be reset. Furthermore, if any two application modules communicate only if both have been reset in the same reset operation *and* all processes have been reset in a reset operation, then the current program state is reachable from the corresponding global state.

1-8 **Masking Tolerance.** The ideal fault tolerance for distributed reset is masking tolerance. A reset program is masking tolerant to a fault class if every reset operation is correct in the presence of the faults in that class. In other words, the safety of distributed reset (i.e., that every reset operation is nonpremature) is satisfied before, during, and after the occurrences of faults. Also, the liveness of distributed reset (i.e., that every reset operation completes) is satisfied after fault occurrences stop.

1-9 Consider a fault that fail-stops a process and repairs it instantaneously. It is possible to design masking tolerance to this fault. In fact, even if the fault only fail-stops processes or only repairs processes, it is still possible to design masking tolerance to the fault, with respect to the processes that are up throughout the reset operation. We therefore redefine a *premature* reset operation as follows: a reset operation is premature if its initiator completes it without resetting the state of all processes that are up throughout the reset operation. In the rest of the paper, we use this refined definition of premature reset.

1-10 **Stabilizing Tolerance.** An alternative fault tolerance for distributed reset is stabilizing tolerance. A reset program is said to be stabilizing tolerant to a fault class if, starting from any arbitrary state, eventually the program reaches a state from where every reset operation is correct, i.e., the safety and liveness of distributed reset are satisfied.

1-11 Stabilizing tolerance is ideal when an arbitrary state may be reached in the presence of faults. Arbitrary states may be reached, for example, in the presence of fail-stops, repairs, or message loss, as demonstrated by Jayaram and Varghese [JV96]. In such cases, masking tolerance cannot be designed, because the fault itself may perturb the program to a state where the reset operation has completed prematurely.

1-12 Since arbitrary states can be reached in the presence of arbitrary transient faults, the ideal tolerance to transient faults is stabilizing tolerance. From the definition of stabilizing tolerance, if the program is stabilizing tolerant to transient faults, it is also stabilizing tolerant to fail-stops and repairs. However, this is not the ideal tolerance to fail-stops and repairs.

1-13 **Multitolerant Reset.** As motivated above, the best-suited tolerance to fail-stop and repair faults is masking, and the best-suited tolerance to transient faults is stabilizing. We therefore design a multitolerant program that offers the best-suited tolerance for each of these two classes. (Note that by being stabilizing tolerant to transient faults, our program is also stabilizing tolerant to fail-stops and repairs; therefore, it is both masking and stabilizing tolerant to fail-stops and repairs.)

1-14 It is important to emphasize that our reset program is not just a stabilizing program. A reset program that is only stabilizing tolerant to fail-stops and repairs permits a reset operation to complete incorrectly in the presence of fail-stops and repairs. (All existing stabilizing reset programs in fact do so.) By way of contrast, our program ensures that in the presence of fail-stops and repairs, every reset operation is correct. In fact, as discussed below, designing a multitolerant reset program is significantly more complex compared to designing just a stabilizing reset program.

1-15 In principle, to design a multitolerant reset program, the initiator of the reset operation needs to "detect" whether all processes have been reset in the current operation. For the program to be masking tolerant to fail-stops and repairs, this "detection" must itself be masking tolerant to fail-stops and repairs. Also, for the program to be stabilizing, this detection must itself stabilize if perturbed to an arbitrary state. Thus, the design of the multitolerant reset program involves the design of a multitolerant detector.

1-16 Also, adding such a multitolerant detector to a stabilizing reset program is not sufficient for the design of a multitolerant reset program. Since the detector and the actions of the stabilizing program execute concurrently, the detector may interfere with the stabilizing program, making it nonstabilizing, and the stabilizing actions may interfere with the detector, causing incorrect detection. Thus, to design a multitolerant program, we also need to ensure freedom from interference between the detector and the actions of the stabilizing program.

1-17 Both these problems are further complicated if the program uses bounded memory. In the masking reset, when the initiator completes the reset operation, it needs to check that all processes are reset in the current reset operation. To this end, each process detects whether all its neighbors have reset their states in the current reset operation. Using bounded sequence numbers to distinguish between different reset operations is tricky, because it is possible that multiple processes will have the same sequence number even if they were last reset in different reset operations.

1-18 (Notice that sequence numbers for old reset operations may exist in the system. This is because some communication channels may be slower than others, communication channels may allow messages to be reordered, processes may repair with incorrect sequence numbers, or transient faults may arbitrarily corrupt the sequence numbers. In Section 5.2, we illustrate this with an example of multiple processes that end up with the same sequence number even though they have been reset in different reset operations.)

1-19 The complication due to bounded memory is more severe in designing masking tolerance than in stabilizing tolerance, as the former requires that the detection be always correct, whereas the latter requires that the detection be eventually correct.

1-20 As discussed in Section 8, our bounded-memory multitolerant solution for distributed reset is also useful in designing a bounded-memory multitolerant solution for any total program [Tel89], e.g., leader election, termination detection, or global snapshot. A total program characteristically has one or more "decider" actions whose execution depends on the state of all processes. The multitolerant detector in the reset program enables the design multitolerance in total programs so that in the presence of fail-stops and repairs, the decider actions always execute correctly, and after the occurrence of transient faults, the decider actions eventually execute correctly.

1-21 **Related Work.** To our knowledge, this is the first bounded-memory multitolerant distributed-reset program. In fact, we are not aware of a bounded-memory distributed-reset program that is masking tolerant to fail-stops and repairs. We note that Afek and Gafni [AG91] have shown a masking-tolerant solution under the severe assumption that processes do not lose their memory if they fail. They do, however, allow channels to fail, and the messages sent on those channels to be lost. Their program is not stabilizing tolerant.

1-22 While little work has been done on bounded-memory masking tolerant resets, bounded-memory stabilizing tolerant resets have received more attention [AG94, Var93, APSV91, APSVD94, AO94, AH93, DH97]. All of these programs are stabilizing tolerant to fail-stops and repairs, but they are not masking tolerant to them. Specifically, in the presence of fail-stops and repairs, they allow premature completion of distributed resets.

1-23 Masuzawa has presented a reset program [TM94] that tolerates two fault classes: transient faults and undetectable crash faults. His solution assumes that at most M processes fail undetectably for some fixed M such that the process graph is (M+1)-connected. While his solution ensures that in a reset operation eventually all processes are reset, it permits premature completion of a reset operation. Also, his solution uses unbounded memory.

1-24 **Outline of the Paper.** In light of the above discussion, we design a distributed reset program that (1) is masking tolerant to fail-stops and repairs, (2) is stabilizing tolerant to transient faults, and (3) has bounded space complexity. Using a stepwise approach for designing multitolerance, we proceed as follows: first, we design a fault-intolerant program for distributed reset. Then, we transform this program so as to add masking tolerance to fail-stops and repairs. Masking tolerance is itself added in two stages. In the first stage, we add nonmasking tolerance to fail-stops and repairs. The nonmasking program ensures that starting from a state reached as a result of fail-stops and repairs, the program eventually reaches a state from where every reset completes correctly. Until such a state is reached, however, some resets may complete incorrectly. In the second stage, we add the multitolerant detector that lets the initiator detect whether or not the reset is premature. If the reset is premature, the initiator repeats the reset operation. Finally, we transform this program again so as to add stabilizing tolerance to transient faults, while ensuring that the masking tolerance to fail-stops and repairs is preserved. (The interested reader is referred to [AK98a] for the foundations of this stepwise design of multitolerance.)

1-25 The rest of the paper is organized as follows: in Section 2, we define programs, faults, and fault tolerances, and state our assumptions about distributed systems. In Section 3, we give an outline of our solution and its proof of correctness. In Section 4, we develop the fault-intolerant program for distributed reset. In Section 5, we transform the fault-intolerant program to add masking tolerance to fail-stops and repairs. In Section 6, we transform the masking-reset program to add stabilizing tolerance to transient faults. In Section 7, we describe how this reset program can be used to design multitolerant application programs. Finally, we make concluding remarks in Section 8.

2 Preliminaries

2-1 In this section, we first recall a formal definition of programs, faults, and fault tolerances [AG93]. We then state our assumptions about the distributed systems considered in this paper.

2-2 **Programs.** A *program* is a set of variables and a finite set of actions. Each variable has a predefined domain. Each action is uniquely identified by a name, and is of the form:

$$\langle \text{name} \rangle :: \langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$$

2-3 The *guard* of an action is a boolean expression over the program variables. The *statement* of an action updates zero or more program variables. An action can be executed only if its guard evaluates to *true*. To execute an action, the statement of that action is executed atomically.

2-4 *State and State Predicate.* A state of program p is defined by a value for each variable of p, chosen from the predefined domain of the variable. A state predicate of p is a Boolean expression over the variables of p. We say that an action of p is enabled in a state if and only if its guard evaluates to *true* in that state.

2-5 *Closure.* Let S be a state predicate of p. An action of p preserves S if and only if in any state where S holds and the action is enabled, executing the statement of the action atomically yields a state where S holds. In a set of actions, S is closed in a set of actions if and only if each action in that set preserves S.

2-6 **Computation.** A computation of p is a fair, maximal sequence of steps; in every step, an action of p that is enabled in the current state is chosen, and the statement of that action is executed atomically. Fairness of the sequence means that each action in p that is continuously enabled along the states in the sequence is eventually chosen for execution. Maximality of the sequence means that if the sequence is finite, then the guard of each action in p is *false* in the final state.

2-7 *The Invariant Predicate.* One state predicate of p is distinguished as its invariant. The invariant predicate of p holds at all states reached by the computations of p which meet the problem specification that p satisfies. Thus, informally speaking, the invariant predicate characterizes the set of all states reached in the "correct" computations of p.

2-8 We assume that in all computations of p that start from a state where the invariant holds, the invariant predicate holds throughout. In other words, the invariant predicate of p is closed in p. Note that many other state predicates may be closed in p, some of which may not be *true* at states where the invariant holds; the state predicate *false* provides a trivial example of such a predicate.

2-9 Techniques for the design of the invariant predicate have been articulated by Dijkstra [Dij76], using the notion of auxiliary variables, and by Gries [Gri81], using the heuristics of state-predicate ballooning and shrinking. Techniques for the mechanical calculation of the invariant predicate have been discussed by Alpern and Schneider [AS87].

2-10 *Convergence.* A state predicate Q converges to a state predicate R in p if and only if Q and R are closed in p, and starting from any state where Q holds, every computation of p reaches a state where R holds.

2-11 **Faults.** The faults that a program is subject to are systematically represented by actions that perturb the state of the program. We emphasize that such representation is readily obtained for a variety of fault classes, including stuck-at, crash, fail-stop, repair, message loss, message corruption, timing faults, and Byzantine. Also, such a representation is readily obtained for faults of different natures, such as permanent, transient, and intermittent faults.

2-12 *The Fault Span.* The fault span of p, with respect to a set of fault actions F, is a predicate that characterizes the set of all states reached when the computations of p are subject to the occurrence of faults in F. Thus, if an enabled action is executed in a state where the fault span holds, the resulting state satisfies the fault span. Also, if a fault in F occurs in a state where the fault span holds, the resulting state also satisfies the fault span. In other words, the fault span of p with respect to a set of fault actions F is closed in both p and F.

2-13 **Fault Tolerances.** Let p be a program with the invariant S, and let F be a set of fault actions. We say that "p is F-tolerant for S" if and only if there exists a state predicate T of p that satisfies the following three conditions:

- inclusion: $T \Leftarrow S$,
- closure: T is closed in F, and
- convergence: T converges to S in p.

2-14 This definition may be understood as follows. At any state where the invariant S holds, executing an action in p yields a state where S continues to hold. However, executing an action in F may yield a state where S does not hold. Nonetheless, the following three facts are true about this resulting state : (1) T, the fault span, holds; (2) subsequent execution of actions in p and F yields states where T holds; and (3) when actions in F stop executing, subsequent execution of actions in p alone eventually yields a state where S holds, from which point the program resumes its intended execution.

2-15 In the context of multitolerance, it is important to note that there may be multiple predicates T for which p satisfies the above three conditions. Moreover, there may be multiple fault classes F for which p is F-tolerant.

2-16 When the invariant S is clear from the context, we omit S. Thus, the statement "p is F-tolerant" abbreviates "p is F-tolerant for the invariant of p."

2-17 **Types of Fault Tolerances.** The definition of fault tolerance lets us formally distinguish various types of fault tolerances, such as *masking*, *nonmasking*, and *stabilizing*:

1. When the definition is instantiated so that T is identical to S, p is masking tolerant to F. In other words, p is masking tolerant to F if S is closed in F; i.e., if a fault in F occurs in a state where S holds, the resulting state continues to satisfy S.

2. When the definition is instantiated so that T differs from S, p is nonmasking tolerant to F. In other words, p is nonmasking tolerant to F if when a fault in F occurs in a state where S holds, the resulting state may violate S; however, the resulting state satisfies T, and the continued execution of p alone yields a state where S is (re)satisfied, from which point the program resumes its intended execution.

3. When the definition is instantiated so that T is identical to *true*, p is stabilizing tolerant to F. In other words, p is stabilizing tolerant to F if when p reaches an arbitrary state, continued execution of p alone yields a state where S is (re)satisfied.

2-18 **Multitolerance.** Let p be a program with invariant S. Let $F1, F2, ..., Fn$ be n fault classes. We say that p is multitolerant to $F1, F2, ..., Fn$ if and only if for each fault class Fj, p is Fj-tolerant for S.

2-19 Informally, the definition can be understood as follows: in the presence of faults in Fj, the program is perturbed to a state where the fault span Tj

holds. After faults in Fj stop occurring, p recovers to a state where S is (re)satisfied. It is important to note that Tj may be different for each fault class Fj, and for each Fj, the tolerance of p to Fj may also be different.

2-20 This definition of multitolerance captures that the effect of different fault classes may be different. For example, in our reset program, we consider two sets of fault classes: (1) fail-stop and repair of processes, and (2) transient faults. Since the program is masking tolerant to the first fault class, the fault span in the prsence of fail-stop and repair will satisfy so that no reset completes prematurely. The program is stabilizing tolerant to the second fault class. Thus, the fault span in the presence of transient faults, namely, *true*, may allow a reset to complete prematurely.

2-21 **System Assumptions.** A distributed system consists of processes, each with a unique integer identifier, and bidirectional channels, each connecting a unique pair of processes. At any time, a process is either *up* or *down*. Only up processes execute their actions.

2-22 As mentioned previously, system execution may be affected by faults. These faults may fail-stop and repair processes, or arbitrarily corrupt the state of the system. Faults may occur in any finite number, in any order, and at any time, but they may never partition the graph of up processes.

Remark 1 Henceforth, we use the terms "process" and "up process" interchangeably. Also, when the context is clear, we use "process" to mean the "reset module of the process."

3 Outline of the Solution

3-1 Recall that in accordance with our approach to designing multitolerance we first design a fault-intolerant reset program, then transform this program to add masking tolerance to fail-stops and repairs, and finally add stabilizing tolerance to the program while ensuring that the masking tolerance to fail-stops and repairs is preserved. In this section, we outline the structure of each of these three programs and their proofs of correctness.

3-2 *The Fault-Intolerant Program.* We use a variation of the diffusing computation program. In particular, we use a tree that spans all up processes in the system, and perform the reset-diffusing computation only over the tree edges. The root of the tree initiates the diffusing computation of each reset. When a process receives a diffusing computation from its parent in the tree, it resets its local state, and propagates the diffusing computation to its children in the tree. When all descendents of process j have reset their local states, j completes its diffusing computation. Thus, when the root completes the diffusing computation, all processes have reset their local states.

3-3 *Adding Masking Tolerance.* In the presence of fail-stops and repairs, the tree may become partitioned. Hence, the diffusing computation initiated by a root process may not reach all processes in the system.

3-4 To add masking tolerance, it suffices that we ensure the following two conditions for every distributed reset:

1. eventually, the local states of all processes are reset, and
2. in the interim, no diffusing computation completes incorrectly.

3-5 To achieve condition 1, we ensure that eventually the tree spanning all up processes is restored, so that a diffusing computation can reach all up processes. To this end, we reuse a nonmasking-tolerant tree program due to Arora [Aro94] that, in the presence of fail-stops and repairs, maintains the graph of the parent relation of all up processes to always be a forest, and when faults stop occurring, restores the graph to be a spanning tree. The details of this program are given in Section 5.1.

3-6 To achieve condition 2, we ensure that when the root process completes a diffusing computation, it can detect whether all processes participated in that diffusing computation. Suppose that some processes have not participated when the root completes: since the up processes remain connected in the presence of fail-stops and repairs, it follows that there exists at least one pair of neighboring processes j and k such that j participated in the diffusing computation but k did not. To detect such pairs, a "result" is associated with each diffusing computation: process j completes a diffusing computation with the result *true* only if all its neighbors have propagated the diffusing computation, and hence, reset their local states. Otherwise, j completes the diffusing computation with the result *false*. The result is propagated in the completion wave of the diffusing computation. In particular, if j has completed a diffusing computation with the result *false*, then the parent of j completes that diffusing computation with the result *false*, and so on. Also, if j fails or moves to a different tree, then the (old) parent of j cannot always determine the result of j. Hence, when j fails or moves to a different tree, the parent of j completes with the result *false*. It follows that when the root completes the diffusing computation with the result *true*, all processes have participated in the diffusing computation.

3-7 Finally, if a root completes a diffusing computation with the result *false*, it starts a new diffusing computation. Since the nonmasking-tolerant tree program eventually spans a tree over the up processes, in any diffusing computation initiated after the tree is constructed and during which no failures occur, the distributed reset completes correctly.

3-8 (We reemphasize that the challenging part of this solution is how j decides—with a bounded sequence number—whether its neighbors have propagated the same diffusing computation as j has, and not just an old diffusing computation with the same sequence number. We discuss the issue of boundedness in Section 5.2.)

3-9 *Adding Stabilizing Tolerance.* To design stabilizing tolerance into the program, we add convergence actions that ensure that eventually the program reaches a state from where subsequent resets will complete correctly. In particular, in the presence of transient faults, the graph of the parent relation may form cycles, and so we add actions to detect and eliminate cycles in the graph of the parent relation. While adding stabilizing tolerance to transient faults, we preserve the masking tolerance to fail-stops and repairs by ensuring that the convergence actions and the actions of the masking program do not interfere. The details of this program are given in Section 6.

3.1 Outline of the Proof

3.1-1 In accordance with the formal definition of a fault-tolerant program given in Section 2, the proof of correctness of all programs in this paper is given in terms of their invariant and fault-span predicates.

3.1-2 For the fault-intolerant program R, we exhibit an invariant S_R. In other words, we show that if the program is executed in a state where S_R holds, then the resulting state satisfies S_R. Also, when the root completes the diffusing computation in a state where S_R holds, the state of the system is reachable from the given global state.

3.1-3 In the presence of fail-stops and repairs, program R may be perturbed to a state where the predicate S_R may be violated. Let T_R denote the predicate characterizing the set of states reached by program R in the presence of fail-stops and repairs. As mentioned above, to add masking tolerance to fail-stops and repairs, we add actions to detect whether all processes participated in a reset wave. Let D denote the invariant of these detection actions added to R to obtain the masking-tolerant program MR. Thus, for MR, the invariant is S_{MR}, $S_{MR} = T_R \ \wedge \ D$. By definition, the fault span of MR is identical to S_{MR}. In other words, we show that if an action of *MR* is executed or a fail-stop/repair fault occurs in a state where S_{MR} holds, the resulting state satisfies S_{MR}. Also, when the root completes the diffusing computation in a state where S_{MR} holds, the state of the system is reachable from the given global state.

3.1-4 In the presence of transient faults, program *MR* may be perturbed to an arbitrary state. Hence, the fault span of our stabilizing-tolerant program, *MSR*, is the predicate *true*. The invariant of the stabilizing-tolerant program is the same as that of the program *MR*, i.e., S_{MR}. As mentioned above, to add stabilizing tolerance to transient faults, we add convergence actions to program *MR* to ensure that upon starting from an arbitrary state, eventually a state satisfying S_{MR} is reached. After the program reaches a state where S_{MR} is satisfied, when a root completes the reset wave, the state of the system is reachable from the given global state. We also ensure that program *MSR* is masking tolerance to fail-stops and repairs. To this end, we show that the convergence actions preserve S_{MR} and do not execute indefinitely after the program reaches a state where S_{MR} holds. Hence, eventually only the actions of *MR* are executed. It follows that eventually a root process will declare that the reset is complete.

4 Fault-Intolerant Distributed Reset

4-1 In this section, we describe a straightforward distributed-reset program in terms of a diffusing computation over a rooted spanning tree. For simplicity, we assume that only the root process of the tree initiates distributed resets.

4-2 When the root process initiates a distributed reset, it marks its state as *reset*, resets its local state, and propagates a reset wave to its children in the tree. Likewise, when a process j receives a reset wave from its parent, j marks its state as *reset*, resets its local state, and propagates the reset wave to its children. We refer to these propagations as *the propagation of the reset wave*.

4-3 When a leaf process j propagates a reset wave, j completes the reset wave, marks its state as *normal*, and responds to its parent. When all children of process j have responded, j completes the reset wave, and responds to its parent. We denote these completions as *the completion of the reset wave*.

4-4 It follows that when a process completes a reset wave, all its descendents have completed that reset wave. In particular, when the root process

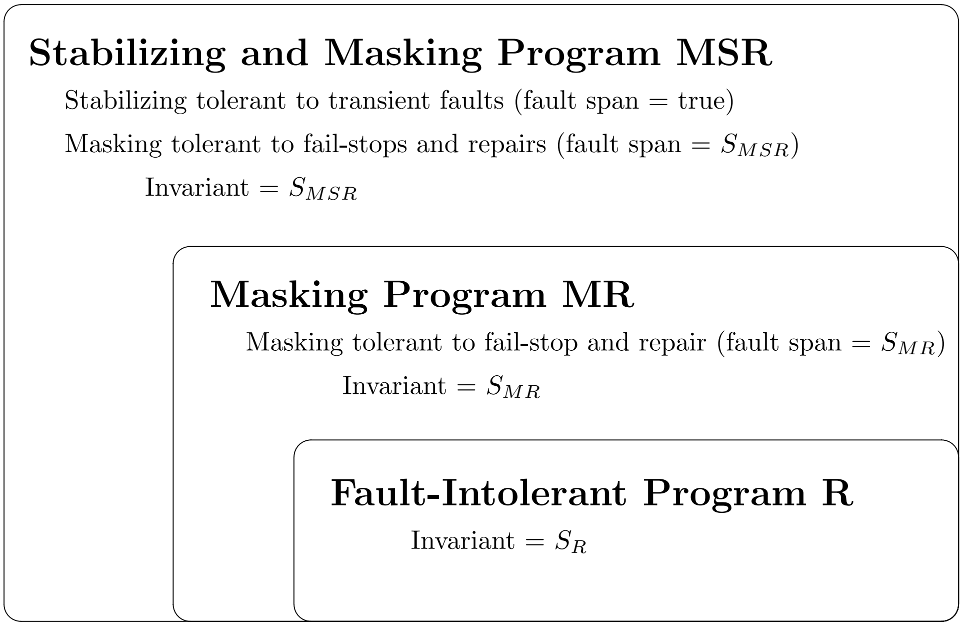

Stabilizing and Masking Program MSR
Stabilizing tolerant to transient faults (fault span = true)
Masking tolerant to fail-stops and repairs (fault span = S_{MSR})
Invariant = S_{MSR}
Masking Program MR
Masking tolerant to fail-stop and repair (fault span = S_{MR})
Invariant = S_{MR}
Fault-Intolerant Program R
Invariant = S_R

Figure 1: Structure of the program and its proof

completes the reset wave, all processes have completed the reset wave, and the root process can declare that the distributed reset has been successfully completed.

4-5 **Variables.** As described above, every process j maintains the following variables:

- *st.j*, the state of j: the state is *reset* if j is propagating a reset wave; otherwise, it is *normal*;

- *sn.j*, the sequence number of j: the sequence number of j is either 0 or 1; and

- *par.j*, the parent of j: the parent of the root process is set to itself.

4-6 **Actions.** As described above, program R consists of three actions at each process j. Action *R1* lets j initiate a new reset wave if j is a root process. Action *R2* lets j propagate a reset wave if *par.j* is propagating a reset wave and the sequence numbers of j and *par.j* are different. Action *R3* lets j complete a reset wave if j is in the reset state and all children of j have completed in that reset wave; if j is a root process, j also declares that the distributed reset is complete.

4-7 Formally, the actions of the program at process j are shown in Figure 2 (let *ch.j* denote the set of children of j):

```
$R1$ ::   $par.j = j \wedge st.j = normal \wedge$ {$j$ needs to initiate a new reset wave}
       $\longrightarrow st.j,\ sn.j := reset,\ sn.j \oplus 1;$   {reset local state of $j$}

$R2$ ::   $st(par.j) = reset \ \wedge \ sn.j \neq sn.(par.j)$
       $\longrightarrow st.j,\ sn.j := reset,\ sn.(par.j);$   {reset local state of $j$}

$R3$ ::   $st.j = reset \ \wedge \ (\forall k : k \in ch.j : sn.j = sn.k \ \wedge \ st.k \neq reset)$
       $\longrightarrow st.j := normal;$   {**if** $par.j = j$, **then** declare reset complete}
```

Figure 2: Actions of program R at process j

4-8 **Invariant.** Observe that if both j and *par.j* are propagating a reset wave, then they have the same sequence number. Also, if *par.j* is in the *normal* state, then j is also in the *normal* state and has the same sequence number as $par.j$. Hence, the predicate $GD = (\forall j :: Gd.j)$ is in the invariant of the program, where:

$$\begin{aligned} Gd.j = \; & ((st.(par.j) = reset \wedge st.j = reset) && \Rightarrow sn.j = sn.(par.j)) \wedge \\ & ((st.(par.j) \neq reset) && \Rightarrow (st.j \neq reset \wedge sn.j = sn.(par.j))) \end{aligned}$$

4-9 Moreover, the invariant of the program is:

$$S_R = GD \wedge \text{graph of the parent relation forms a tree.}$$

Remark 2 Although, for simplicity, we assumed that the reset request was initiated by the root process, our program can be generalized to let any process initiate a reset wave. To this end, any request made by a process is propagated toward the root, which then performs the reset. This extension is identical to the one given in [AG94], and is not discussed in this paper.

5 Masking Fault-Tolerant Distributed Reset

5-1 In this section, we transform the fault-intolerant program of Section 4 to add masking tolerance to fail-stop and repair faults. As described in Section 3, to add masking tolerance to fail-stop and repair faults, it suffices that:

- after faults stop occurring, the program eventually reaches a state from where no distributed reset ever completes incorrectly; and
- when a root process declares that a distributed reset has completed, all up processes have participated in that reset wave.

5-2 We design the masking-tolerant program in two stages. In the first stage, we transform the fault-intolerant program to add nonmasking tolerance to fail-stop and repair faults. The nonmasking fault-tolerant program ensures that in the presence of fail-stop and repair of processes, the program eventually reaches a state from where no distributed reset will complete incorrectly. In the second stage, we transform the nonmasking fault-tolerant program into one that is masking-fault tolerant, by restricting the actions of the former program so that a process declares completion of a distributed reset only if all up processes have participated in the last reset wave. (The interested reader is referred to [AK98b] for the foundations of this two-stage method for adding masking tolerance.)

5-3 In Section 5.1, we design the nonmasking program to tolerate fail-stop and repair faults. Then, in Section 5.2, we restrict the actions of the nonmasking program, so that the resulting program is masking-fault tolerant.

5.1 Nonmasking Fault-Tolerant Distributed Reset

5.1-1 Observe that if the program reaches a state where the invariant S_R holds, then starting from such a state no distributed reset will complete incorrectly. To design a nonmasking-reset program for fail-stop and repair faults, we add convergence actions that restore the program to a state where S_R holds; i.e., we add actions that (re)construct the rooted spanning tree and restore $Gd.j$ for every process.

5.1-2 To construct a spanning tree, we use Arora's program for tree maintenance [Aro94], which allows fail-stops and repairs to yield states where there are multiple, possibly unrooted, trees. Next we briefly describe how the program deals with multiple trees and unrooted trees, and eventually converges to a state where there is exactly one tree spanning all processes. We refer the reader to [Aro94] for the proof of the nonmasking program.

5.1-3 The program merges multiple trees such that no cycles are formed: each process maintains a variable *root.j* to denote the process that j believes to be the root. When process j observes a neighbor k such that *root.k* is greater than *root.j*, j merges in the tree of k by setting *root.j* to *root.k* and *par.j* to k. By merging thus, cycles are not formed, and the root value of each process remains at most the root value of its parent. This process continues until no merge action is enabled, at which point all processes have the same *root* value.

5.1-4 The program has actions to let each process detect if it is in an unrooted tree. To detect whether a process is in an unrooted tree, each process j maintains a variable *col.j* to denote the color of process j (which is either red or green). Whenever j detects that a parent of j has failed, j sets *col.j* to red, denoting that j is in an unrooted tree. This color is propagated from the tree root to the leaves so that all descendents of j detect that they are in an unrooted tree; i.e., when a process l observes that the parent of l has set its color to red, denoting that the parent of l is in an unrooted tree, l sets *col.l* to red. Finally, when a leaf process sets its color to red, it separates from the tree, forms a tree consisting only of itself, and sets its color to green, denoting that it is no longer in an unrooted tree. Thus, Arora's tree-maintenance program ensures that after faults stop occurring, the parent tree is (re)constructed.

5.1-5 To restore *Gd.j* at every process, we proceed as follows. We ensure that if *j* and *par.j* are in the same tree (i.e., if *root.j* is the same as *root.*(*par.j*) and their color is green), then *Gd.j* is satisfied. Also, when *j* merges into the tree of *k*, *j* satisfies *Gd.j* by copying the state and sequence number from *k*. It follows that in all stable states where the root values of all processes are equal, *Gd.j* holds for all processes.

5.1-6 **Variables.** Every process *j* maintains the following variables:

- *col.j*, the color of the process *j*: the color of *j* is either green or red; and
- *root.j*, the root of the process *j*: the identifier of the process that *j* believes to be the root.

5.1-7 **Actions.** Program *NR* consists of six actions for each process *j*. The first three actions implement the reset wave. Action *NR1* is the initiation action; it is the same as action *R1*. Action *NR2* is the propagation action; it is a restricted version of action *R2*, where *j* executes the action *NR2* only if *j* and *par.j* are green and in the same tree. Action *NR3* is the completion action; it is a restricted version of *R3*, where *j* executes action *R3* only if *col.j* is green and the *root* value of the children of *j* is the same as *root.j*.

5.1-8 The remaining three actions maintain a spanning tree. Action *NR4* deals with unrooted trees: if *j* detects that *par.j* has failed or that *col.*(*par.j*) is red, *j* sets *col.j* to red. Action *NR5* lets a leaf process change its color from red to green: if *j* is a red leaf, then *j* separates from its tree and resets its color to green, thus forming a tree consisting only of itself. Action *NR6* merges two trees: a process *j* merges into the tree of a neighboring process *k* when $root.k > root.j$. Upon merging, *j* sets *root.j* to be equal to *root.k* and *par.j* to be equal to *k*, and it copies the state and sequence number from *k*.

5.1-9 Formally, the actions of program *NR* at process *j* are as shown in Figure 3 (let *Adj.j* denote the neighbors of *j*):

```
NR1 ::      $R1$

NR2 ::      $root.j = root.(par.j) \;\wedge\; col.(par.j) = \text{green} \;\wedge$
            $st(par.j) = reset \;\wedge\; sn.j \neq sn.(par.j)$
        $\longrightarrow$ $st.j,\ sn.j := reset,\ sn.(par.j)$;   {reset local state of $j$}
```

```
$NR3 ::$          $st.j = reset \ \wedge \ col.j = \text{green}$
                  $(\forall k : k \in ch.j : root.j = root.k \ \wedge \ sn.j = sn.k \ \wedge \ st.k \neq reset)$
    $\longrightarrow$ $st.j := normal$;   **if** $par.j = j$, **then** {declare reset complete}

$NR4 ::$          $col.j = \text{green} \ \wedge \ (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = \text{red})$
    $\longrightarrow$ $col.j := \text{red}$

$NR5 ::$          $col.j = \text{red} \ \wedge \ (\forall k : k \in Adj.j : par.k \neq j)$
    $\longrightarrow$ $col.j, \ par.j, \ root.j := \text{green}, \ j, \ j$

$NR6 ::$          $k \in Adj.j \ \wedge \ root.j < root.k \ \wedge \ col.j = \text{green} \ \wedge \ col.k = \text{green}$
    $\longrightarrow$ $par.j, root.j, := k, root.k; \ st.j, sn.j := st.k, sn.k$
```

Figure 3: Actions of program NR at process j

5.1.3 Fault Actions

The fail-stop and repair actions are shown in Figure 4.

```
$fail\text{-}stop ::$   $up.j$      $\longrightarrow$   $up.j := false$
$repair ::$      $\neg up.j$   $\longrightarrow$   $up.j, par.j, root.j, col.j := true, j, j, \text{red}$
```

Figure 4: Fail-stop and repair actions of program NR

5.1-10 **Fault Span and Invariant.** From Arora's tree-maintenance program, we know that in the presence of fail-stops and repairs, the program actions preserve the acyclicity of the graph of the parent relation, as well as the fact that the root value of each process is at most the root value of its parent. They also preserve the fact that if a process is colored red, then its parent has failed or its parent is colored red. Thus, the predicate T_{TREE} is in the fault span, where:

T_{TREE} = the graph of the parent relation is a forest $\wedge$ $(\forall j : up.j : T1.j)$, where

$$T1.j = ((col.j = \text{red} \ \Rightarrow \ (par.j \notin Adj.j \cup \{j\} \ \ \vee \ \ col(par.j) = \text{red})) \ \wedge$$

$$(par.j = j \;\Rightarrow\; root.j = j) \;\wedge\; (par.j \neq j \;\Rightarrow\; root.j > j) \;\wedge$$
$$(par.j \in Adj.j \;\Rightarrow\; (root.j \leq root.(par.j) \;\vee\; col(par.j) = \text{red})))$$

5.1-11 Observe that $Gd.j$ is preserved when $root.j$ is the same as $root.(par.j)$, and $col.(par.j)$ is green. Hence, the predicate $NGD = (\forall j :: NGd.j)$ is in the fault span, where:

$$NGd.j \;=\; (root.j = root.(par.j) \;\wedge\; col.(par.j) = \text{green} \quad\Rightarrow\quad Gd.j)$$

Thus, the fault span of the program is:

$$T_{NR} \;=\; T_{TREE} \;\wedge\; NGD$$

In a stable state, the color of all processes is green, and the root values of all processes are identical. Thus, the invariant of the program S_{NR} is:

$$S_{NR} \;=\; T_{NR} \wedge S_R \wedge (\forall j, k :: par.j \in Adj.j \wedge col.j = \text{green} \wedge root.j = root.k)$$

Theorem 1 *Program NR is nonmasking tolerant to fail-stop and repair faults for invariant S_{NR} and fault span T_{NR}.*

5.2 Enhancing Tolerance to Masking

5.2-1 Program *NR*, being nonmasking tolerant, allows a reset operation to complete prematurely. To enhance the tolerance of *NR* to masking, we now add actions that detect whether all processes participated in the given reset operation. Recall from Section 3 that this detection is made possible by letting each process maintain for each reset wave a "result" that is true only if its neighbors have propagated that wave. The result of each process is propagated toward the initiator in the completion of the reset wave. In particular, if j has completed a reset wave with the result *false*, then the parent of j completes that reset wave with the result *false*, and so on. Also, if j fails or changes its tree, the (old) parent of j cannot always determine the result of j. Hence, when j fails or changes its tree, the parent of j completes the reset wave with the result *false*. It follows that when the root completes the reset wave with the result *true*, all processes have participated in the reset wave.

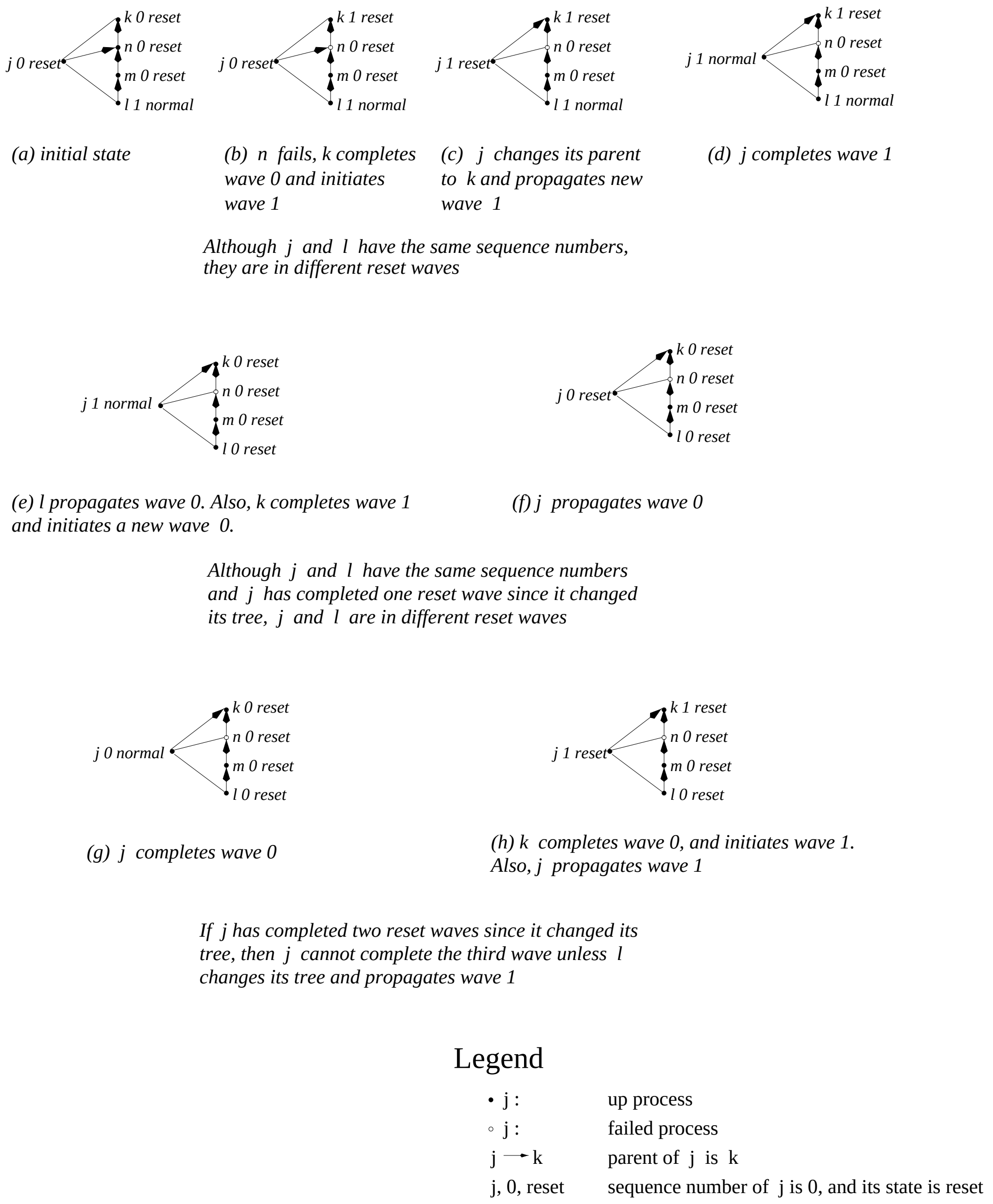
5.2-2 It remains to specify how a process detects whether its neighbors have propagated the current wave. One possibility is for j to detect whether the sequence numbers of all of its neighbors are the same as that of j. Unfortunately, because the sequence numbers are bounded, such a detection is insufficient. We illustrate this by an example. In particular, we exhibit a program computation whereby (1) even though j and l have the same sequence number, they are in different reset waves; and (2) even though j completes one reset wave after it changes its tree, and j and l have the same sequence number, they are in different reset waves.

5.2-3 Let the initial state be as shown in Figure 5a. Process k is the root, and the root value of all processes is k. Also, k has initiated a reset wave with sequence number 0, which all processes except l have propagated. The computation proceeds as follows:

- Process n fails.
- Process k completes its reset wave and initiates a new reset wave with sequence number 1 (Figure 5b).
- Process j separates from the tree, changes its parent to k, and propagates the reset wave with sequence number 1 (Figure 5c).
- Process j completes its reset wave (Figure 5d). Observe that when j completes this reset wave, although the sequence number of l is the same as that of j, l has not propagated the current reset wave of k. Thus, (1) is satisfied.
- Process l propagates the reset wave with sequence number 0. Also, k completes its reset wave and initiates a new reset wave with sequence number 0 (Figure 5e).
- Process j propagates the new reset wave of k with sequence number 0 (Figure 5f). Observe that although j has completed one reset wave since it changed its tree, and the sequence numbers of j and l are the same, they are in different reset waves. Thus, (2) is satisfied.

5.2-4 From the above computation, we observe that after j changes its tree, it cannot safely detect whether l is in the same wave as j during the subsequent two waves.

5.2-5 Fortunately, in the third wave after j changes its tree, j can safely detect whether l is in the same wave, provided j and l do not change their trees in the interim. Returning to our example:



(a) initial state

(b) n fails, k completes wave 0 and initiates wave 1

(c) j changes its parent to k and propagates new wave 1

(d) j completes wave 1

Although j and l have the same sequence numbers, they are in different reset waves

(e) l propagates wave 0. Also, k completes wave 1 and initiates a new wave 0.

(f) j propagates wave 0

Although j and l have the same sequence numbers and j has completed one reset wave since it changed its tree, j and l are in different reset waves

(g) j completes wave 0

(h) k completes wave 0, and initiates wave 1. Also, j propagates wave 1

If j has completed two reset waves since it changed its tree, then j cannot complete the third wave unless l changes its tree and propagates wave 1

Figure 5: Detecting whether a neighbor is reset in the current reset operation

- Process j completes the reset wave with sequence number 0. Again, observe that when j completes this second reset wave, l is still propagating an old reset wave (Figure 5g).
- Process k completes its reset wave and initiates a new reset wave with sequence number 1. Also, process j propagates this reset wave (Figure 5h).

5.2-6 Observe that starting from the state in Figure 5h, j can complete its reset wave only when the sequence number of l is 1. However, since all ancestors of l have sequence number 0 and none of them is a root, l cannot change its sequence number to 1 unless l changes its tree. Also, in any tree that l joins and completes two reset waves, l cannot propagate the next reset wave unless k is an ancestor of l, in which case the reset wave propagated by l is the current reset wave of k. Thus, if j and l both complete at least two reset waves since they changed their respective trees, j can safely detect whether the reset wave propagated by l is the current reset wave of k. Thus, the following lemma holds.

Lemma 1 (Sufficiency Condition for Bounded-Memory Safety Detection) *Let j, k, l be processes such that j and l are neighbors. Consider a state in S_{MR} where $root.j = root.l = root.k = k$, and k is an ancestor of j. In every computation of MR starting from this state, if j and l do not change their tree and complete two reset waves, then the next reset wave propagated by j (respectively, l) is the current reset wave being propagated by k.*

5.2-7 We use Lemma 1 to specify how j detects whether its neighbors have propagated the same reset wave as that of j, as follows. Process j maintains a ternary variable *new.j*, whose value is either 0, 1, or 2. When j changes its tree, *new.j* is set to 2. When j completes a reset wave, if *new.j* is nonzero, j decrements *new.j* by 1. Thus, j detects that it has completed at least two reset waves since it last changed its tree by checking that *new.j* is zero. Also, j detects that all its neighbors have propagated at least two reset waves since they changed their tree by checking that their *new* values are zero.

5.2-8 Thus, *new* implements the "result" associated with the reset wave. The *new* value being zero is equivalent to the result being *true*, and the *new* value being nonzero is equivalent to the result being *false*. Therefore, if the *new* value of j or any neighbor of j is nonzero, j sets *new.*(*par.j*) to a nonzero value to ensure that when *par.j* completes the reset wave, *par.j* sets *new*.(*par*.(*par*.*j*)) to a nonzero value, and so on. Thus, when the initiator completes the reset wave, its *new* value is nonzero, and the initiator concludes that the reset wave has completed incorrectly.

5.2-9 As mentioned in Section 3, if the initiator detects that the reset wave has completed incorrectly, it initiates a new reset wave. Also, if a process j fails, the parent of j cannot obtain the value of $new.j$. In that case, the parent of j aborts that reset wave by setting its *new* value to 2.

5.2-10 **Variables.** As described above, each process j additionally maintains the variable $new.j$, which is either 0, 1, or 2.

5.2-11 **Actions.** Program *MR* consists of six actions for each process j. The first three actions implement the reset wave. Action *MR1* is the initiation action; it is the same as action *NR1*. Action *MR2* is the propagation action; it is the same as action *NR2*. Action *MR3* is the completion action; it is a restricted version of *NR3*, where j executes action *NR3* only if the predicate $(\forall l : l \in Adj.j : root.j = root.l \ \wedge \ sn.j = sn.l)$ holds. Also, j updates the variable *new* as described above. If the initiator completes a reset wave incorrectly, it initiates a new reset wave. Actions *MR4*, *MR5*, and *MR6* maintain a spanning tree. When j changes its tree, by executing actions *NR5* and *NR6*, j sets the *new* value of j and the old parent of j to 2.

5.2-12 Formally, the actions for the process j are as in Figure 6 (for simplicity, we let actions *MR3*, *MR5*, and *MR6* at j update the value of $new.(par.j)$. As discussed in the Appendix (Section A3), this program can be refined so that j writes only the variables at process j.)

```
MR1 ::  NR1

MR2 ::  NR2

MR3 ::  $st.j = reset \wedge col.j = \text{green} \wedge$          $\longrightarrow$ $st.j := normal$;
        $(\forall l : l \in Adj.j : root.l = root.j \wedge$          if $(\exists l : l \in Adj.j \cup \{j\} : new.l > 0)$, then
                                   $sn.j = sn.l) \ \wedge$              if $(par.j = j)$, then
        $(\forall l : l \in ch.j : st.l \neq reset)$                            $st.j,\ sn.j := reset, sn.j \oplus 1$;
                                                                 {reset local state of $j$}
                                                             else
                                                                 $new.(par.j) := max(new.(par.j), 1)$
                                                         else if $(par.j = j)$ then    {declare reset
                                                         $new.j := max(0, new.j - 1)$     complete};
MR4 ::  NR4
```

$$\begin{array}{lll}
\mathit{MR5} :: & \mathit{col.j} = \text{red} \wedge & \longrightarrow \mathit{col.j},\ \mathit{par.j},\ \mathit{root.j} := \text{green},\ j,\ j; \\
 & (\forall k : k \in \mathit{Adj.j} : \mathit{par.k} \neq j) & \quad \mathit{new.j}, \mathit{new.}(\mathit{par.j}) := 2, 2 \\
 & & \\
\mathit{MR6} :: & k \in \mathit{Adj.j} \ \wedge \ \mathit{root.j} < \mathit{root.k} \wedge & \longrightarrow \mathit{par.j}, \mathit{root.j} := k, \mathit{root.k}; \\
 & \mathit{col.j} = \text{green} \ \wedge \ \mathit{col.k} = \text{green} & \quad \mathit{st.j}, \mathit{sn.j} := \mathit{st.k}, \mathit{sn.k}; \\
 & & \quad \mathit{new.j}, \mathit{new.}(\mathit{par.j}) := 2, 2
\end{array}$$

Figure 6: Actions of program MR at process j

5.2-13 **Fault Actions.** As described above, when a process j fail-stops, *par.j* needs to set *new.*(*par.j*) to 2. Since a process does not know whether j is its child, we implement this by letting all neighbors of j set their *new* value to 2. When a process j is repaired, j sets *new.j* to 2. Formally, the actions are described in Figure 7.

$$\begin{array}{lll}
\mathit{fail\text{-}stop} :: & \mathit{up.j} & \longrightarrow \mathit{up.j} := \mathit{false};\ (\forall l : l \in \mathit{Adj.j} : \mathit{new.l} := 2) \\
\mathit{repair} :: & \neg \mathit{up.j} & \longrightarrow \mathit{up.j}, \mathit{par.j}, \mathit{root.j}, \mathit{col.j}, \mathit{d.j}, \mathit{new.j} := \mathit{true}, j, j, \text{red}, 0, 2
\end{array}$$

Figure 7: Fail-stop and repair actions of program MR

Remark 3 In the fail-stop action, we have overloaded the operator ∀ to denote that the statement "*new.l* := 2" is executed at all processes in *Adj.j*. In the Appendix (Section A3), we observe that this parallel execution can be refined so that these statements are executed asynchronously.

5.2-14 **Invariant.** While we relegate the detailed invariant (S_{MR}) of the masking reset program to the Appendix (Section A1), we sketch here the essential predicates in the invariant. To characterize the predicates of the invariant, we define:

$$\begin{array}{ll}
X.j = & \{j\} \ \textbf{if} \ \mathit{par.j} = j \ \vee \ \mathit{par.j} \notin \mathit{Adj.j} \ \vee \ \mathit{col.j} = \text{red} \ \vee \ \mathit{root.j} \neq \mathit{root.}(\mathit{par.j}) \\
 & \{j\} \cup X.(\mathit{par.j}) \ \textbf{otherwise}
\end{array}$$

pc.j.k = set of processes to whom j propagated the current reset wave of k; thus, if j is propagating the current reset wave initiated by k, and a child of j (say l) propagates this reset wave by executing action *MR2*, l is added to *pc.j.k*. If j has not propagated the current reset wave initiated by k, or if k is not a root process, *pc.j.k* is the empty set

$des.j.k = \{j\} \cup (\bigcup l : l \in pc.j.k : des.l.k)$

$nbrs(des.j.k) = des.j.k \cup \{l : \exists j :: (l \in Adj.j \ \wedge \ j \in des.j.k)\}$

$failed.k =$ set of processes that repaired after k started its reset wave

5.2-15 Intuitively, $X.j$ is the set of ancestors of j that have the same root value as j and their color is green, and $des.j.k$ is the set of processes that have propagated the current reset wave initiated by k *via* j. It follows that $des.k.k$ denotes the set of processes that have propagated the current reset wave of k.

5.2-16 When j completes a reset wave initiated by k, j detects whether all of its neighbors have propagated the current reset wave of k. It also detects that all processes that have propagated this reset wave *via* j (i.e., $des.j.k$) have completed their detections successfully. Thus, all neighbors of $des.j.k$ have propagated the current reset wave of k (i.e., they are in $des.k.k$) or they are newly repaired processes (they are in $failed.k$). Thus, $nbrs(des.j.k)$ is a subset of $(des.k.k \cup failed.k)$.

5.2-17 Observe that when a process j, such that $k \in X.j$, completes a reset wave, j detects the predicate $nbrs(des.j.k) \subseteq (des.k.k \cup failed.k)$ by checking $new.j$ and the new values of its neighbors. If j cannot conclude that $nbrs(des.j.k) \subseteq (des.k.k \cup failed.k)$ holds, j sets $new.(par.j)$ to a nonzero value. And, $new.(par.j)$ remains nonzero until $par.j$ completes this reset wave. Thus, the predicate $I1$ is in the invariant, where:

$$\begin{aligned} I1 = (&k \in X.j \wedge new.j = 0 \wedge \\ &st.j \neq reset \wedge \\ &st.k = reset \wedge sn.j = sn.k) \Rightarrow nbrs(des.j.k) \subseteq (des.k.k \cup failed.k) \vee \\ &\qquad\qquad\qquad\qquad\qquad\quad new.(par.j) > 0 \vee \\ &\qquad\qquad\qquad\qquad\qquad\quad (par.j \neq j \ \wedge \ st.(par.j) \neq reset) \end{aligned}$$

5.2-18 Finally, the predicates T_{TREE} and NGD are in the invariant.

Lemma 2 *At any state where S_{MR} holds, if a root process declares that a distributed reset has completed correctly, then $nbrs(des.k.k) \subseteq (des.k.k \cup failed.k)$.*

Theorem 2 *Program MR is masking tolerant to fail-stop and repair faults for invariant S_{MR}.*

6 Stabilizing- and Masking-Tolerant Distributed Reset

6-1 In this section, we transform program *MR* to add stabilizing tolerance to transient faults. To this end, we add convergence actions to program *MR* that restore it from an arbitrary state to a state where S_{MR} holds.

6-2 We proceed as follows. Since each state where S_{MR} holds satisfies T_{TREE} and NGD, we add convergence actions to program *MR* that restore it from an arbitrary state to a state where $T_{TREE} \ \wedge \ NGD$ holds. Further, we show that starting from a state where T_{TREE} and NGD hold, the program converges to a state where S_{MR} holds.

6-3 By the definitions of T_{TREE} and NGD, $T_{TREE} \ \wedge \ NGD$ may be violated if any of the following three conditions hold:

1. the graph of the parent relation contains cycles,
2. there exists a process j such that $T1.j$ is violated, or
3. there exists a process j such that $NGd.j$ is violated.

6-4 To handle condition 1, each cycle in the graph of the parent relation is detected and removed. To detect cycles, j maintains a variable $d.j$ to be the distance between j and $root.j$ in the graph of the parent relation. Thus, if j is a root process, $d.j$ is zero; otherwise $d.j$ is $d.(par.j) + 1$. Observe that if the graph of the parent relation is acyclic and there are at most K up processes, then for all processes, their distance is less than K. Hence, whenever $par.j \in Adj.j$ and $d.(par.j) < K$, j maintains $d.j$ to be $d.(par.j) + 1$. If j belongs to a cycle, then $d.j$ increases repeatedly. Whenever $d.j$ exceeds $K-1$, a cycle is detected. To remove the detected cycle, j sets the parent of j to j.

6-5 To handle condition 2, whenever $T1.j$ is violated, j corrects $T1.j$ by separating from the tree and setting $par.j$ and $root.j$ to j.

6-6 To handle condition 3, whenever $NGd.j$ is violated, j corrects $NGd.j$ by copying the state and sequence number values from its parent.

6-7 **Variables.** As described above, each process maintains a variable $d.j$ to denote the distance of j from $root.j$ in the graph of the parent relation.

6-8 **Actions.** Program *MSR* consists of nine actions for each process j. Actions *MSR1–MSR6* construct the spanning tree and implement the reset wave; they are identical to the actions *MR1–MR6*. Action *MSR7* corrects

$d.j$ whenever $par.j \in Adj.j$ and $d.(par.j) < K$, by setting $d.j$ to $d.(par.j)+1$. Action *MSR8* is executed when $d.j$ exceeds K or $T1.j$ is violated. As described earlier, when j executes *MSR8*, j sets $par.j$ and $root.j$ to j and $d.j$ to zero. Since j changes its tree in action *MSR8*, j sets both $new.j$ and the new of the old parent of j to 2. Action *MSR9* is executed when j violates $NGd.j$. Additionally, j corrects $NGd.j$ by copying the state and sequence number from its parent.

6-9 Formally, the actions are as shown in Figure 8.

$$\begin{array}{lll}
\mathit{MSR1} :: & \mathit{MR1} & \\
 & & \vdots \\
\mathit{MSR6} :: & \mathit{MR6} & \\
\mathit{MSR7} :: & & par.j \in Adj.j \ \wedge \ d.j \neq d.(par.j)+1 \ \wedge \ d.(par.j) < K \\
 & \longrightarrow & d.j := d.(par.j)+1 \\
 & & \\
\mathit{MSR8} :: & & d.j \geq K \ \vee \ \neg T1.j \vee \ (par.j = j \ \wedge \ d.j \neq 0) \\
 & \longrightarrow & par.j, col.j, root.j, d.j, new.j, new.(par.j) := j, \mathrm{red}, j, 0, 2, 2 \\
 & & \\
\mathit{MSR9} : & & \neg NGd.j \\
 & \longrightarrow & st.j, sn.j := st.(par.j), sn.(par.j)
\end{array}$$

Figure 8: Actions of program MSR at process j

6-10 **Invariant.** The invariant of the stabilizing and masking program *MSR* is:

$$S_{MSR} = S_{MR}$$

6-11 As mentioned in Section 3.2, to preserve the masking tolerance to fail-stop and repair faults, we prove the following lemma.

Lemma 3 *Actions MSR7–MSR9 preserve S_{MR}, and do not execute indefinitely after program MSR reaches a state in S_{MR}.*

Theorem 3 *Program MSR is masking tolerant to fail-stop and repair faults, and is stabilizing tolerant to transient faults for invariant S_{MSR}.*

6-12

Remark 4 Since the program *MSR* is masking tolerant to fail-stop and repair faults, the fault span in the presence of fail-stop and repair faults, $T1$, is identical to the invariant S_{MSR}. Since the *MSR* is stabilizing tolerant to transient faults, the fault span in the presence of transient faults, $T2$, is *true*.

7 Bounding the Sequence Numbers of Multitolerant Applications

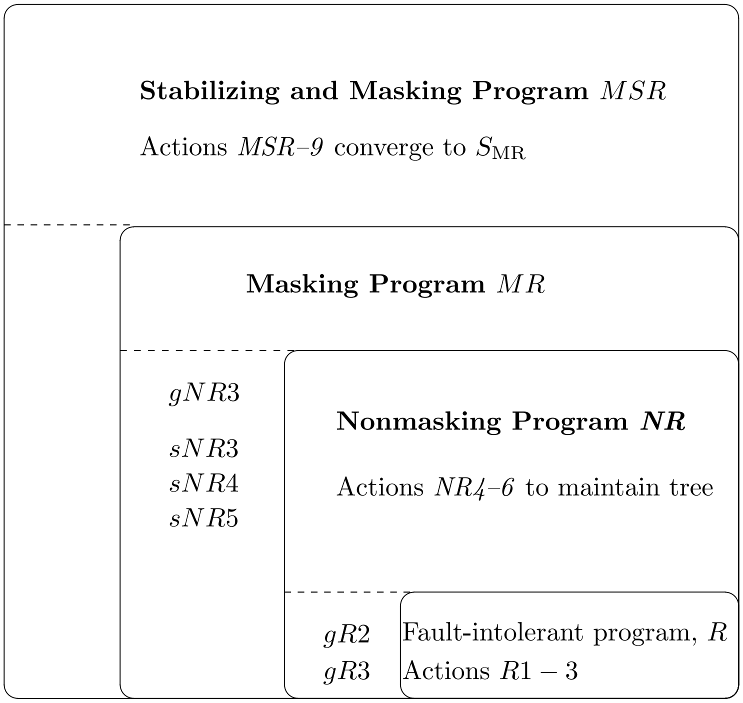
7-1 In this section, we discuss how applications use the bounded sequence numbers associated with the multitolerant distributed reset to become multitolerant. For ease of exposition, we focus our attention on the case when the application has to be made masking-fault tolerant to some fault class. In this case, when a fault is detected, the application is reset while ensuring that both the old and the new instances of the application execute correctly. In particular, we ensure that two processes communicate only when they have been reset in the same reset wave.

7.1 Need for Incarnation Numbers

7.1-1 If the sequence numbers associated with the distributed resets are unbounded, ensuring this property is simple: it suffices to restrict the communication between any two processes to occur only when their sequence numbers are identical. When the sequence numbers are bounded, however, it is possible for two processes to have the same sequence number even though they have reset their states in different reset waves.

7.1-2 We therefore introduce an *incarnation number* for each process. Two processes have the same incarnation number if and only if they have reset in the same reset wave. Hence, it suffices to restrict the communication between two processes to occur only when their incarnation numbers are identical.

7.1-3 It turns out that a bound of two on the incarnation number is insufficient. To see this, consider the following computation: The initial state is as shown in Figure 10a: processes 1, 2, and 3 form a tree rooted at 3. Let 3 initiate a reset wave to change the incarnation numbers to 1. Processes 1 and 3 propagate this reset wave, and change their incarnation number to 1 (Figure 10b). Before 2 propagates this reset wave, 3 fails, and the resulting tree is rooted at 2 (Figure 10c). Unfortunately, the state in Figure 10c can


Stabilizing and Masking Program MSR
Actions *MSR–9* converge to S_{MR}
Masking Program MR
$gNR3$
$sNR3$
$sNR4$
$sNR5$
Nonmasking Program *NR*
Actions *NR4–6* to maintain tree
$gR2$
$gR3$
Fault-intolerant program, R
Actions $R1-3$

Legend:

Action Yi of program Y is obtained by action Xi of program X with gXi added to the guard of Xi and sXi added to the statement of Xi. Unless explicitly mentioned, gXi is true and sXi is skip

$gR2 = (\mathit{root}.j = \mathit{root}.(\mathit{par}.j))$ and $\mathit{col}.(\mathit{par}.j) =$ green
$gR3 =$ (forall $k : k$ in $\mathit{ch}.j : \mathit{root}.j = \mathit{root}.k$ and $\mathit{col}.j =$ green)

$gNR3 =$ (forall $l : l$ in $\mathit{Adj}.j : \mathit{root}.j = \mathit{root}.l$ and $\mathit{sn}.j = \mathit{sn}.l$)

$sNR3 =$ update $\mathit{new}.j, \mathit{new}.(\mathit{par}.j)$
$sNR4 = \mathit{new}.j, \mathit{new}.(\mathit{par}.j) := 2, 2$
$sNR5 = \mathit{new}.j, \mathit{new}.(\mathit{par}.j) := 2, 2$

Figure 9: Composition of the masking- and stabilizing-reset program (*MSR*)

alternatively be reached by starting from a state in Figure 10d and letting process 2 initiate a new reset wave. In the first computation, all processes should change their incarnation numbers to 1 in the future. In the second computation, all processes should change their incarnation numbers to 0 in the future. Thus, keeping only two incarnation numbers is insufficient, as processes cannot determine which incarnation number is current.

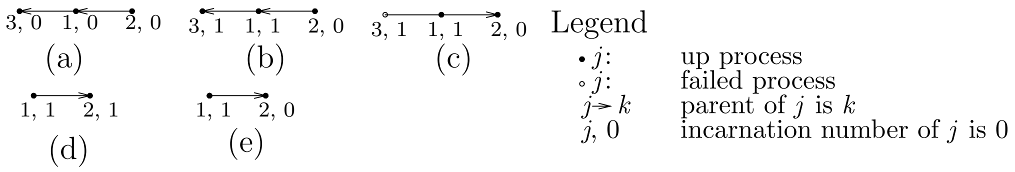


Figure 10: Insufficiency of two incarnation numbers

7.2 Designing Multitolerant Applications using Three Incarnation Numbers

7.2-1 It is possible to determine the current incarnation number if we use ternary $(0, 1, \text{or } 2)$ incarnation numbers and follow the rule that a new incarnation number $B + 1$ can be created only if the incarnation number $B - 1$ does not exist in the system (in this section, we let $+$ and $-$ denote modulo 3 addition and subtraction, respectively.) It follows that at any invariant state, at most two incarnation numbers may coexist. Also, for any pair of incarnation numbers, it is easy to determine which of the two is associated with the current reset wave.

7.2-2 *When to Reset the Local State of a Process.* When process j propagates a reset wave, it resets its state if and only if the incarnation number of the parent of j is one greater (in mod 3 arithmetic) than the incarnation number of j. When j resets its state, it also copies the incarnation number of its parent.

7.2-3 *When to Create a New Incarnation Number.* As motivated above, all three incarnation numbers should not exist simultaneously. Hence, we ensure that a process changes its incarnation number from B to $B + 1$ only if no process with incarnation number $B - 1$ exists in the system. From the invariant of the reset program, we know that when the root completes a reset wave successfully, all processes have propagated that reset wave. Thus, if a process with incarnation number $B-1$ existed in the system when the root initiated the reset wave, this process would have set its incarnation number to B upon propagation of the reset wave. Also, no process can change its incarnation number from B to $B-1$. Hence, when the root process completes a reset wave successfully, it can safely increment its incarnation number.

7.2-4 *When to Declare that Reset is Complete.* When a root completes a reset wave successfully with incarnation number B, no process in the system has incarnation number $B-1$. It is, however, possible that some processes have incarnation number $B+1$. (Such a state may arise if the tree program converges to a state where the root does not have the current incarnation number). Hence, we let the root declare completion of a reset only if the reset completes correctly *and* the root can detect that the incarnation numbers of all processes are the same. To this end, we let each process check whether all its children have the same incarnation numbers when it completes in the reset wave. The completion wave propagates this information toward the root so that the root can determine whether the incarnation numbers of all processes are the same.

7.2-5 **Variables.** Every process j maintains the following variables:

- $inc.j$, the incarnation number: one of 0, 1, or 2, and
- $ares.j$, application result: a Boolean.

7.2-6 **Actions.** Program *APP* consists of nine actions for each process j. The first three actions implement the reset wave. These actions are obtained by modifying actions *MSR1*–*MSR3* to update *inc* and *ares* appropriately. Action *APP1* is the initiation action; whenever j initiates a reset wave, j increments its incarnation number if and only if $new.j$ is zero. Action *APP2* is the propagation action; whenever j propagates a reset wave, j increments its incarnation number if and only if the incarnation number of the parent of j is one higher than that of j. Action *APP3* is the completion action; whenever j completes a reset wave, j sets $ares.j$ to *true* if and only if all children of j have the same incarnation number and their application result is *true*. Finally, the remaining actions *APP4*–*APP9* break cycles in the graph of parent relation and maintain the underlying tree; these actions are identical to the actions *MSR4*–*MSR9* of program *MSR*.

```
APP1 ::   $par.j = j \;\wedge\; st.j = normal \wedge \{j$ needs to initiate a new reset wave$\}$
          $\longrightarrow st.j,\; sn.j := reset,\; sn.j \oplus 1;$
```

```
                if $(new.j=0)$ then        $inc.j := inc.j+1$; {reset application state}

$APP2$ ::       $root.j=root.(par.j) \ \wedge\ col.(par.j)=\text{green} \ \wedge\ st(par.j)=reset$
                $\wedge\ sn.j \neq sn.(par.j)$
        $\longrightarrow$ $st.j,\ sn.j := reset,\ sn.(par.j)$;
                if $((inc.j+1) = inc.(par.j)))$ then
                    $inc.j := inc.(par.j)$; {reset application state}

$APP3$ ::       $st.j=reset \ \wedge\ col.j=\text{green} \ \wedge$
                $(\forall l : l \in ch.j : st.l \neq reset) \ \wedge$
                $(\forall l : l \in Adj.j : root.l = root.j \wedge sn.j = sn.l)$
        $\longrightarrow$ $st.j, ares.j := normal, (\forall l : l \in ch.j : ares.l \ \wedge\ (inc.j=inc.l))$;
                if $(\exists l : l \in Adj.j \cup \{j\} : new.l > 0)$ then
                    if $(par.j=j)$ then $st.j,\ sn.j := reset, sn.j \oplus 1$;
                    else $new.(par.j) := max(new.(par.j), 1)$
                else if $(par.j=j \ \wedge\ ares.j)$ then        {declare reset complete}
                else if $(par.j=j \ \wedge\ \neg ares.j)$ then
                    $st.j, sn.j, inc.j := reset, sn.j \oplus 1, inc.j + 1$;
                    {reset application state};
                $new.j := max(0, new.j-1)$

$APP4$ :: $MSR4$
                ⋮
$APP9$ :: $MSR9$
```

Figure 11: Actions of program *APP* at process j

7.2-7 *Outline of the Proof.* We need to show that:

1. when a root process with incarnation number B declares that a reset is complete, all processes have incarnation number B;
2. at most two incarnation numbers coexist at any invariant state; and
3. after a reset operation is initiated at an invariant state, the root process eventually declares that the reset is complete.

7.2-8 When j completes a reset wave, it checks that the incarnation number of j is the same as its children. Thus, when j completes its reset wave, it can detect whether all processes that received the reset wave via j have the same incarnation number. Recall that when the root declares that reset is complete, we know from *MSR* that all processes have propagated that reset wave. Hence, when the root declares that a reset is complete, all processes have the same incarnation number. Thus, condition 1 above is satisfied.

7.2-9 To satisfy condition 2, we show that when a process creates a new incarnation number $B+1$ after completing a reset wave with incarnation number B, no process has incarnation number $B-1$. Suppose not: consider the first process, say j, that has incarnation number $B-1$ after propagating this reset wave. We consider three cases, depending upon the value of the incarnation number of $par.j$ when j propagated this reset wave: (1) the incarnation number of $par.j$ cannot be $B-1$, since $par.j$ has propagated the reset wave and j is the first process to change its incarnation number to $B-1$; (2) if the incarnation number of $par.j$ is B, then after propagating the reset wave, the incarnation number of j is either B or $B+1$; and (3) if the incarnation number of $par.j$ is $B+1$, then before j propagates the reset wave, the incarnation number of j cannot be $B-1$, as only two incarnation numbers coexist. Thus, condition 2 above is satisfied.

7.2-10 From *MSR*, we know that eventually the root with incarnation number B will either declare that reset is complete or initiate a new reset wave with incarnation number $B+1$. Upon propagation of this reset wave, all processes with incarnation number B change their incarnation number to $B+1$. It follows that the root will eventually declare that the reset is complete. Thus, condition 3 is satisfied.

8 Conclusions

8-1 In this paper, we presented a bounded-memory distributed reset program that is distinguished by its ability to be both masking tolerant to any finite number of fail-stops and repairs and stabilizing tolerant to transient faults. The program was designed by iteratively transforming a fault-intolerant program so as to add tolerance to each class of faults.

8-2 Distributed reset programs belong to the class of total programs [Tel89]. Total programs characteristically contain one or more “decider” actions, whose execution depends on the state of all processes. In the case of distributed reset programs, the action that declares the completion of a reset operation is a decider action. The safety of the execution of this decider action was achieved in this paper by using a detector that ensures that (1) if fail-stops and repairs occur, the decider action is executed only after all processes are contacted; and (2) after transient faults occur, eventually the program reaches a state from where the decider action executes only after all processes are contacted. This strategy provides a basis for making other total programs likewise multitolerant: before a process executes a decider action, it waits for the detector to collect the state of all processes (i.e., whenever a process is contacted, its state is collected). From points 1 and 2, it follows that the resulting total program is multitolerant.

8-3 Our discussion of distributed reset has intentionally omitted how the application module chooses the global reset state parameter for each reset operation, but it is worthwhile to point out that these global states may be determined dynamically, say by a checkpointing program.

8-4 In our program, in stable states, the processes form a rooted tree and all reset operations are performed on this underlying tree. An alternative approach would be to construct the tree dynamically in every reset operation, as in [APSV91, Var93]. Analogous to their reset program, there exists a reset program that is masking tolerant to fail-stop and repair faults and stabilizing tolerant to transient faults and that does not use an underlying tree.

8-5 Finally, we reiterate that our assumption that the distributed reset request be initiated only at the root process is not necessary. The program is easily extended to allow any process to initiate a reset, by propagating a reset request toward the root along the parent tree. (The interested reader is referred to [AG94] for the implementation of this request propagation.) Moreover, the program is systematically extended to add masking tolerance to channel failure and repairs, as well as to use message-passing communication.

A Appendix

A.1 Proof of Masking-Fault Tolerance of Program *MR*

A.1-1 **Invariant for the Masking-Distributed Reset.** As described in Section 5, the predicates T_{TREE}, NGD, and $I1$ are in the invariant, S_{MR}, of the masking reset. Here, we characterize the remaining predicates in S_{MR}.

A.1-2 First, we recall the definition of $X.j$ from Section 5.2:

$$\begin{array}{rcll} X.j & = & \{j\} & \textbf{if } par.j = j \ \vee\ par.j \notin Adj.j \\ & & & \quad \vee\ col.j = \text{red} \ \vee\ root.j \neq root.(par.j) \\ & & \{j\} \cup X.(par.j) & \textbf{otherwise} \end{array}$$

A.1-3 Intuitively, $X.j$ denotes the green-colored ancestors of j that have the same root value as j.

A.1-4 For the following discussion (predicates $I2$–$I8$), we assume that j and l are two processes such that $k \in X.j$ (i.e., there exists a path from j to k in the graph of the parent relation, and all processes on this path are green and have the root value k), and $l \in Adj.j$ (i.e., l is a process adjacent to j).

A.1-5 When j completes a reset wave, $sn.j$ is the same as $sn.l$. The predicate $sn.j = sn.l$ is violated only if either $sn.j$ or $sn.l$ is updated. The variable $sn.j$ is updated only if j changes trees or propagates a reset wave. If $j(l)$ changes its tree, violating $sn.j = sn.l$, then $new.j(new.l)$ is set to 2. When l propagates a reset wave, from *NGD*, all processes in $X.l$ have the same sequence number. When j propagates a reset wave, j is in the reset state and it cannot complete the reset wave until $sn.j = sn.l$ is (re)satisfied. Thus, the predicate $I2$ is in the invariant, where:

$$\begin{array}{lcl} I2 = (new.j \neq 2 \ \wedge\ new.l \neq 2) & \Rightarrow & \quad (st.j \neq reset \ \equiv\ sn.j = sn.l) \\ & & \vee\ (\forall m : m \in X.l : sn.m = sn.l) \end{array}$$

A.1-6 When j completes a reset wave and decrements $new.j$ from 1 to 0, j is in the reset state and $sn.j$ is the same as $sn.l$. Thus, from the predicate $I2$, the second disjunct, $(\forall m : m \in X.l : sn.m = sn.l)$, must be *true*; i.e., all processes in $X.l$ have the same sequence number. Also, when j completes a reset wave, $sn.l$ is the same as $sn.j$, which in turn is equal to $sn.k$ (from *NGD*). Thus, the predicate $I3$ is in the invariant, where:

$$\begin{array}{lcl} I3 = (st.j \neq reset \ \wedge\ sn.j = sn.k & & \\ \quad\ \wedge\ new.j = 0 \ \wedge\ new.l \neq 2) & \Rightarrow & (\forall m : m \in X.l : sn.m = sn.k) \end{array}$$

A.1-7 Consider the case where k starts a new reset wave by changing $sn.k$: From $I3$, observe that all processes in $X.l$ have the same sequence number (in this case, different from $sn.k$) unless $k \in X.l$. If $k \in X.l$, then trivially there exists a process in $X.l$ (namely, k itself) which has propagated the current reset wave of k. Thus, the predicate $I4$ is in the invariant, where:

$$\begin{array}{lll} I4 = (sn.j \neq sn.k \;\wedge\; new.j = 0 & & \\ \quad\;\; \wedge\; new.l \neq 2) & \Rightarrow & (\forall m : m \in X.l : sn.m \neq sn.k) \\ & & \vee\; (sn.l \neq sn.k \;\wedge \\ & & \quad (\exists m : m \in X.l \cap des.k.k : sn.m = sn.k)) \\ & & \vee\; (sn.l = sn.k \;\wedge\; l \in des.k.k) \end{array}$$

A.1-8 Starting from a state where $I4$ holds, if j propagates a reset wave initiated by k, then the consequent of $I4$ continues to hold. When l executes the completion action and *new.l* is decremented from 2 to 1, from *NGD*, all ancestors of l have the same sequence number. Thus, the predicate $I5$ is in the invariant, where:

A.1-9

$$\begin{array}{lll} I5 = (sn.j = sn.k \wedge st.j = reset \wedge & & \\ \quad\;\; new.j = 0 \wedge new.l \neq 2) & \Rightarrow & (\forall m : m \in X.l : sn.m \neq sn.k) \\ & & \vee\, (sn.l \neq sn.k \wedge \\ & & \quad (\exists m : m \in X.l \cap des.k.k : sn.m = sn.k)) \\ & & \vee\, (sn.l = sn.k \wedge l \in des.k.k) \\ & & \vee\, (new.l = 1 \wedge \\ & & \quad (\forall m : m \in X.l : sn.m = sn.k) \wedge st.l \neq reset) \end{array}$$

A.1-10 As claimed earlier, when a process j completes a reset wave, if *new.l* and *new.j* are zero, then l has propagated the current reset wave initiated by k. Thus, the predicate $I6$ is in the invariant, where:

$$\begin{array}{lll} I6 = & (sn.j = sn.k = sn.l \;\wedge\; st.j = reset \;\wedge & \\ & new.j = 0 \;\wedge\; new.l = 0) & \Rightarrow \quad (l \in des.k.k) \end{array}$$

A.1-11 When k starts a new reset wave, if *sn.j* is different from *sn.k*, j has not propagated this reset wave, i.e., *pc.j.k* is the empty set. When j propagates a reset wave, all processes in *pc.j.k* are children of j, unless some of these children have moved to a different tree or failed. If one of the processes in *pc.j.k* moves to a different tree or fails, *new.j* is set to 2. Thus, the predicate $I7$ is in the invariant, where:

$$\begin{array}{llll} I7 = & (new.j = 0) & \Rightarrow & (sn.j \neq sn.k \;\Rightarrow\; pc.j.k = \phi) \\ & & & \wedge\; (sn.j = sn.k \;\Rightarrow\; pc.j.k \in ch.j) \end{array}$$

A.1-12 Finally, if j and $par.j$ are propagating a reset wave initiated by k and $par.j$ is propagating the current wave of k, j is also propagating the current wave of k. Thus, the predicate $I8$ is in the invariant, where:

$$I8 = \quad (par.j \in des.k.k \wedge sn.j = sn.(par.j) \wedge st.j = reset) \Rightarrow (j \in des.k.k)$$

A.1-13 Thus, the invariant of the masking distributed-reset program, *MR*, is:

$$\begin{aligned} S_{MR} = (NGD \ \wedge \ T_{TREE} \ \wedge \ (\forall j,l : k \in X.j \ \wedge \ l \in Adj.j \ \wedge \ root.j = root.l = k :& \\ I1 \ \wedge I2 \ \wedge I3 \ \wedge I4 \ \wedge I5 \ \wedge I6 \ \wedge I7 \ \wedge I8)& \end{aligned}$$

Lemma 4 *Consider a state in S_{MR} where $root.j = root.l = root.k = k$ holds, j and l are neighbors, and k is an ancestor of j. In every computation of MR starting from this state, if j and l do not change their tree and complete two reset waves, then the next reset wave propagated by j (respectively, l) is the current reset wave being propagated by k.*

Proof of Lemma 4 When j concludes that j and l are propagating the same reset wave, $new.j$ and $new.l$ is zero. From $I6$, it follows that l has propagated the current reset wave propagated by k.

Proof of Lemma 4 □

Lemma 5 *At any state where S_{MR} holds, if a root process k declares that a distributed reset has completed (by executing action MR3), then $nbrs(des.k.k) \subseteq (des.k.k \cup failed.k)$.*

Proof of Lemma 5 When k declares that reset is complete by executing action *MR3*, we have:

$$\begin{aligned}
& (\forall l : l \in (Adj.k \cup \{k\}) : new.l = 0 \ \wedge \ root.l = root.k \ \wedge \ sn.l = sn.k) \ \wedge \\
& (\forall l : l \in ch.k : st.j \neq reset) \ \wedge \ (st.k = reset) \\
\Rightarrow \quad & \{\text{from } I6, I1, I7\} \\
& (\forall l : l \in (Adj.k \cup \{k\}) : l \in des.k.k) \ \wedge \\
& (\forall l : l \in ch.k : nbrs(des.l.k) \subseteq (des.k.k \cup failed.k)) \ \wedge \ (pc.k.k \subseteq ch.k) \\
\Rightarrow \quad & \{\text{by predicate calculus}\} \\
& (\forall l : l \in Adj.k : l \in des.k.k) \ \wedge \\
& (\forall l : l \in pc.k.k : nbrs(des.l.k) \subseteq (des.k.k \cup failed.k)) \\
\Rightarrow \quad & \{\text{by definition of } des.k.k\} \\
& nbrs(des.k.k) \subseteq (des.k.k \cup failed.k)
\end{aligned}$$

Proof of Lemma 5 □

A.2 Proof of Stabilizing Fault Tolerance of Program *MSR*

Theorem 4 *Program MSR is masking tolerant to fail-stop and repair faults, and stabilizing tolerant to transient faults for invariant* S_{MSR}.

Proof of Theorem 4 Here, we show that *MSR* is stabilizing tolerant to transient faults. The proof of masking tolerance follows from Lemma 6 below. Proof of stabilization is based upon the "convergence stair" method of Gouda and Multari [GM91]. In particular, we exhibit predicates $S.1, S.2, S.3$, and $S.4$, where:

$$\begin{aligned}
&S.1 = true\\
&S.2 = NGD\\
&S.3 = S.2 \wedge \text{graph of the parent relation forms a tree} \wedge\\
&\qquad\qquad (\forall j,k :: root.j = root.k \wedge col.j = green \wedge par.j \in Adj.j \wedge\\
&\qquad\qquad\qquad (par.j = j \Rightarrow d.j = 0) \wedge (par.j \neq j \Rightarrow d.j = d.(par.j) + 1))\\
&S.4 = S.3 \ \wedge \ S_{MSR}
\end{aligned}$$

and show that for each i, $1 \leq i < 4$, $S.i$ converges to $S.(i+1)$ in program *MSR*. It follows that starting from an arbitrary state, program *MSR* reaches a state where $S.4$ and, hence, S_{MSR} hold.

A.2-1 To prove $S.1$ converges to $S.2$, consider the set of processes for which $NGd.j$ is violated. When a process executes action *MSR9*, the cardinality of this set decreases by 1. The cardinality of the set never increases. Thus, eventually, the cardinality of the set reduces to zero, and hence $S.2$ holds. Also, it is easy to observe that $S.2$ is closed under the execution of program *MSR*.

A.2-2 To prove that $S.2$ converges to $S.3$, we use a proof identical to the convergence proof in Arora and Gouda's reset paper [AG94]. For brevity, we omit this proof here. We merely note that in a state that satisfies $S.3$, since the graph of the parent relation forms a tree and the predicate $NGd.j$ holds for all processes, actions *MSR4*–*MSR9* are disabled. Actions *MSR1*–*MS3* trivially preserve $S.3$. Thus, $S.3$ is closed under execution of program *MSR*. Also, if the set of processes forms a rooted tree and all processes are green, the predicate T_{TREE} holds trivially.

A.2-3 To prove $S.3$ converges to $S.4$, we need to prove that the program converges to a state where the predicates $I1$–$I8$ hold. We first prove that the program reaches a state where k, the root of the graph of the parent tree, is in the *normal* state. We then prove that in such a state, the predicates $I1$–$I8$ hold.

A.2-4 As mentioned above, in a state satisfying $S.3$, only actions *MSR1*–*MSR3* may be enabled. If action *MSR1* is executed, then k is in the *normal* state. Hence, to prove that eventually the program reaches a state where k is in the *normal* state, we can assume that action *MSR1* is not executed.

A.2-5 Consider the variant function $|\{j : st.j = reset\}| + 2|\{j : sn.j \neq sn.k\}|$. When a process executes either the propagation or the completion action, the value of this function decreases. Thus, eventually the system will reach a state where the value of this function is zero, in which case k is in the *normal* state.

A.2-6 We now prove that when k is in the *normal* state, the predicates $I1$–$I8$ hold. From NGD, we have that all processes are in the *normal* state with the sequence number $sn.k$. Hence, the following predicate is *true* for each process j:

$$(\forall j :: sn.j = sn.k \ \wedge \ st.j \neq reset)$$

A.2-7 Since k is not in the reset state, predicates $I1, I5$, and $I6$ are satisfied. Since all processes have the same sequence number, the predicate $(\forall m : m \in X.l : sn.m = sn.k)$ holds for all processes l, i.e., predicates $I2$ and $I3$ are satisfied. Since $sn.j$ is equal to $sn.k$ and j is in the *normal* state, predicates $I4, I7$, and $I8$ are also satisfied. Thus, when k is in the *normal* state, the predicate S_{MSR} holds.

A.2-8 It now follows that starting from an arbitrary state, the program execution converges to a state where the predicate S_{MSR} holds.

A.2-9 To prove that *MSR* is masking tolerant to fail-stops and repairs, we show that newly added actions *MSR7*, *MSR8*, and *MSR9* do not interfere with the actions of program *MR*. Toward this end, we prove the following lemma.

Proof of Theorem 4 □

Lemma 6 *Actions MSR7, MSR8, and MSR9 preserve S_{MR}, and do not execute indefinitely after program MSR reaches a state in S_{MR}.*

Proof of Lemma 6 From the above proof, eventually the program reaches a state where $S.3$ is satisfied. In a state where $S.3$ is satisfied, *MSR7*–*MSR9* are disabled. Since $S.3$ is closed in *MSR*, these actions remain disabled. Thus, actions *MSR7*–*MSR9* do not execute indefinitely.

A.2-10 Since action *MSR7* does not update any variable in S_{MR} and *MSR9* is disabled in S_{MR}, these two actions trivially preserve S_{MR}. Observe that when j executes action *MSR8*, the effect is equivalent to a fail-stop of j followed by an instantaneous repair of j. And, we know that S_{MR} is preserved under fail-stop and repair faults. Thus, *MSR8* also preserves S_{MR}.

Proof of Lemma 6 □

A.3 Refinement to Low Atomicity

A.3-1 In program *MR*, the actions *MR3*, *MR4*, and *MR6* of process j update the $new.(par.j)$ variable of $par.j$ simultaneously with the variables of j. We now refine program *MR* so that in each action a process updates only its own variables. The refinement is made possible by the fact that the parent of j cannot complete its reset wave until j completes its reset wave. Hence, $new.(par.j)$ can be updated after the variables of j have been updated.

A.3-2 In the refined version of *MR*, for each of its neighbors k, j maintains a variable $new.j.k$ as a local copy of the nonlocal variable $new.k$. Whenever j needs to update the *new* value of k, j updates the value of $new.j.k$. Process k asynchronously reads $new.j.k$ and updates $new.k$. More specifically, *MR* is modified as follows (the modification to *MSR* is analogous):

1. Add a variable $new.j.k$ in j for every neighbor k.
2. Replace the statements $new.k := m$ by $new.j.k := m$.
3. Add the following two actions, the first to k and the second to j:
   - $new.j.k > 0 \ \wedge \ new.j.k > new.k \quad \longrightarrow new.k := new.j.k$
   - $new.j.k > 0 \ \wedge \ new.j.k \leq new.k \quad \longrightarrow new.j.k := 0$
4. Replace action *MR3* of j as follows:
   - $(\forall j : j \in Adj.j : new.j.k = 0) \ \wedge$ guard $\longrightarrow$ statement

   where *guard* and *statement* denote the guard and statement of action *MR3*, respectively.

A.3-3 With this refinement, the predicates $I1$ and $I7$ of the invariant S_{MR} change as follows:

$$\begin{array}{lll} I1' = & I1 \;\vee\; new.j.(par.j) > 0 & \\ I7' = & (new.j = 0) \quad \Rightarrow & \quad (sn.j \neq sn.k \;\Rightarrow\; pc.j.k = \phi) \\ & & \wedge\; (sn.j = sn.k \;\Rightarrow\; pc.j.k \in ch.j \\ & & \vee\; (\exists l : l \in Adj.j : new.l.j > 0)) \end{array}$$

A.3-4 And the invariant S_{MR} changes as follows:

$$\begin{array}{ll} S'_{MR} = (NGD \;\wedge\; T_{TREE} \wedge & (\forall j, l : j \in KERN.k \;\wedge\; l \in Adj.j \\ & \wedge\; root.j = root.l = k : \\ & I1' \;\wedge I2 \;\wedge I3 \;\wedge I4 \;\wedge I5 \;\wedge I6 \;\wedge I7' \;\wedge I8)) \end{array}$$

A.3-5 Finally, in the fail-stop action of program *MR*, the state of all of its neighbors is updated when process j fail-stops. This action can be easily refined so that each neighbor asynchronously updates its own state when it detects that j has fail-stopped. Thus, in the refined program, each process updates only its own state.

A.4 An Illustration of Tree Correction

A4.1-1 We illustrate how the tree maintenance program constructs a tree in the presence of fail-stop and repair faults with an example. Let the initial state be as shown in Figure 12a. In this state, the invariant of the tree program holds, the processes 1–4 are up, and they form a tree rooted at process 4. (The arrows define the parent relation.) Now, consider the following computation.

- Process 4 fail-stops and process 5 repairs (see Figure 12b).

- Since the parent of process 2 has failed, 2 changes its color to red. And, since 5 has no children, 5 resets its color to green (Figure 12c).

- Since the color of process 2—the parent of 1—is red, 1 sets its color to red (Figure 12d).

- Since 1 has no children, 1 changes its color to green and sets its root value to 1. Also, since the root value of 3 is smaller than that of 5, 3 changes its parent to 5 and copies its root value (Figure 12e).

- Since 2 has no children, 2 changes its color to green and sets its root value to 2 (Figure 12f).

- Finally, since 3 has a higher root value, 1 and 2 both change their parent to 3. The resulting state satisfies the invariant, and the tree is rooted at 5 (Figure 12g).

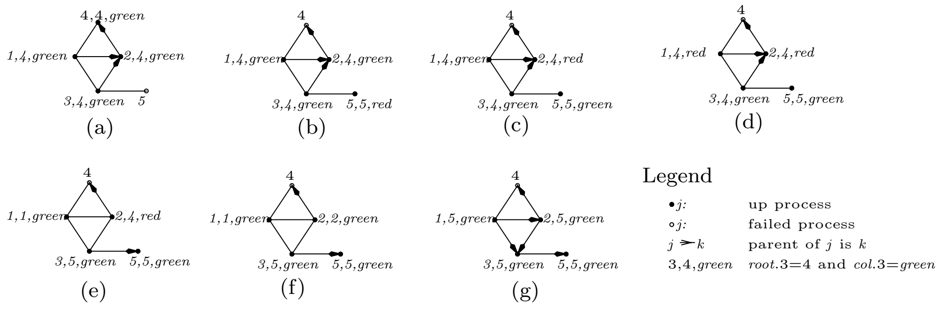


Figure 12: Tree correction

A.5 Notation

Symbols	
p $F, F1, F2$	Program Faults
$D, S, T, I1-I8$ R, NR, MR, MSR, APP $R1-R3, NR1-NR6, MR1-MR6$ $MSR1-MSR9, APP1-APP9$	State predicates Programs Actions

Variables	Domain	Read As
$st.j$	{reset, normal}	State
$sn.j$	{0, 1}	Sequence number
$par.j$	Process ID	Parent
$col.j$	{green, red}	Color
$root.j$	Process ID	Root
$new.j$	{0, 1, 2}	New
$d.j$	$\{1...K\}$	Distance
$inc.j$	{0, 1, 2}	Incarnation number
$ares.j$	Boolean	Application result

Propositional Connectives (in decreasing order of precedence)	
$\neg$	Negation
$\wedge, \vee$	Conjunction, disjunction
$\Rightarrow, \Leftarrow$	Implication, consequence
$\equiv, \not\equiv$	Equivalence, inequivalence

First-Order Quantifiers	
$\forall, \exists$	Universal, existential

Acknowledgment of support: Research supported in part by NSF grant CCR-93-08640 and OSU grant 221506.

References

[AG91] Y. Afek and E. Gafni. Bootstrap network resynchronization. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 295–307, New York, 1991. ACM.

[AG93] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[AG94] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[AH93] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *WDAG93: Distributed Algorithms 7th International Workshop Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188, Berlin, 1993. Springer-Verlag.

[AK98a] A. Arora and S. S. Kulkarni. Component-based design of multitolerance. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[AK98b] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6), June 1998.

[AO94] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC94: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 254–263, New York, 1994. ACM.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91: Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, Los Alamitos, CA, 1991. IEEE.

[APSVD94] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilizing by local checking and global reset. In *WDAG94: Distributed Algorithms 8th International Workshop Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339, Berlin, 1994. Springer-Verlag.

[Aro94] A. Arora. Efficient reconfiguration of trees: A case study in the methodical design of nonmasking fault-tolerance. In *Proceedings* *of the Third International Symposium on Formal Techniques in Real Time and Fault-Tolerance*, pages 110–127, 1994. Also to appear in *Science of Computer Programming.*

[AS87] B. Alpern and F. B. Schneider. Proving Boolean combinations of deterministic properties. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 131–137, Ithaca, NY, 1987. IEEE Computer Society.

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997(4), December 1997.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs, NJ, 1976. Prentice-Hall.

[GM91] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[Gri81] D. Gries. *The Science of Programming.* Berlin, 1981. Springer-Verlag.

[JV96] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. In *ACM Symposium on Principles of Distributed Computing*, New York, 1996. ACM.

[Tel89] G. Tel. Structure of distributed algorithms. PhD Thesis, University of Utrecht; also published by Cambridge University Press, 1989.

[TM94] S. Tsang and E. Magill. Detecting feature interactions in the intelligent network. *Feature Interactions in Telecommunications Systems II*, 1994.

[Var93] G. Varghese. Self-stabilization by local checking and correction. PhD Thesis, Massachusetts Institute of Technology; also Technical Report LCS-TR-583, Cambridge, MA, 1993.