Designing Masking Fault-tolerance via Nonmasking Fault-tolerance

Anish Arora *Sandeep S. Kulkarni*[1]

Abstract

Masking fault-tolerance guarantees that programs continually satisfy their specification in the presence of faults. By way of contrast, nonmasking fault-tolerance does not guarantee as much: it merely guarantees that when faults stop occurring, program executions converge to states from where programs continually (re)satisfy their specification.

We present in this paper a component based method for the design of masking fault-tolerant programs. In this method, components are added to a fault-intolerant program in a stepwise manner, first, to transform a fault-intolerant program into a nonmasking fault-tolerant one and, then, to enhance the fault-tolerance from nonmasking to masking. We illustrate the method by designing programs for agreement in the presence of Byzantine faults, data transfer in the presence of message loss, triple modulo redundancy in the presence of input corruption, and mutual exclusion in the presence of process fail-stops. These examples also serve to demonstrate that the method accommodates a variety of fault-classes, it provides alternative designs for programs usually designed with extant design methods, and it offers the potential for improved masking fault-tolerant programs.

Keywords: masking and nonmasking fault-tolerance, component based design, correctors, detectors, stepwise design formal methods, distributed systems

[1] Authors' Address: Dreese 395, Dept. of Comp. Sc., The Ohio State University, Columbus, OH 43210
Tel: (614)292-1836 , Fax: (614)292-2911 , Email: {anish, kulkarni}@cis.ohio-state.edu
This research was supported in part by NSF Grant CCR-93-08640, NSA Grant MDA904-96-1-1011, and OSU Grant 221506

1 Introduction

In this paper, we present a new method for the design of "masking" fault-tolerant systems [1-4]. We focus our attention on masking fault-tolerance because it is often a desirable —if not an ideal— property for system design: Masking the effects of faults ensures that a system always satisfies its problem specification and, hence, users of the system always observe expected behavior. By the same token, when the users of the system are other systems, the design of these other systems becomes simpler.

To motivate the design method, we note that designers of masking fault-tolerant systems often face the potentially conflicting constraints of maximizing reliability while minimizing overhead. As a result, designers reject methods that yield complex designs, since the complexity itself may result in reduced reliability. Moreover, they reject methods that yield inefficient implementations, since system users are generally unwilling to pay a significant cost in price or performance for the sake of masking fault-tolerance. Therefore, a key goal for our method is to yield wellstructured —and hence more reliable— systems, while still offering the potential for efficient implementation. Other goals of the method include the ability to deal with a variety of fault-classes and the ability to provide designs —albeit alternative ones— for systems which are typically designed by using classical methods for masking fault-tolerance such as replication, exception handling, and recovery blocks.

With these goals in mind, our method is based on the use of components that add tolerance properties to a fault-intolerant system. Component based design yields systems that are wellstructured and, moreover, maintainable, reusable, and extensible. It also divides the design complexity into that of designing relatively simpler components and that of adding the components to the fault-intolerant system. And, by focusing attention on the efficient implementation of the components themselves, it offers the potential for the resulting system to be efficiently implemented.

To manage the complexity of adding components to a system, the method proceeds in a stepwise fashion. More specifically, instead of adding the components, which will satisfy the problem specification in the presence of faults, all at once, the method adds components in two stages. In the first stage, the method merely adds components for nonmasking fault-tolerance. By nonmasking fault-tolerance we mean that, when faults stop occurring, the system execution eventually reaches a "good" state from where the system continually (re)satisfies its problem specification. Viewing the problem specification as consisting of a safety specification and a liveness specification (see Section 2 for a precise description of safety and liveness specifications), it follows that in a nonmasking fault-tolerant system the liveness specification is satisfied starting from every state, but the safety specification is guaranteed only starting from good states and not necessarily from other states that are reached in the presence of faults. Therefore, in the second stage, the fault-tolerance of the system is enhanced from nonmasking to masking, by adding components that ensure that the safety specification is satisfied in all states reached in the presence of faults.

We call the components added in the first stage *correctors* and those added in the second stage *detectors*. Efficient implementation of correctors and detectors is important, as noted above, for offering the potential for efficient masking fault-tolerant implementations.

As in any component based design, to prove the correctness of the resulting composite system, we need to ensure that the components do not interfere with each other, i.e., they continue to accomplish their task even if they are executed concurrently with the other components. To this end, in the first stage, we ensure that fault-intolerant system and the correctors added to it do not interfere with each other. And, in the second stage, we ensure that the resulting nonmasking fault-tolerant system and the detectors added to it do not interfere with each other.

To demonstrate that our method accommodates a variety of fault-classes, we use the method to design programs that are masking fault-tolerant to Byzantine faults, input corruption, message loss, and fail-stop failures. More specifically, we design: (1) a Byzantine agreement program whose processes are subject to Byzantine faults; (2) an alternating-bit data transfer program whose channel messages may be lost; (3) a triple modulo redundancy (TMR) program whose inputs may be corrupted; and (4) a new token-based mutual exclusion program whose processes may fail-stop in a detectable manner.

The TMR and Byzantine agreement examples also serve to provide alternative designs for programs usually associated with the method of replication. The alternating-bit protocol example serves to provide an alternative design for a program usually associated with the method of exception handling or that of rollback recovery. And, in contrast to the TMR example where attention is not focussed on the addition of efficient components (i.e., the components are more redundant than need be), our case study in mutual exclusion serves to demonstrate that the method enables the design of improved programs.

We proceed as follows. First, in Section 2, we recall a formal definition of programs, faults, and what it means for programs to be masking or nonmasking fault-tolerant. Then, in Section 3, we present our two-stage method for design of masking fault-tolerance. Next, in Section 4, we illustrate the method by designing standard masking fault-tolerant programs for Byzantine agreement, data transfer, and TMR. In Section 5, we present our case study in the design of masking fault-tolerant token-based mutual exclusion. Finally, we compare our method with extant methods for designing masking fault-tolerant programs and make concluding remarks in Section 6.

2 Programs, Faults, and Masking and Nonmasking Tolerances

In this section, we recall formal definitions of masking and nonmasking fault-tolerance of programs [5], in order to characterize a relationship between these two tolerance types and to motivate our design method, which is presented in Section 3.

Programs. A program p is defined recursively to consist of a (possibly empty) program q, a set of "superposition variables", and a set of "superposition actions". The superposition variables of p are disjoint from the remaining variables of p, namely the variables of q. Each superposition action of p has one of two forms:

$$\langle \text{name} \rangle :: \langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle \quad \text{, or}$$
$$\langle \text{name} \rangle :: \langle \text{action of } q \rangle \parallel \langle \text{statement} \rangle$$

A guard is a boolean expression over the variables of p. Thus, the evaluating a guard may involve accessing the variables of q. Note that there is a guard in each action of p: in particular, the guard of the actions of the second (i.e., $\|$) form is that of the corresponding action of q. A statement is an atomic, terminating update of zero or more of the superposition variables of p. Thus, the superposition actions of the first form do not update the variables of q, whereas those of the second may since they are based on an action of q. Note that, since statements of p do not update the variables of q, the only actions of p that update the variables of q are the actions of q.

Thus, programs are designed by superposition of variables and actions on underlying programs [6]. Superposition actions may access, but not update, the underlying variables, whereas underlying actions may not access or update the superposition variables. Operationally speaking, the superposition actions of the first form execute independently (asynchronously) of other actions and those of the second form execute in parallel (synchronously) with the underlying action they are based upon.

State. A state of a 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. An action of p is enabled at a state iff its guard is true at that state. We use the term "S state" to denote a state that satisfies a state predicate S.

Closure. An action "preserves" a state predicate S iff in any state where S holds and the action is enabled, executing all of the statements in the action instantaneously in parallel yields a state where S holds. S is "closed" in a set of actions iff each action in that set preserves S.

It follows from this definition that if S is closed in (the actions of) p then executing any sequence of actions of p starting from a state where S holds yields a state where S holds.

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 all of its statements are instantaneously executed in parallel. (Recall that actions of the second form consist of multiple statement composed in parallel.) 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. Note that the set of computations is suffix closed.

Invariant. An invariant of p is a state predicate S such that S is closed in p and every computation of p starting from a state in S satisfies the problem specification of p. Informally, the invariant of a program includes the states reached in the fault-free execution of p. Note that p may have multiple invariants.

Informally, the problem specification of p consists of a safety specification and a liveness specification [7]. A safety specification identifies a set of "bad" finite computation prefixes that should not appear in any program computation. Dually, a liveness specification identifies a set of "good" computation suffixes such that every computation has a suffix that is in this set. Hence, a program computation satisfies the problem specification iff it satisfies the safety specification and the liveness specification in that specification.

(Remark: Our definition of liveness is stronger than Alpern and Schneider's definition [7]: the two definitions become identical if the liveness specification is fusion closed; i.e., if computations $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ satisfy the liveness specification then computations $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$ also satisfy the liveness specification, where α, β are finite computation prefixes, γ, δ are computation suffixes, and x is a program state.)

Techniques for the design of an invariant of the program have been articulated by Dijkstra [8], using the notion of auxiliary variables, and by Gries [9], using the heuristics of state predicate ballooning and shrinking. Techniques for the mechanical calculation of an invariant predicate have been discussed by Alpern and Schneider [10].

Convergence. A state predicate Q "converges to" R in p iff Q and R are closed in p and, starting from any state where Q holds, every computation of p has a state where R holds. Note that the converges-to relation is transitive.

Lemma 2.1. If Q converges to R in p and every computation of p starting from states where R holds satisfies a liveness specification, then every computation of p starting from states where Q holds satisfies that liveness specification.

Proof. Consider a computation c of p starting from a Q state. Since Q converges to R in p, c has a suffix c_1 starting from an R state. Since every computation of p starting from an R state satisfies the liveness specification, c_1 has a suffix c_2 that is identified by the liveness specification. Since c_2 is also a suffix of c, it follows that c also satisfies the liveness specification. Thus, every computation of p starting form a Q state satisfies that liveness specification. $\square$

Faults. The faults that a program is subject to are systematically represented by actions whose execution perturbs the program state. We emphasize that such representation is possible notwithstanding the type of the faults —be they stuck-at, crash, fail-stop, omission, timing, performance, or Byzantine—, their nature —be they permanent, transient, or intermittent—, their observability —be they detectable or not—, or their repairability —be they correctable or not.

In some cases, such representation of faults introduces auxiliary variables. For example, to represent a fail-stop fault as a state perturbation, we introduce an auxiliary variable *up*. Each action is restricted to execute only when *up* is true. The fail-stop fault is represented by the action that changes *up* from true to false, thereby disabling all the actions in a detectable manner. Moreover, the repair of a fail-stopped program can be represented by the fault action that changes *up* from false to true. In other words, fail-stop and repair faults are respectively represented by the fault actions:

$$\begin{array}{llll} \textit{Fail-stop} :: & up & \longrightarrow & up := false \\ \textit{Repair} :: & \neg up & \longrightarrow & up := true \end{array}$$

To represent a Byzantine fault as a state perturbation, we introduce an auxiliary variable b. The specified actions of the program are restricted to execute only when b is false, i.e., the program is non-Byzantine. If b is true, i.e., the program is Byzantine, the program is allowed to execute actions that can change its state arbitrarily. Thus, the Byzantine fault is represented by the action that changes b from false to true, thereby enabling the program to enter a mode where it executes actions that change its state arbitrarily. In other words, Byzantine fault is represented by the fault action:

$$\mathit{Byzantine} :: \neg b \quad \longrightarrow \quad b := \mathit{true}$$

Fault-span. A fault-span of program p for a fault-class F is a predicate T such that T is closed in p and F. Informally, the fault-span includes the set of states that p reaches when executed in the presence of actions in F. Note that p may multiple have fault-spans for F.

If program p with invariant S is subject to a fault-class F, the resulting states of p may no longer satisfy S. However, these states satisfy fault-span of p, say T. Moreover, every state in S also satisfies T.

Fault-Tolerance: Masking and Nonmasking. We are now ready to give a formal definition of fault-tolerance [5]. Instantiations of this definition yield definitions of masking and nonmasking fault-tolerance.

Let p be a program, F be a set of fault actions, and S be an invariant of p. We say that "p is F-tolerant for S" iff there exists a state predicate T of p such that the following three conditions hold:

- Inclusion: $T \Leftarrow S$
- Closure: T is closed in p and F
- Convergence: T converges to S in p

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, but executing an action in F may yield a state where S does not hold. Nonetheless, the following three facts are true about this last state : (i) T, the fault-span, holds, (ii) subsequent execution of actions in p and F yields states where T holds, and (iii) 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.

When the definition is instantiated so that the fault-span T is identical to the invariant S, we get that p is masking F-tolerant for S. And when the definition is instantiated so that T differs from S, we get that p is nonmasking F-fault-tolerant for S.

In the rest of this paper, the predicate S_p denotes an invariant of program p. Moreover, the predicate T_p denotes a fault-span predicate for a program p that is F-tolerant for S_p. Finally, when the fault-class F is clear from the context, we omit mentioning F; thus, "masking tolerant" abbreviates "masking F-tolerant".

3 A Method for Designing Masking Tolerance

From the definitions in the previous section, we observe that masking and nonmasking fault-tolerance are related as follows.

Theorem 3.1. For a program p,

If	there exists S_p and T_p such that p is nonmasking F-tolerant for S_p *and* every computation of p starting from a state where T_p holds satisfies the safety specification of p
Then	there exists S_p such that p is masking F-tolerant for S_p.

Proof. Let S_{np}, T_{np} be state predicates satisfying the antecedent. Then every computation of p starting from a state where S_{np} holds satisfies its problem specification, and starting from a state where T_{np} holds satisfies its safety specification. It follows from Lemma 2.1 that every computation of p starting from a T_{np} state satisfies its problem specification. Thus, choosing $S_p = T_{np}$ satisfies the consequent. □

The Method. Theorem 3.1 suggests that an intolerant program can be made masking tolerant in two stages: In the first stage, the intolerant program is transformed into one that is nonmasking tolerant, for the invariant and fault-span say S_{np} and T_{np} respectively. In the second stage, the tolerance of resulting program is enhanced from nonmasking to masking, as follows. The nonmasking tolerant program is transformed so that every computation upon starting from a state where T_{np} holds, in addition to eventually reaching a state where S_{np} holds, also satisfies the safety specification of the problem at hand.

We address the details of both stages, next.

Stage 1. For a fault-intolerant program, say p, the problem specification is satisfied by computations of p that start at a state where its invariant holds but not necessarily by those that start at a state where its fault-span holds. Hence, to add nonmasking tolerance to p, a program component is added to p that restores it from fault-span states to invariant states.

We call the program component added to p for nonmasking tolerance a *corrector*. Wellknown examples of correctors include reset procedures, rollback-recovery, forward recovery, error correction codes, constraint (re)satisfaction, exception handlers, and alternate procedures in recovery blocks. The design of correctors has been studied extensively, and we do not discuss it further here, except to note that correctors can be designed in a stepwise and hierarchical fashion.

Specifically, a large corrector can be designed by parallel and/or sequential composition of small correctors. One simple parallel composition strategy is to superpose small correctors on others. An alternative composition strategy, due to Arora, Gouda, and Varghese [11], is to order the small correctors in a linear manner (or, more generally, a well-founded manner) such that each corrector does not interfere with the recovery task of the correctors lower than it in the chosen ordering. For a detailed discussion of corrector compositions, we refer the reader to [12].

Stage 2. For a nonmasking program, say np, even though the problem specification is satisfied *after* computations of np converge to invariant states, the safety specification need not be satisfied in all computations of np that start at fault-span states. Therefore, in the second stage, we restrict the actions of np so that the safety specification is preserved *during* the convergence of computations of np to invariant states. By Theorem 3.1, it follows that the resulting program is masking tolerant.

To see that restriction of actions of np is sufficient for preserving safety during convergence, recall that the safety specification essentially rules out certain finite prefixes of computation of np. Now consider any prefix of a computation of np that is not ruled out by the safety specification: Execution of an action following this prefix increases the length of the computation prefix by one. As long as the elongated prefix is not one of the prefixes ruled out by the safety specification, safety is not violated. In other words, it suffices that whenever an action is executed, the resulting prefix be one that is not ruled out by the safety specification.

It follows that there exists, for each action of np, a set of computation prefixes for which execution of that action preserves the safety specification. Assuming the existence of auxiliary state (which in the worst case would record the history of the computation steps), there therefore exists, for each action of np, a state predicate— which we call the *safe predicate* of that action —that is true in exactly those states where the execution of the action preserves safety.

The restriction of the actions of np so as to enhance the tolerance of np to masking can now be stated precisely. Each action of np is restricted to execute only when its safe predicate holds. Moreover, for each action of np, the detection of its safe predicate may require the addition of a program component to np.

We call a program component added to np for detecting that the safe predicate of an action holds a *detector*. Wellknown examples of detectors include snapshot procedures, acceptance tests, error detection codes, consistency checkers, watchdog programs, snooper programs, and exception conditions. Analogous to the compositional design of large detectors, large detectors can be designed in a stepwise and hierarchical fashion, by parallel and/or sequential composition of small detectors.

Thus, in sum, the second stage adds at most one detector per action of np and restricts each action of np to execute only when the detector of that action witnesses that its safe predicate holds. Before concluding our discussion of this stage, we make two experimental observations about its application in practice:

1. The safe predicate of several program actions is the trivially detected state predicate *true*; and
2. The safe predicate of most other actions requires only simple detector components, which introduce only little additional state to check the safe predicate.

Observation (1) follows from the fact that the actions of masking tolerant programs can be conceptually characterized as either "critical" or "noncritical", with respect to the safety specification. Critical actions are those actions whose execution in the presence of faults can violate the safety specification; hence, only they require non-trivial safe predicates. In other words, the safe predicate of all non-critical actions is merely *true*.

For example, in terminating programs, e.g. feed-forward circuits or database transactions, only the actions that produce an output or commit a result are critical. In reactive programs, e.g. operating systems or plant controllers, only the actions that control progress while maintaining safety are critical. In the rich class of "total" programs for distributed systems [13], e.g. distributed consensus, infima finding, garbage collection, global function computation, reset, routing, snapshot, and termination detection, only the "decider" actions that declare the outcome of the computation are critical.

Observation (2) follows from the fact that conventional specification languages typically yield safety specifications that are tested on the current state only or on the current computation step only; i.e., the set of finite prefixes that their safety specifications rule out can be deduced from the last or their last two states of computation prefixes. Thus, most safety specifications in practice do not require maintenance of unbounded "history" variables for detection of the safe predicates of each action.

Verification obligations. The addition of corrector and detector components as described above may add variables and actions to an intolerant program and, hence, the invariant and the fault-span of the resulting program may be different from those of the original program. The addition of corrector and detector components thus creates some verification obligations for the designer.

Specifically, when a corrector is added to an intolerant program, the designer has to ensure that the corrector actions and the intolerant program actions do not interfere with each other. That is, even if the corrector and the fault-intolerant program execute concurrently, both accomplish their tasks: The corrector restores the intolerant program to a state from where the problem specification of the intolerant program is (re)satisfied. And starting from such a state the intolerant program satisfies its problem specification.

Similar obligations are created, when detectors are added to a nonmasking program. Even if the detectors and the nonmasking program are executed concurrently, the designer has to ensure that the detector components and the components of the nonmasking program all accomplish their respective tasks.

Another set of verification obligations is due to the fact that the corrector and detector components are themselves subject to the faults that the intolerant program is subject to. Hence, the designer is obliged to show that these components accomplish their task in spite of faults. More precisely, the corrector tolerates the faults by ensuring that when fault actions stop executing it eventually restores the program state as desired. In other words, the corrector is itself nonmasking tolerant to the faults. And each detector tolerates the faults by never falsely witnessing its detection predicate, even in the presence of the faults. In other words, each detector is itself masking tolerant to the faults. As can be expected, our two-stage design method can itself be used to design masking tolerance in the detectors, if their original design did not yield masking tolerant detectors.

Adding detectors components by superposition. One way of simplifying the verification obligations is to add components to a program by superposing them on the program: if a program p is designed by a superposition on the program q, then it is trivially true that p does not interfere with q (although the converse need not be true, i.e., q may interfere with p).

In particular, superposition is wellsuited for the addition of detector components to a nonmasking tolerant program, np, in Stage 2, since detectors need only to read (but not update) the state of np. (It is for this reason that we've stated the definition of programs in Section 2 in terms of superposition.) Thus, the detectors do not interfere with the tasks of the corrector and the underlying program components in np.

When superposition is used, the verification of the converse obligation, i.e. that np does not interfere with the detectors, may be handled as follows. Ensure that the corrector in np terminates after it restores np to an invariant state and that as long as it has not terminated it prevents the detectors from witnessing their safe predicate. Aborting the detectors during the execution of the corrector guarantees that the detectors never witness their safe predicate incorrectly, and the eventual termination of the corrector guarantees that eventually detectors are not prevented from witnessing their safe predicate.

More specifically, the simplified verification obligations resulting from superposition are explained from Theorems 3.2 and 3.3. Let program p be designed by superposition on q such that $T_p \Rightarrow T_q$ and $S_p \Rightarrow S_q$.

Theorem 3.2. If q is nonmasking F-tolerant for S_q, then T_p converges to S_q in p. □

Theorem 3.3. If q is nonmasking F-tolerant for S_q, then

$$(T_p \wedge S_q \text{ converges to } S_p \text{ in } p) \Rightarrow (T_p \text{ converges to } S_p \text{ in } p)$$

Proof: Since q is nonmasking fault-tolerant, T_q converges to S_q in q. Since p is designed by a superposition on q, it follows that ($T_p \wedge T_q$ converges to $T_p \wedge S_q$). Since the converges-to relation is transitive and ($T_p \wedge S_q$ converges to $S_p \wedge S_q$), it follows that ($T_p \wedge T_q$ converges to $S_p \wedge S_q$), i.e., T_p converges to S_p in p. □

Theorems 3.2 and 3.3 imply that if p is designed by superposition on a nonmasking tolerant program q, then to reason about p, it suffices to assume that q always satisfies its invariant S_q, even in the presence of faults. For a discussion of alternative strategies for verifying interference freedom, we refer the reader to [12].

4 Design Examples

In this section, we demonstrate that our method is wellsuited for the design of classical examples of masking tolerance, which span a variety of fault-classes. Specifically, our examples of masking tolerance achieve Byzantine agreement in the presence of Byzantine failure, data transfer in the presence of message loss in network channels, and triple modulo redundancy (TMR) in the presence of input corruption.

Notation. For convenience in presenting these designs, we will partition the actions of a program into "processes".

4.1 Example 1 : Byzantine agreement

Recall the Byzantine agreement problem: A unique process, the general, g, asserts a binary value $d.g$. Every process j in the system is required to eventually finalize its decision such that the following two conditions hold: (1) if g is non-Byzantine, the final decision reached by every non-Byzantine process is identical to $d.g$; and (2) even if g is Byzantine, the final decisions reached by all non-Byzantine processes are identical.

Faults corrupt processes permanently and undetectably such that the corrupted processes are Byzantine. It is wellknown that masking tolerant Byzantine agreement is possible iff there are at least $3f{+}1$ processes, where f is the number of Byzantine processes [14]. For ease of exposition, we will restrict our attention to the case where the total number of processes (including g) is 4 and, hence, f is 1.

As prescribed by our method, we will design the masking tolerant solution to the Byzantine agreement problem in two stages. Starting with an intolerant program for Byzantine agreement, we will first transform that program to add nonmasking tolerance, and subsequently enhance the tolerance to masking.

Intolerant Byzantine agreement. The following simple program suffices for agreement but not for tolerance to faults: Process g is assumed to have *a priori* finalized its decision $d.g$. Each process j other than g receives the value $d.g$ from process g and then finalizes its decision to that value. To this end, the program maintains two variables for each process j: a boolean $f.j$ that is true iff j has finalized its decision, and $d.j$ whose value denotes the decision of j.

The program has three actions for each process j. The first action, $IB1$, copies $d.g$ into the decision variable $d.j$: to denote that j has not yet copied $d.g$, we add a special value $\bot$ to the domain of $d.j$; thus, j copies $d.g$ only if $d.j$ is $\bot$. The second action, $IB2$, finalizes the decision j: if j has copied $d.g$, j finalizes its decision by truthifying $f.j$. These two actions are executed by j only if it is non-Byzantine. The third action, $IB3$, is executed by j only if it is Byzantine: this action nondeterministically changes $d.j$ to either 0 or 1 and $f.j$ to either *true* or *false*. Formally, the actions of the intolerant program, IB, are as follows:

$$
\begin{array}{lcl}
IB1 :: \neg b.j \ \wedge\ j{\neq}g \ \wedge\ d.j{=}\bot & \longrightarrow & d.j := d.g \\
IB2 :: \neg b.j \ \wedge\ j{\neq}g \ \wedge\ d.j{\neq}\bot & \longrightarrow & f.j := \mathit{true} \\
IB3 :: b.j & \longrightarrow & d.j, f.j := ?, ?
\end{array}
$$

Invariant. In program IB, when any non-Byzantine process finalizes its decision, $d.j \neq \bot$.

Moreover, if g is non-Byzantine, it has *a priori* finalized its decision, and the final decision of each non-Byzantine process is identical to $d.g$. Finally, if g is Byzantine, the intolerant program works correctly if it starts in a state where all processes have correctly finalized their decisions. Hence, the invariant of program IB is S_{IB}, where

$$
\begin{aligned}
S_{IB} = {} & (\forall j : \neg b.j : f.j \;\Rightarrow\; d.j \neq \perp) \quad \wedge \\
& (\neg b.g \;\Rightarrow\; (f.g \;\wedge\; d.g \neq \perp \;\wedge\; (\forall j : \neg b.j : d.j \neq \perp \;\Rightarrow\; d.j = d.g))) \quad \wedge \\
& (b.g \;\Rightarrow\; (\forall j, k : \neg b.j \wedge \neg b.k : f.j \;\wedge\; d.j = d.k))
\end{aligned}
$$

Remark. A formula $(\forall j : R.j : X.j)$ may be read as for all j (in this case j is a process) if $R.j$ is true then so is $X.j$. If $R.j$ is true, i.e., the predicate $X.j$ is true for all processes, we omit $R.j$. Similarly, a formula $(\exists j : R.j : X.j)$ may be read as there exists a process where both $R.j$ and $X.j$ are true. This notation is from [15]

Fault Actions. The faults in this example make one process Byzantine, provided that no other process is Byzantine. As discussed in Section 2, these faults would be represented by the following fault action at each j :

$$(\forall k :: \neg b.k) \quad \longrightarrow \quad b.j := true$$

Nonmasking tolerant Byzantine agreement. Program IB is intolerant because if g becomes Byzantine before all processes have finalized their decisions, g may keep changing its $d.g$ arbitrarily and hence the final decisions reached by the non-Byzantine processes may differ. We now add nonmasking tolerance to IB so that eventually the decisions reached by all non-Byzantine processes are identical.

Since IB eventually reaches a state where the decisions of all processes differ from $\perp$ (i.e., are 0 or 1), it follows that eventually the decisions of at least two of the three processes other than g will be identical. Hence, if all of these processes ensure that their decision is the same as that of the majority, the resulting program will be nonmasking tolerant.

Our nonmasking tolerant program consists of four actions for each process j: the first three are identical to the actions of IB, and the fourth action, $NB4$, changes the decision of j to the majority of the three processes. Formally, the actions of the nonmasking program, NB, are as follows:

$$
\begin{aligned}
& NB1 :: IB1 \\
& NB2 :: IB2 \\
& NB3 :: IB3 \\
& NB4 :: \underline{majdefined} \wedge d.j \neq \underline{maj} \quad \longrightarrow \quad d.j := \underline{maj}
\end{aligned}
$$

---

where,
$$
\begin{aligned}
\underline{majdefined} &\equiv (\exists j, k : j \neq k \;\wedge\; j \neq g \;\wedge\; k \neq g : d.j = d.k \;\wedge\; d.j \neq \perp) \\
\underline{maj} &= (majority\ j : j \neq g : d.j)
\end{aligned}
$$

Invariant and fault-span. As in program IB, in program NB, when any non-Byzantine process finalizes its decision, $d.j \neq \bot$. Also, if g remains non-Byzantine, all other non-Byzantine processes reach the same decision value as process g. Hence, the fault-span of NB, T_{NB} is:

$$\begin{aligned} T_{NB} = \; & (\forall j : \neg b.j : f.j \;\Rightarrow\; d.j \neq \bot) \quad \wedge \\ & (\neg b.g \;\Rightarrow\; (f.g \;\wedge\; d.g \neq \bot \;\wedge\; (\forall j : \neg b.j : d.j \neq \bot \;\Rightarrow\; d.j = d.g))) \end{aligned}$$

and the invariant, S_{NB}, is the same as that of IB, i.e.,

$$S_{NB} = S_{IB}$$

Enhancing the tolerance to masking. Program NB is not yet masking tolerant as a non-Byzantine j may first finalize its decision incorrectly, and only later correct its decision to that of the majority of the other processes. Hence, to enhance the tolerance of NB to masking, it suffices that j finalize its decision only when $d.j$ is the same as the majority.

The masking program thus consists of four actions at each process j: the three actions are identical to actions $NB1, NB3$ and $NB4$, and the fourth action, $MB2$ is restricted so that j finalizes its decision only when $d.j$ is the same as the majority. Formally, the actions of the masking program, MB, are as follows:

$$\begin{array}{lll} MB1 :: NB1 & & \\ MB2 :: \neg b.j \;\wedge\; d.j \neq \bot \wedge \; \underline{majdefined} \wedge d.j = \underline{maj} & \longrightarrow & f.j := true \\ MB3 :: NB3 & & \\ MB4 :: NB4 & & \end{array}$$

Invariant. The fault-span of the nonmasking program, T_{NB}, is implied by the invariant, S_{MB}, of the masking program. Also, in S_{MB}, j finalizes its decision only when $d.j$ is the same as that of the majority. Thus, S_{MB} is:

$$S_{MB} \;=\; T_{NB} \;\wedge\; (\forall j : \neg b.j : f.j \;\Rightarrow\; d.j = \underline{maj})$$

Theorem 4.1 The Byzantine agreement program MB is masking fault-tolerant for invariant S_{MB}.

4.2 Example 2 : Data transfer

Recall the data transfer problem: An infinite input array at a sender process is to be copied, one array item at a time, into an infinite output array at a receiver process. The sender and receiver communicate via a bidirectional channel that can hold at most one message in each direction at a time. It is required that each input array item be copied into the output array exactly once and in the same order as sent. Moreover, eventually the number of items copied by the receiver should grow unboundedly.

Data transfer is subject to the faults that lose channel messages.

As before, we will design the masking tolerance for data transfer in two stages. The resulting program is the wellknown *alternating-bit protocol*.

Intolerant program. Iteratively, a simple loop is followed: sender s sends a copy of one array item to receiver r. Upon receiving this item, r sends an acknowledgment to s, which enables the next array item to be sent by s and so on. To this end, the program maintains binary variables rs in s and rr in r; rs is 1 if s has received an acknowledgment for the last item it sent, and rr is 1 if the r has received an item but has not yet sent an acknowledgment.

The 0 or 1 items in transit from s to r are denoted by the sequence cs, and the 0 or 1 acknowledgments in transit from r to s are denoted by the sequence cr. Finally, the index in the input array corresponding to the item that s will send next is denoted by ns, and the index in the output array corresponding to the item that r last received is denoted by nr.

The intolerant program contains four actions, the first two in s and the last two in r. By $ID1$, s send an item to r, and by $ID2$, s receives an acknowledgment from r. By $ID3$, r receives an item from s, and by $ID4$, r sends an acknowledgment to s. Formally, the actions of the intolerant program, ID, are as follows (where $c1; c2$ denotes concatenation of sequences $c1$ and $c2$):

$$
\begin{array}{lcl}
ID1 :: rs=1 & \longrightarrow & rs, cs := 0, cs; \langle ns \rangle \\
ID2 :: cr \neq \langle\rangle & \longrightarrow & rs, cr, ns := 1, tail(cr), ns+1 \\
\\
ID3 :: cs \neq \langle\rangle & \longrightarrow & cs, rr, nr := tail(cs), 1, head(cs) \\
ID4 :: rr=1 & \longrightarrow & rr, cs := 0, cr; \langle nr \rangle
\end{array}
$$

Invariant. When r receives an item, $nr=ns$ holds, and this equation continues to hold until s receives an acknowledgment. When s receives an acknowledgment, ns is exactly one larger than nr and this equation continues to hold until r receives the next item. Also, if cs is nonempty, cs contains only one item, $\langle ns \rangle$. Finally, in any state, exactly one of the four actions is enabled. Hence, the invariant of program ID is, S_{ID}, where

$$
\begin{array}{l}
S_{ID} = ((rr=1 \ \vee\ cr \neq \langle\rangle) \ \Rightarrow\ nr=ns) \ \wedge\ ((rs=1 \ \vee\ cs \neq \langle\rangle) \ \Rightarrow\ nr=ns-1) \ \wedge \\
\qquad (cs=\langle\rangle \ \vee\ cs=\langle ns \rangle) \ \wedge\ (|cs| + |cr| + rs + rr \ = \ 1)
\end{array}
$$

Fault Actions. The faults in this example lose either an item sent from s to r or an acknowledgment sent from r to s. The corresponding fault actions are as follows:

$$
\begin{array}{lcl}
cs \neq \langle\rangle & \longrightarrow & cs := tail(cs) \\
cr \neq \langle\rangle & \longrightarrow & cr := tail(cr)
\end{array}
$$

Nonmasking tolerant program. Program ID is intolerant as it deadlocks when a fault loses an item or an acknowledgment. Hence, we add nonmasking tolerance to this fault by adding an action by which s detects that an item or acknowledgment has been lost and recovers ID by retransmitting the item.

Thus, the nonmasking program consists of five actions; four actions are identical to the actions of program ID, and the fifth action retransmits the last item that was sent. This action is executed when both channels, cs and cr, are empty, and rs and rr are both zero. In practice, this action can be implemented by waiting for a some predetermined timeout so that the source can be sure that either the item or the acknowledgment is lost, but we present only the abstract version of the action. Formally, the actions of the nonmasking program, ND, are as follows:

$ND1 :: ID1$
$ND2 :: ID2$
$ND5 :: cs = \langle\rangle \ \wedge\ cr = \langle\rangle \ \wedge\ rs = 0 \ \wedge\ rr = 0 \quad \longrightarrow \quad cs \ := \ cs; \langle ns \rangle$

$ND3 :: ID3$
$ND4 :: ID4$

Fault-span and invariant. If an item or an acknowledgment is lost, the program reaches a state where cs and cr are empty and rs and rr are both equal to zero. Also, even in the presence of faults, if cs is nonempty, it contains exactly the item whose index in the input array is $\langle ns \rangle$. Thus, the fault-span of the nonmasking program is

$$T_{ND} = (cs = \langle\rangle \ \vee\ cs = \langle ns \rangle) \quad \wedge \quad (|cs| + |cr| + rs + rr \ \leq\ 1)$$

and the invariant is the same as the invariant of ID, i.e.,

$$S_{ND} = S_{ID}$$

Enhancing the tolerance to masking. Program ND is not yet masking tolerant, since r may receive duplicate items if an acknowledgment from r to s is lost. Hence, to enhance the tolerance to masking, we need to restrict the action $ID3$ so that r copies an item into the output array iff it is not a duplicate.

Upon receiving an item, if r checks that nr is exactly one less than the index number received with the item, r will receive every item exactly once. Thus, we can enhance its tolerance to masking by adding such a check to program ND. However, this check forces the size of the message sent from the s to r to grow unboundedly. But we can exploit the fact that in ND, ns and nr differ by atmost 1, in order to simulate this check by sending only a single bit with the item as follows.

Process s adds one bit, bs, to every item it sends such that the bit values added to two consecutive items are different and the bit values added to an item and its duplicates are the same. Thus, to detect that a message is duplicate, r maintains a bit, br, that denotes the sequence number of the last message it received. It follows that an item received by r is a duplicate iff br is the same as the sequence number in that message.

The masking program consists of five actions. These actions are as follows:

$$
\begin{array}{lll}
MD1 :: rs\!=\!1 & \longrightarrow & rs, cs \;:=\; 0, cs; \langle ns, bs\rangle \\
MD2 :: cr\!\neq\!\langle\rangle & \longrightarrow & rs, cr, ns, bs \;:=\; 1, tail(cr), ns+1, bs \oplus 1 \\
MD5 :: cs\!=\!\langle\rangle \;\wedge\; cr\!=\!\langle\rangle \;\wedge & & \\
\qquad\quad rs\!=\!0 \;\wedge\; rr\!=\!0 & \longrightarrow & cs \;:=\; cs; \langle ns, bs\rangle \\
 & & \\
MD3 :: cs\!\neq\!\langle\rangle & \longrightarrow & \text{if } ((head(cs))_2 \!\neq\! br) \text{ then } nr, br \;:=\; (head(cs))_1, (head(cs))_2; \\
 & & cs, rr := tail(cs), 1; \\
MD4 :: rr\!=\!1 & \longrightarrow & rr, cs \;:=\; 0, cr; \langle nr, br\rangle
\end{array}
$$

Invariant. In any state reached in the presence of program and fault actions, if cs is nonempty, cs has exactly one item, $\langle ns, bs\rangle$. Also, bs is the same as (ns mod 2), br is the same as (nr mod 2), and exactly of the five actions is enabled. Finally, nr is the same as ns or nr is one less than ns. Thus, the invariant of the masking program is S_{MD}, where

$$
\begin{array}{rl}
S_{MD} = & (cs\!=\!\langle\rangle \;\vee\; cs\!=\!\langle ns, bs\rangle) \quad\wedge\quad (|cs| + |cr| + rs + rr \;\leq\; 1) \quad\wedge \\
 & bs\!=\!(ns \; mod \; 2) \quad\wedge\quad br\!=\!(nr \; mod \; 2) \quad\wedge\quad (nr\!=\!ns \;\vee\; nr = ns\!-\!1)
\end{array}
$$

Theorem 4.2. The alternating-bit program, MD, is masking tolerant for invariant S_{MD}.

4.3 Example 3 : TMR

Recall TMR: Three processes share an output, out. A binary value is input to $in.j$, for each of process j. It is required that the output be set to this binary value.

Faults corrupt the input value of any one of the three processes.

Intolerant TMR. In the absence of faults, it suffices that out be set to $in.j$, for any process j. Hence, the actions of program IR in each process j are as follows (where $out\!=\!\perp$ denotes that the output has not yet been set):

$$IR1 :: \quad out\!=\!\perp \quad \longrightarrow \quad out := in.j$$

Fault actions. In this example, the faults corrupt the input value $in.j$ of at most one process. They are represented by the following fault actions, one for each j (where k also ranges over the processes):

$$(\forall k :: in.j\!=\!in.k) \quad \longrightarrow \quad in.j :=?$$

Nonmasking TMR. Program IR is intolerant since out may be set incorrectly from a corrupted $in.j$. Therefore, to add nonmasking tolerance to IR, we add a corrector that eventually corrects out. Since at most one $in.j$ is corrupted, the correct output can differ from at most one $in.j$. Hence, if out differs from the $in.j$ of two processes, the corrector resets out to the $in.j$ value of those two. Thus, the nonmasking program, NR, consists of two actions in each process j: action $NR1$ is the same as $IR1$ and action $NR2$ is the corrector. Formally, these two actions are as follows (where $\oplus$ denotes modulo 3 addition):

$$
\begin{array}{lllll}
NR1 :: & out = \bot & & \longrightarrow & out := in.j \\
NR2 :: & out \neq in.j \ \wedge \ (in.j = in.(j \oplus 1) \ \ \vee \ \ in.j = in.(j \oplus 2)) & & \longrightarrow & out := in.j
\end{array}
$$

Enhancing the tolerance to masking. Program NR is not yet masking tolerant since out may be set incorrectly before being corrected. Therefore, to enhance the tolerance to masking, we restrict the action $NR1$ so that the output is always set to an uncorrupted $in.j$. A safe predicate for this restriction of action $NR1$ is $((in.j = in.(j \oplus 1) \ \ \vee \ \ (in.j = in.(j \oplus 2))$. Restricting action $NR1$ with this safe predicate yields a stronger version of action $NR2$, thus the resulting masking tolerant program MR consists of only one action for each j:

$$
MR1 :: \quad out \neq in.j \ \wedge \ (in.j = in.(j \oplus 1) \ \ \vee \ \ in.j = in.(j \oplus 2)) \quad \longrightarrow \quad out := in.j
$$

Invariant. In program MR, if out is equal to $in.j$ for some j, then there exists another process whose input value is the same as $in.j$. Hence, the invariant of program MR is, S_{MR}, where

$$
S_{MR} \ = \ (out = in.j \quad \Rightarrow \quad (out = in.(j \oplus 1) \ \vee \ out = in.(j \oplus 2))
$$

Theorem 4.3 The triple modulo redundancy program MR is masking tolerant for invariant S_{MR}.

5 Case Study : Mutual Exclusion

In this section, we design a new and improved masking tolerant solution for the mutual exclusion problem using our two-stage method. Recall the mutual exclusion problem: Multiple processes may each access their critical sections provided that at any time at most one process is accessing its critical section. Moreover, no process should wait forever to access its critical section, assuming that each process leaves its critical section in finite time.

We assume that the processes have unique integer ids. At any instant, each process is either "up" or "down". Only up processes can execute program actions. Actions executed by an up process j may involve communication only with up processes connected to j via channels. Channels are bidirectional.

A fault fail-stops one of the processes, i.e., renders an up process down. Fail-stops may occur in any (finite) number, in any order, at any time, and at any process as long as the set of up processes remain connected.

One class of solutions for mutual exclusions is based on tokens. In token-based solutions, a unique token is circulated between processes, and a process enters its critical section only if (but not necessarily if) it has the token. To ensure that no process waits forever for the token, a fair strategy is chosen by which if any process requests access to its critical section then it eventually receives the token. An elegant token-based program is independently due to Raymond [16] and Snepscheut [17]; this program uses a fixed tree to circulate the token.

The case study is organized as follows. In Section 5.1, we recall (an abstract version of) the intolerant mutual exclusion program of Raymond and Snepscheut. In Section 5.2, we transform this fault-intolerant program into a nonmasking tolerant one by adding correctors. Finally, in Section 5.3, we enhance the tolerance to masking by adding detectors. The resulting solution is compared with other masking tolerant token-based mutual exclusion solutions in the next section.

5.1 The Fault-Intolerant Program

The processes are organized in a tree. Each process j maintains a variable $P.j$, to denote the parent of j in this tree; a variable $h.j$, to denote the holder process of j which is a neighbor of j in the direction of the process with the token; and a variable $Request.j$, to denote the set of requests that were received from the neighbors of j in the tree and that are pending at j.

The program consists of three actions for each process, the first for making or propagating to the holder process a request for getting the token; the second for transmitting the token to satisfy a pending request from a neighbor; and the third for accessing the critical section when holding the token. The actions are as follows:

$IM1 ::$	$h.j \neq j \ \wedge \ j \notin Request(h.j)$	$\longrightarrow Request.(h.j) := Request.(h.j) \cup \{j\}$
$IM2 ::$	$h.k = k \ \wedge \ j \in Request.k \ \wedge$ $(P.j = k \ \vee \ P.k = j)$	$\longrightarrow h.k, h.j := j, j;$ $Request.k := Request.k - \{j\}$
$NM3 ::$	$h.j = j$	$\longrightarrow$ access critical section

These actions maintain the holder relation so that it forms a directed tree rooted at the process that has the token. The holder relation, moreover, conforms to the parent tree; i.e., if k is the holder of j then j and k are adjacent in the tree.

$$
\begin{aligned}
S_{IM} = \ & h.j \ \in \ (\{j, P.j\} \cup ch.j) \ \wedge \\
& (j \neq P.j \quad \Rightarrow \quad ((h.j = P.j \ \vee \ h.(P.j) = j) \quad \wedge \quad \neg(h.j = P.j \ \wedge \ h.(P.j) = j)) \quad \wedge \\
& \text{the graph of the parent relation forms a tree}
\end{aligned}
$$

Remark. Note that action $IM1$ is executed either to request a token for j itself or to propagate the request of a descendent process of j in the holder tree. Therefore, upon receiving a token, j may decide to itself enter the critical section or to propagate the token to one of its children; and it is not essential that j enters the critical section infinitely often. We have abstracted away the strategy by which j takes this decision. We have also abstracted away the fair strategy by which j decides to which of its children in holder tree it should transmit the token to. The details of these strategies are local to each process and irrelevant for our purposes.

5.2 A Nonmasking Tolerant Version

In the presence of faults, the parent tree used by IM may become partitioned. As a result, the holder relation may also become inconsistent. Moreover, the token circulated by IM may be lost, e.g., when the process that has the token (i.e., whose holder equals itself) fail-stops. Hence, to add nonmasking tolerance to fail-stops, we need to add a corrector that restores the parent tree and the holder tree. We build this corrector by superposing two correctors: NT which corrects the parent tree and NH which corrects the holder tree. In particular, we ensure that in the presence of fail-stops eventually the parent tree is constructed, the holder relation is identical to the parent relation and, hence, the root process has the token.

5.2.1 Designing a Corrector NT for the *parent* Tree

For a corrector that reconstructs the parent tree, we reuse Arora's program [18] for tree maintenance. This program allows faults to yield program states where there are multiple trees and unrooted trees. Continued execution of the program ensures convergence to a fixpoint state where there is exactly one rooted spanning tree.

To deal with multiple trees, the program has actions that merge trees. The merge actions use an integer variable $root.j$, denoting the id of the process that j believes to be its tree root, as follows. 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 $P.j$ to be k. Also, j aligns its holder relation along the parent relation by setting $h.j$ to k. Observe that, by merging thus, no cycles are formed and the root value of each process remains at most the root value of its parent. When no merge actions are enabled, it follows that all rooted processes have the same root value.

To deal with unrooted trees, the program has actions that inform all processes in unrooted trees that they have no root process. These actions use a variable $col.j$, denoting the color of j, as follows. When a process detects that its parent has failed or the color of its parent is red, the process sets its color to red. When a leaf process obtains the color red, it separates from its tree and resets its color to green, thus forming a tree consisting only of itself. When a leaf separates from its tree, it aligns its holder relation along the parent relation by setting its holder to itself. Formally, the actions of the corrector NT for process j are as follows ($Adj.j$ denotes the set of up neighbors of process j):

$$
\begin{array}{lll}
NT1 :: & col.j = green \wedge & \\
 & (P.j \notin Adj.j \cup \{j\} \ \vee \ col.(P.j) = red) \longrightarrow & col.j := red \\[1em]
NT2 :: & col.j = red \wedge & \\
 & (\forall k : k \in Adj.j : P.k \neq j) \qquad \longrightarrow & col.j, P.j, root.j, h.j := green, j, j, j \\[1em]
NT3 :: & k \in Adj.j \ \wedge \ root.j < root.k \ \wedge & \\
 & col.j = green \ \wedge \ col.k = green \qquad \longrightarrow & P.j, root.j, h.j := k, root.k, k
\end{array}
$$

Fault-span and Invariant. In the presence of faults, the actions of NT 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 is also colored red. Thus, the fault-span of corrector NT is the predicate T_{NT}, where

$$
\begin{aligned}
T_{NT} = {} & \text{the graph of the parent relation is a forest } \wedge \\
& (\forall j : up.j : (col.j = red \;\Rightarrow\; (P.j \notin Adj.j \cup \{j\} \;\vee\; col.(P.j) = red)) \;\wedge \\
& \qquad (P.j = j \;\Rightarrow\; root.j = j) \;\wedge\; (P.j \neq j \;\Rightarrow\; root.j > j) \;\wedge \\
& \qquad (P.j \in Adj.j \;\Rightarrow\; (root.j \leq root.(P.j) \;\vee\; col.(P.j) = red)))
\end{aligned}
$$

In the absence of faults, the graph of parent relation forms a rooted spanning tree. In particular, the root values of all processes are identical. Furthermore, if a process is colored red then all its children are colored red, i.e., all processes in any unrooted tree are colored red. Finally, all processes are colored green, i.e., no process is in an unrooted tree. Thus, the invariant of corrector NT is the predicate S_{NT}, where

$$
\begin{aligned}
S_{NT} = T_{NT} \wedge (\forall j : up.j : & (col.j = red \;\Leftarrow\; (P.j \notin Adj.j \cup \{j\} \;\vee\; col.(P.j) = red)) \;\wedge \\
& (col.j = green) \;\wedge\; (\forall k : k \in Adj.j : root.j = root.k))
\end{aligned}
$$

Remark. Henceforth, for brevity, we use the term $ch.j$ to denote the children of j; the term j *is a root* to denote that the parent of j is j, $col.j$ is green, and j is up; and the term $nbrs(X)$ to denote the set of processes adjacent to processes in the set of processes X (including X). Formally,

$$
\begin{aligned}
& ch.j = \{k : P.k = j\} - \{j\}. \\
& j \textit{ is a root} \;\equiv\; (P.j = j \;\wedge col.j = green \;\wedge\; up.j) \\
& nbrs(X) \;=\; (X \cup \{l : (\exists m : m \in X \wedge l \in Adj.m)\})
\end{aligned}
$$

5.2.2 Designing a Corrector NH for the *holder* Tree

After the parent tree is reconstructed, the holder relation may still be inconsistent, in two ways. (1) The holder of j need not be adjacent to j in the parent tree, or (2) the holder of j may be adjacent to j in the tree but the holder relation forms a cycle. Hence, the corrector NH that restores the holder relation consists of two actions: Action $NH1$ corrects the holder of j when (1) holds, by setting $h.j$ to $P.j$. Action $NH2$ corrects the holder of j when (2) holds: if the parent of k is j, holder of j is k and the holder of k is j, j breaks this cycle by setting $h.j$ to $P.j$. The net effect of executing these actions is that eventually the holder relation is identical to the parent relation and, hence, the root process has the token.

$NH1 :: \;\; ((h.j \notin (\{j, P.j\} \cup ch.j)) \;\;\vee\;\; (P.j \neq j \;\wedge\; h.j \neq P.j \;\wedge\; h.(P.j) \neq j))) \quad \longrightarrow h.j := P.j$

$NH2 :: \;\; h.j = k \;\wedge\; h.k = j \;\wedge\; j \neq k \;\wedge\; P.j = k \quad \longrightarrow h.j := P.j$

Fault-Span and Invariant. The corrector NH ensures that the holder of j is adjacent to j in the parent tree and for every edge $(j, P.j)$ in the parent tree, either $h.j$ is the same as $P.j$, or $h.(P.j)$ is the same as j, but not both. Thus, NH corrects the program to a state where $S_{NH} \equiv (\forall j : up.j : S_{NH1}.j \wedge S_{NH2}.j)$ is satisfied, where

$$S_{NH1}.j \equiv (h.j \ \in \ (\{j, P.j\} \cup ch.j) \ \wedge \ (j \neq P.j \quad \Rightarrow \quad (h.j = P.j \ \vee \ h.(P.j) = j))$$

$$S_{NH2}.j \equiv (j \neq P.j \quad \Rightarrow \quad \neg(h.j = P.j \ \wedge \ h.(P.j) = j))$$

5.2.3 Adding the corrector : Verifying interference freedom

As described earlier, the corrector we add to IM is built by superposing two correctors NT and NH. NH updates only the holder relation and NT does not read the holder relation. Therefore, NH does not interfere with NT. Also, after NT reconstructs the tree and satisfies S_{NT}, none of its actions are enabled. Therefore, NT does not interfere with NH.

IM updates variables that are not read by NT. Therefore, IM does not interfere with NT. Also, NH reconstructs the holder relation by satisfying the predicates $S_{NH1}.j$ and $S_{NH2}.j$ for each process j, both of which are respectively preserved by IM. Therefore, IM does not interfere with NH. Finally, after the tree and the holder relation is reconstructed and $(S_{NT} \ \wedge \ S_{NH})$ is satisfied, actions of NT and NH are disabled. Therefore, NT and NH do not interfere with IM.

It follows that the corrector consisting of both NT and NH ensures that a state satisfying $(S_{NT} \wedge S_{NH})$ is reached, even when executed concurrently with IM. Since $(S_{NT} \wedge S_{NH} \quad \Rightarrow \quad S_{IM})$, we may add the corrector to IM to obtain the nonmasking tolerant program NM, whose actions at process j are as follows:

$NM1 :: \quad IM1$
$NM2 :: \quad IM2$
$NM3 :: \quad IM3$

$NM4 :: \quad NT1$
$NM5 :: \quad NT2$
$NM6 :: \quad NT3$

$NM7 :: \quad NH1$
$NM8 :: \quad NH2$

Fault-Span and Invariant. The invariant of program NM is the conjunction of S_{NT}, and S_{NH}. Thus, the invariant of NM is

$S_{NM} \equiv (S_{NT} \wedge S_{NH})$.

The fault-span of program NM is equal the T_{NT}, i.e.,

$T_{NM} \equiv T_{NT}$

Theorem 5.1. The mutual exclusion program NM is nonmasking fault-tolerant for S_{NM}.

5.3 Enhancing the Tolerance to Masking

Actions $NM5, NM7$ and $NM8$ can affect the safety of program execution only when the process executing them sets the holder to itself, thereby generating a new token. The safe predicate that should hold before generation of the a token is therefore the condition "no process has a token". Towards detection of this safe predicate, we exploit the fact that NM is nonmasking tolerant: if the token is lost, NM eventually converges to a state where the graph of the parent relation is a rooted tree and the holder of each processes is its parent and, hence, it suffices to check whether the program is at such a state. To perform this check, we let j initiate a diffusing computation whenever j executes action $NM5, NM7$ or $NM8$. Only when j completes the diffusing computation successfully, does it safely generate a token.

Actions $NM2$ and $NM3$, which respectively let process k transmit a token to process j and let j enter its critical section, can affect the safety of program execution only if they involve a spurious token generated in the presence of fail-stops. The safe predicate that should hold before these actions execute would certify that the token is not spurious. Towards detection of this safe predicate, we exploit the fact that fail-stops are detectable faults, and hence we can let the fail-stop of a process force its neighboring processes to participate in a diffusing computation. Recalling from above that a new token is safely generated only after a diffusing computation completes, we can define the safe predicate for $NM2$ to be defined to be "k is not participating in a diffusing computation" and for action $NM3$ to be "j is not participating in a diffusing computation".

Observe that the safe predicate detection to be performed for the first set of actions ($NM5, NM7$, and $NM8$) is global, in that it involves the state of all processes, whereas the safe predicate detection to be performed for the second set of actions ($NM2$ and $NM3$) is local. We will design a separate detector for each set of actions, such that superposition of these detectors on NM yields a masking fault-tolerant program.

5.3.1 Designing the Global Detector, GD

As discussed above, the global detector, GD, uses a diffusing computation to check if some process has a token. Only a root process can initiate a diffusing computation. Upon initiation, the root propagates the diffusing computation to all of its children. Each child likewise propagates the computation to its children, and so on. It is convenient to think of these propagations as a propagation wave. When a leaf process receives the propagation wave, it completes and responds to its parent. Upon receiving responses from all its children, the parent of the leaf likewise completes and responds to its parent, and so on. It is convenient to think of these completions as a completion wave. In the completion wave, a process responds to its parent with a result denoting whether the subtree rooted at that process has a token. Thus, when the root receives a completion wave, it can decide whether some process has a token by inspecting the result.

The diffusing computation is complicated by the following situations: multiple (root) processes may initiate a diffusing computation concurrently, processes may fail-stop while the diffusing computation is in progress, and a process may receive a token after responding to its parent in a diffusing computation that it does not have the token.

To deal with concurrent initiators, we let only the diffusing computation of the highest id process to complete successfully; those of the others are aborted, by forcing them to complete with the result false. Specifically, if a process propagating a diffusing computation observes another diffusing computation initiated by a higher id process, it starts propagating the latter and aborts the former diffusing computation by setting the result of its former parent (the process from it received the former diffusing computation) to false. This ensures that the former parent completes the diffusing computation of the lower id process with the result false.

To deal with the fail-stop of a process, we abort any diffusing diffusing computations that the neighboring processes may be propagating: Specifically, if j is waiting for a reply from k to complete in a diffusing computation and k fail-stops then j cannot decide if some descendent of k has a token. Hence, upon detecting the fail-stop of k, j aborts its diffusing computation by setting its result to false.

Finally, to deal with the potential race condition where a diffusing computation "misses" a token because the token is sent to some process that has already completed in the diffusing computation with the result true, we ensure that even if this occurs the diffusing computation completes at the initiator only with the result false. Towards this end, we modify the global detector as follows: A process completes in a diffusing computation with the result true only if all its neighbors have propagated that diffusing computation. And, the variable result is maintained to be false if the process ever had a token since the last diffusing computation was propagated. To see why this modification works, consider the first process, say j, that receives a token after it has completed in a diffusing computation with the result true. Let l denote the process that sent the token to j. It follows that l has at least propagated the diffusing computation and its result is false. Moreover, since j is the first process to receive a token after completing the diffusing computation with the result true, l can only complete that diffusing computation with the result false. Since the result of l is propagated towards the initiator of the diffusing computation in the completion wave, the initiator is guaranteed to complete the diffusing computation with the result false.

In sum, the diffusing computation deals with each of these complications via an *abort* mechanism that, by setting the result of the appropriate processes to false, fails the appropriate diffusing computations.

When the initiator of a diffusing computation completes with the result false, it starts yet another diffusing computation. Towards this end, the diffusing computation provides an *initiation* mechanism that lets a root process initiate a new diffusing computation. To distinguish between the different computations initiated by some process, we let each process maintain a sequence number that is incremented in every diffusing computation. Furthermore, when a process propagates a new diffusing computation, it resets its result to true provided that it does not have the token.

From the above discussion, process j needs to maintain a phase, $phase.j$, a sequence number, $sn.j$, and a result, $res.j$. The phase of j is either *prop* or *comp* and denotes whether j is propagating a diffusing computation or it has completed its diffusing computation. The sequence number of j distinguishes between successive diffusing computations initiated by a root process. Finally, the result of j denotes whether j completed its diffusing computation correctly or it aborted its diffusing computation.

Actions for the global detector. The global detector consists of four actions, *viz* $INIT$, $PROP$, $COMP$, and $ABORT$.

$INIT$ lets process j initiate a diffusing computation by incrementing its sequence number. We specify here only the statement of $INIT$; the conditions under which j executes $INIT$ are specified later.

$PROP$ lets j propagate a diffusing computation when j and $P.j$ are in the same tree and $sn.j$ is different from $sn.(P.j)$. If the holder relation of j is aligned along the parent relation and $P.j$ is in the propagate phase, j propagates that diffusing computation and sets its result to true. Otherwise, j completes that diffusing computation with the result false.

$COMP$ lets j complete a diffusing computation if all children have completed the diffusing computation and all neighbors have propagated or completed that diffusing computation. The result computed by j is set to true iff the result returned by all its children is true, all neighbors of j have propagated that diffusing computation, and the result of j is true. If the root completes a diffusing computation with the result true, the safe predicate has been detected and the root process can proceed to safely generate a new token and, consequently, change its result to false.

$ABORT$ lets j complete a diffusing computation prematurely with the result false. When j aborts a diffusing computation, j also sets the result of its parent to false to ensure that the parent of j completes its diffusing computation with the result false. We specify here only the statement of $ABORT$; the conditions under which j executes $ABORT$ are specified later. Formally, the actions of detector GD for process j are as follows:

$$
\begin{array}{lll}
INIT(j) :: & \text{if } (P.j = j) \text{ then } phase.j,\ sn.j,\ res.j := prop, newseq(),\ true & \\
PROP(j) :: & root.j = root.(P.j)\ \wedge\ sn.j \neq sn.(P.j) & \longrightarrow sn.j := sn.(P.j); \\
& & \quad \text{if } (phase.(P.j) = prop\ \wedge\ (h.j = P.j)) \\
& & \qquad phase.j,\ res.j := prop,\ true \\
& & \quad \text{else } res.j := false \\
COMP(j) :: & phase.j = prop\ \wedge & \longrightarrow res.j := (\forall k : k \in Adj.j \cup \{j\} : res.k); \\
& (\forall k : k \in Adj.j : root.j = root.k\ \wedge & \quad phase.j := comp; \\
& \qquad sn.j = sn.k)\ \wedge & \quad \text{if } (P.j = j\ \wedge\ \neg\, res.j) \text{ then } \quad \text{INIT(j)}; \\
& (\forall k : k \in ch.j : phase.j \neq prop) & \quad \text{else if } (P.j = j\ \wedge\ res.j) \text{ then } res.j := false \\
ABORT(j) :: & phase.j,\ res.j := comp, false;\ \ \text{if } (P.j \in Adj.j) \text{ then } res.(P.j) := false &
\end{array}
$$

Remark. In the $ABORT$ action, j synchronously updates its parent's states in addition to its own. This action can be refined, since the parent of j completes its diffusing computation only after j completes its diffusing computation, so that j only updates its own state and $P.j$ reads the state of j later.

Fault Actions. When a process fail-stops, all of its neighbors abort any diffusing computation that they are propagating. Moreover, if the initiator aborts its diffusing computation it initiates a new one. Hence, the fault action is

$$
failstop ::\quad up.j \quad \longrightarrow \quad up.j := false; (\| \ l : l \in Adj.j \cup \{j\} : ABORT(l); INIT(l))
$$

Invariant. We relegate the invariant S_{GD} of the global detector to Appendix A1.

5.3.2 Designing the Local Detector LD

The safe predicate for action $NM2$ is "k is not participating in a diffusing computation"; that is, $phase.k = comp$. The safe predicate for action $NM3$ is "j is not participating in a diffusing computation"; that is, $phase.k = comp$. Therefore, these actions are modified as follows:

$$
\begin{array}{lll}
LD1 :: & h.k = k\ \wedge\ j \in Request.k\ \wedge & \longrightarrow h.k, h.j := j, j; \\
& (P.j = k\ \vee\ P.k = j)\ \wedge\ phase.k = comp & \qquad Request.k := Request.k - \{j\} \\
LD2 :: & h.j = j\ \wedge\ phase.j = comp & \longrightarrow \text{access critical section}
\end{array}
$$

5.3.3 Adding the detectors : Verifying interference Freedom

Actions $NM5, NM7$, and $NM8$ are restricted to execute $INIT$, to initiate a diffusing computation whose successful completion, i.e. execution of $COMP$ with result true will generate a new token. And, as described above, actions $NM2$ and $NM3$ are restricted with the local detectors, to obtain $LD1$ and $LD2$, respectively. We still need to verify that the composition is free from interference.

Note that the global detector, GD, does not update the variables that are updated by NM. It follows that GD is a superposition on NM and, hence, GD does not interfere with NM. To ensure that NM does not interfere with GD, we restrict all actions of NM, other than $NM5, NM7$, and $NM8$, to execute $ABORT$. (The alert reader will note that this last restriction is overkill: some actions of NM need not be thus restricted, but we leave that optimization as an exercise for the reader.) As long as the correctors of NM are executing, GD is safely aborted. Once the correctors of NM terminate, their terminal actions will initiate a new diffusing computation and GD will make progress. Hence, NM does not interfere with GD. Also, execution of GD eventually reaches a state where the phase of all processes is *comp*. Thus, LD does not interfere with NM, and since LD detects the safe predicate atomically, it is not interfered by NM and GD.

Formally, the actions of the resulting masking tolerant program MM are as follows:

$MM1 ::$ $IM1 \;\|\; ABORT(j)$
$MM2 ::$ $LD1 \;\|\; ABORT(j)$
$MM3 ::$ $LD2 \;\|\; ABORT(j)$

$MM4 ::$ $NM4 \;\|\; ABORT(j)$
$MM5 ::$ $NM5 \;\|\; INIT(j)$
$MM6 ::$ $NM6 \;\|\; ABORT(j)$
$MM7 ::$ $NM7 \;\|\; INIT(j)$
$MM8 ::$ $NM8 \;\|\; INIT(j)$

$MM9 ::$ $PROP(j)$
$MM10 ::$ $COMP(j)$

Fault Actions. The fault action is identical to the fault action described in Section 5.3.1.

Invariant. The invariant of program MM is the conjunction of T_{NM} and S_{GD}. Thus, the invariant of MM is

$$S_{MM} \;\equiv\; (T_{NM} \;\wedge\; S_{GD})$$

Theorem 6.3. The mutual exclusion program MM is masking tolerant for S_{MM}.

Remark. A leader election program can be easily extracted from our mutual exclusion case study. To this end, we drop the variables h and *Request* from program MM. Thus, the resulting program consists of the corrector NT (actions $MM4{-}6$) and the detector GD (actions $MM9$ and $MM10$). In this program, a process is a leader iff it is a root and its phase is *comp*. This program is derived by adding detector GD to the nonmasking tolerant program NT. The detailed design of such a leader election program is presented in [23,24].

6 Discussion and Concluding Remarks

In this paper, we presented a compositional method for designing masking fault-tolerant programs. First, by corrector composition, a nonmasking fault-tolerant program was designed to ensure that, once faults stopped executing, the program eventually reached a state from where the problem specification was satisfied. Then, by detector composition, the program was augmented to ensure that, even in the presence of the faults, the program always satisfied its safety specification.

We demonstrated the method by designing classical examples of masking fault-tolerant programs. Notably, the examples covered a variety of fault-classes including Byzantine faults, message faults, input faults and processor fail-stops and repairs. Also, they illustrated the generality of the method, in terms of its ability to provide alternative designs for programs usually associated with other well-known design methods for masking fault-tolerance: Specifically, the TMR and Byzantine examples are usually associated with the method of *replication* or, more generally, the *state-machine-approach* for designing client-server programs [19]. The alternating-bit protocol example is usually associated with the method of *exception handling* or that of *rollback-recovery* —with the "timeout" action, $MD5$, being the exception-handler or recovery-procedure.

We found that judicious use of this method offers the potential for the design of improved masking tolerant solutions, measured in terms of the scope of fault-classes that are masked and/or the performance of the resulting programs. This is because, in contrast to some of the wellknown design methods, the method is not committed to the overhead of replication; instead, it encourages the design of minimal components for achieving the required tolerance. And, in contrast to the sometimes ad hoc treatment of exception-handling and recovery procedures, it focuses attention on the systematic resolution of the interference between underlying program and the added tolerance components.

One example of an improved masking tolerant solution designed using the method is our token-based mutual exclusion program. In terms of performance, in the absence of faults, our program performs exactly as its fault-intolerant version (due to Raymond [16] and Snepscheut [17]) and thus incurs no extra overhead in this case. By way of contrast, the acyclic-graph-based programs of Dhamdhere and Kulkarni [20] and Chang, Singhal, and Liu [21] incur time overhead for providing fault-tolerance, even in the absence of faults. Also, in the tree based program of Agrawal and Abbadi [22], the amount of work performed for each critical section may increase when processes fail (especially when the failed processes are close to the tree root); in our program, failure of a process causes an overhead only during the convergence phase, but not after the program converges. Moreover, in terms of tolerance, our program is more tolerant than that of [20] (which is intolerant to the fail-stop of the process that holds the token), and [22] (which in the worst case is intolerant to more than $log\ n$ process fail-stops).

We note in passing that our mutual exclusion program can be systematically extended to tolerate process repairs as well as channel failures and repairs. Also, it can be systematically transformed so that processes cannot access the state of their neighbors atomically but only via asynchronous message passing. For other examples of improved solutions designed using the method, the interested reader is referred to our designs for leader election [23, 24], termination detection [23, 24], and distributed reset [25].

We also note that although superposition was used for detector composition in each of our example designs, superposition is only one of the possible strategies for detector composition. The advantage of superposing the detectors on the underlying nonmasking tolerant program is the immediate guarantee that the detectors did not interfere with the closure and convergence properties of the underlying program.

One useful extension of the method would be to design programs that are nonmasking tolerant to one fault-class and masking tolerant to another or, more generally, that possess multiple tolerance properties (see [12, 26, 25]). The design of such multitolerant programs is motivated by the insight that the fault-span of a program need not be unique [5]. Hence, multiple fault-spans may be associated with a program, for instance, if the program is subject to multiple fault-classes. It follows that the program can be nonmasking tolerant to one of these fault-classes and masking tolerant to another. More generally, we find that multitolerance has several practical applications [12].

Another useful extension would be to augment the method to allow "tolerance refinement", i.e., to allow refinement of a tolerant program from an abstract level to a concrete level while preserving its tolerance property. Tolerance refinement is orthogonal to the "tolerance addition" considered in the paper, which adds the desired masking tolerance directly at any desired (but fixed) level of implementation. With this extension we could, for instance, refine our mutual exclusion program so that neighboring processes communicate only via asynchronous message passing within the scope of the method itself.

Finally, alternative design methods based on detector and corrector compositions would be worth studying. An alternative stepwise method would be to first perform detector composition and then perform corrector composition, which we view as designing masking tolerance via fail-safe tolerance [12]. Another alternative (but not stepwise) method would be to compose detectors and correctors simultaneously. It would be especially interesting to compare these methods with respect to design-complexity versus performance-complexity tradeoffs.

Acknowledgments. We are grateful to Ted Herman for helpful comments on a preliminary version of this paper and thank the anonymous referees for their detailed, constructive suggestions.

References

[1] F. Bastani, I.-L Yen, and I. Chen. A class of inherently fault-tolerant diffusing programs. *IEEE Transactions on Software Engineering*, 14:1432–1442, 1988.

[2] P. Jalote. *Fault-tolerance in Distributed Systems.* Prentice Hall, 1994.

[3] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engg.*, pages 220–232, 1975.

[4] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, pages 2–11, 1985.

[5] 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.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[7] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 7 October 1985.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

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

[10] B. Alpern and F. Schneider. Proving boolean combinations of deterministic properties. *Proceedings of the Second Symposium on Logic in Computer Scien ce*, pages 131–137, 1987.

[11] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[12] A. Arora and S. S. Kulkarni. Multitolerance and its design. Technical Report OSU-CISRC TR37, Ohio State University, 1996. Submitted to *IEEE Transactions on Software Engineering, Special Issue on Formal Methods.*

[13] G. Tel. *Structure of Distributed Algorithms.* PhD thesis, University of Utrecht; also published by Cambridge University Press, 1989.

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[15] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics.* Springer-Verlag, 1990.

[16] K. Raymond. A tree based algorithm for mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

[17] J. L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2):113–115, 1987.

[18] A. Arora. Efficient reconfiguration of trees: A case study in the methodical design of nonmasking fault-tolerance. *Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault-Tolerance*, 1994.

[19] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[20] D. M. Dhamdhere and S. S. Kulkarni. A token based k resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50:151–157, 1994.

[21] Y. I. Chang, M. Singhal, and M. T. Liu. A fault tolerant algorithm for distributed mutual exclusion. *IEEE Computer Society*, pages 146–154, 1990.

[22] D. Agrawal and A. Abbadi. An efficient fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, 1991.

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

[24] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *ftp://ftp.cis.ohio-state.edu/pub/anish/papers/mask.ps.Z* , 27 pages, February 1995.

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

[26] S. Kulkarni and A. Arora. Stepwise design of tolerances in barrier computations. Technical Report *OSU-CISRC TR17*, Ohio State University, 1996. Submitted to *Information Processing Letters.*

Appendix

A1 : Correctness of Program MM

Invariant for the global detector, GD : To characterize the invariant S_{GD}, we first define the auxiliary variables $pc.j.k$, $des.j.k$ and $lres.j.k$ as follows:

$$\begin{array}{lcl} pc.j.k & = & \text{the set of processes to whom } j \text{ propagated the current diffusing computation of } k. \\ des.j.k & = & \{j\} \cup (\bigcup l : l \in pc.j.k : des.l.k) \\ lres.j.k & = & \text{the result of } j \text{ when it completed it current diffusing computation of } k. \end{array}$$

By definition, when k initiates a new diffusing computation, $pc.j.k$ is set to the empty set. If l is a child of j and l propagates the current diffusing computation of k for the first time, l is added to $pc.j.k$. Intuitively, $des.j.k$ is the set of processes that received the latest diffusing computation initiated by k *via* j. The value of $lres.j.k$ is *undefined, true,* or *false*. When k initiates a new diffusing computation, $lres.j.k$ is set to undefined. When j completes the current diffusing computation of k for the first time, if j completes it with the result true, $lres.j.k$ is set to true. If j sets its result to false while propagating the current diffusing computation of k, $lres.j.k$ is set to false. Observe that by definition, $lres.j.k$ cannot change from true to false, and vice versa.

Using these auxiliary variables, we proceed to define the invariant S_{GD} for the detector GD. Observe that j propagates a diffusing computation of k only if the id of j is less than that of k. Moreover, if j moves to a different tree or j fails, j sets the result of its old parent to false. Also, when j propagates its diffusing computation it sets its result to true only if its holder relation is aligned along the parent relation. Thus, the predicate S_{GD1} is in S_{GD}, where

$$\begin{array}{ll} S_{GD1} \equiv & (j \in des.k.k \;\Rightarrow\; ((j \leq k) \;\wedge\; (lres.j.k = false \;\Rightarrow\; (\exists l : j \in des.l.k : \neg res.l)))) \;\wedge \\ & (phase.j = prop \;\Rightarrow\; h.j = P.j) \end{array}$$

Suppose that j has propagated the current diffusing computation of k. If j changes its root value or sequence number, or if j fails, then j ensures that its parent will abort that diffusing computation. Thus, the predicate S_{GD2} is in S_{GD}, where

$$\begin{array}{ll} S_{GD2} \equiv & j \in des.k.k \;\wedge\; lres.j.k \neq true \;\wedge \\ & (root.j \neq k \;\vee\; j \notin (pc.(P.j).k \cup \{j\}) \;\vee \\ & P.j \notin (Adj.j \cup \{j\}) \;\vee\; sn.j \neq sn.k) \qquad \Rightarrow \quad (\exists l : j \in des.l.k \;\wedge\; j \neq l : lres.l.k = false) \end{array}$$

When j propagates a diffusing computation of k and sets its result to true, j does not have a token. If j receives a token before completing that diffusing computation, j aborts that diffusing computation. When j completes its diffusing computation, it checks that all its neighbors have propagated that diffusing computation. Thus, if j receives a token from its neighbor, say l, l aborts that diffusing computation. It follows that if j receives a token after propagating a diffusing computation, there exists some process l that has completed that diffusing computation with the result false. Thus, the predicate S_{GD3} is in S_{GD}, where

$$S_{GD3} \equiv (j \in des.k.k \ \wedge \ phase.k = prop) \quad \Rightarrow \quad ((h.j = P.j) \ \wedge \ (P.j \neq j \ \vee \ phase.j = prop)) \ \vee \ (\exists l : l \in des.k.k : lres.l.k = false)$$

When j propagates the current diffusing computation of k, $pc.j.k$ is empty. A process is added to $pc.j.k$ only if it is a child of j. And, if a child of j moves to a different tree or fails, j aborts its diffusing computation. Thus, the predicate S_{GD4} is in S_{GD}, where

$$S_{GD4} \equiv (j \in des.k.k \ \wedge \ phase.j = prop) \ \Rightarrow \ ((pc.j.k \subseteq ch.j) \ \vee \ (\exists l : j \in des.l.k : lres.l.k = false))$$

When j propagates the current diffusing computation of k, $root.j$ is the same as k, $sn.j$ is the same as $sn.k$, and $res.j$ is true. Thus, the predicate S_{GD5} is in S_{GD}, where

$$S_{GD5} \equiv (k \ is \ a \ root \ \wedge \ root.j = k \ \wedge \ sn.j = sn.k \ \wedge \ res.j) \quad \Rightarrow \quad j \in des.k.k$$

When j completes its diffusing computation with the result true, it detects that all its descendents ($des.j.k$) have completed that diffusing computation with the result true and the neighbors of its descendents ($nbrs(des.j.k)$) have propagated that diffusing computation. Thus, the predicate S_{GD6} is in S_{GD}, where

$$\begin{aligned} S_{GD6} \equiv\ & (root.j = k \ \wedge \ sn.j = sn.k \ \wedge \\ & phase.j = comp \ \wedge \ lres.j.k = true) \quad \Rightarrow \quad (((\forall l : l \in des.j.k : lres.l.k = true) \ \wedge \\ & \qquad (nbrs(des.j.k) \subseteq des.k.k))) \\ & \qquad \vee \\ & \qquad (\exists l : j \in des.l.k : lres.l.k = false) \end{aligned}$$

Finally, in any state, at most one process has a token, i.e., the predicate S_{GD7} is in S_{GD}, where

$$S_{GD7} \equiv (h.j = j \ \wedge \ phase.j = comp \ \wedge \ h.k = k \ \wedge \ phase.k = comp) \quad \Rightarrow \quad j = k$$

From the above description, the predicates S_{GD1-7} are in S_{GD}. We now define S_{GD} as

$$S_{GD} \equiv (\forall j, k :: S_{GD1} \ \wedge \ S_{GD2} \ \wedge \ S_{GD3} \ \wedge \ S_{GD4} \ \wedge \ S_{GD5} \ \wedge \ S_{GD6} \ \wedge \ S_{GD7})$$

Proof of Correctness.

We need to prove that no process has a token when a root k completes the diffusing computation with the result true. Towards this end, we use the predicates S_{GD3} and S_{GD6}. The predicate S_{GD6} is used to show that when k completes its diffusing computation, all processes participated in that diffusing computation. The predicate S_{GD3} is used to show in a state where the k can complete its diffusing computation with the result true, no can be accessing its critical section.

If k has completed its diffusing computation with the result true, from S_{GD6}, we have

$(((\forall l : l \in des.k.k : lres.l.k = true) \wedge (nbrs(des.k.k) \subseteq des.k.k))$
$\vee\ (\exists l : k \in des.l.k : lres.l.k = false))$
$\Rightarrow$ { by definition of $des.k.k$ }
$(((\forall l : l \in des.k.k : lres.l.k = true) \wedge (nbrs(des.k.k) \subseteq des.k.k)) \ \vee\ lres.k.k = false)$
$\Rightarrow$ { by definition of $des.k.k$ and S_{GD1} }
$(((\forall l : l \in des.k.k : lres.l.k = true) \wedge (nbrs(des.k.k) \subseteq des.k.k)) \ \vee\ \neg res.k.k)$
$\Rightarrow$ { when k completes its diffusing computation with the result true, $res.k$ is true. }
$(((\forall l : l \in des.k.k : lres.l.k = true) \wedge (nbrs(des.k.k) \subseteq des.k.k))$
$\Rightarrow$ { since the graph of processes remains connected }
$((\forall l : l \in des.k.k : lres.l.k = true) \ \wedge\ (\forall l :: l \in des.k.k))$
$\Rightarrow$ { by predicate calculus }
$((\forall l :: lres.l.k = true) \ \wedge\ (\forall l :: l \in des.k.k))$
$\Rightarrow$ { by predicate calculus }
$((\neg(\exists l :: lres.l.k = false)) \ \wedge\ (\forall l :: l \in des.k.k))$
$\Rightarrow$ { by S_{GD3} }
$(\forall j :: (h.j = P.j) \ \wedge\ (P.j \neq j \ \vee\ phase.j = prop))$
$\Rightarrow$ { from guard of action $NM3$ }
$(\forall j :: j \textit{ cannot be accessing its critical section})$

A2 : Correctness of Nonmasking Tolerance of Mutual Exclusion Program

Theorem 5.1. The mutual exclusion program NM is nonmasking tolerant for S_{NM}.

Proof. We need to prove that T_{NM} converges to S_{NM}. Our proof of convergence is in three stages (recall from Section 5.2 that $S_{NM} \equiv (S_{NT} \ \wedge\ (\forall j :: S_{NH1.j} \ \wedge\ S_{NH2.j})))$:

1.	T_{NM}	converges to	S_{NT}
2.	S_{NT}	converges to	$S_{NT} \ \wedge\ (\forall j :: S_{NH1.j})$
3.	$S_{NT} \ \wedge\ (\forall j :: S_{NH1.j})$	converges to	S_{NM}

Proof of Stage 1: Follows immediately from the convergence of Arora's nonmasking tree program [18], i.e., $T_{NM}(= T_{NT})$ converges to S_{NT}.

Proof of Stage 2: Consider the set of processes $X = \{j : \neg S_{NH1.j}\}$. Action $NM7$ is enabled at these processes. When a process executes $NM7$ the cardinality of the set X decreases. No action increases the cardinality of X. When the cardinality of the set X is zero, $(\forall j :: S_{NH1.j})$ holds. Thus, S_{NT} converges to $S_{NT} \wedge (\forall j :: S_{NH1.j})$.

Proof of Stage 3: Let $d.j$ denote the distance of j from the root of the tree. Consider the variant function $(\sum j : \neg S_{NH2.j} : d.j)$. At any state if $S_{NH2.j}$ is not satisfied, action $NM8$ is enabled at process j. When j executes action $NM8$, the value of the variant function decreases. Since, the value of the variant function is nonnegative, and non-increasing, eventually the program reaches a state where $(\forall j :: S_{NH2.j})$ holds, i.e., $S_{NT} \wedge (\forall j :: S_{NH1.j})$ converges to S_{NM}.