Constraint Satisfaction as a Basis for Designing Nonmasking Fault-Tolerance

Anish Arora [1] , *Mohamed Gouda* [2], *George Varghese* [3]

1. Department of Computer Science, The Ohio State University at Columbus
2. Department of Computer Sciences, The University of Texas at Austin
3. Laboratory for Computer Science, Massachussetts Institute of Technology

Abstract

We present a method for the design of nonmasking fault-tolerant programs. In our method, a set of constraints is associated with each program. As long as faults do not occur, the constraints are continually satisfied under the execution of program actions. Whenever some of the constraints are violated, due to certain faults, all constraints are eventually reestablished by subsequent execution of the program actions. To design programs thus, two types of program actions are distinguished: "closure" actions and "convergence" actions. Closure actions are the actions that perform the intended computation of the program when all of the constraints are satisfied. Convergence actions are the actions that reestablish the constraints when they have been violated. Sufficient conditions for the validation of closure and convergence actions are formalized in terms of a "constraint graph". These conditions are illustrated by designing nonmasking fault-tolerant programs for diffusing computations, atomic actions, and token rings.

Keywords: design, methodology, distributed constraints, nonmasking fault-tolerance, closure, convergence, stabilization.

1 Introduction

One way to achieve program fault-tolerance is to ensure that when faults occur the program continues to satisfy its input-output relation. Programs designed thus are said to "mask" the effects of faults. Over the last few decades, masking fault-tolerant programs have been studied in great depth by the fault-tolerance community and, as a result, a variety of design methods have resulted. These include — to name but a few — methods based on redundancy, error correction codes, replication and voting, broadcast and agreement, and multiprocess synchronization.

In certain situations, however, the potential for program fault-tolerance either cannot or should not be realized via masking fault-tolerance. For instance, there are situations where achieving masking fault-tolerance is

- *Impossible* : e.g., there is no asynchronous distributed program whose processes reach consensus on a binary value and mask the effect of a process crash [1],
- *Impractical* : e.g., the amount of redundancy and synchronization required to implement fail-stop processors that mask the effect of byzantine faults can be prohibitively expensive [2], or
- *Unnecessary* : e.g., a call-back telephone service that eventually establishes a connection is useful even if it does not mask its initial failure to establish a connection.

In such situations, an alternative way to achieve program fault-tolerance is to ensure that when faults occur the input-output relation of the program is violated only temporarily. In other words, the program is guaranteed to eventually resume satisfying its input-output relation. Such "nonmasking" fault-tolerant programs work by restoring the program to some previously checkpointed state, by replaying some part of the computation or by repairing faulty program parts, whenever a violation of the input-output relation is detected. In comparison to masking fault-tolerant programs, the design of nonmasking ones has received lesser study [3].

In this paper, we present a method for the design of nonmasking fault-tolerant programs. Our method is motivated by a formal definition of what it means for a program to be fault-tolerant [4, 5]. In the method, a set of constraints is associated with each program. As long as faults do not occur, the constraints are continually satisfied under the execution of program actions. Whenever some of the constraints are violated, due to certain faults, all constraints are eventually reestablished by subsequent execution of the program actions. To design programs thus, two types of program actions are distinguished: closure actions and convergence actions. Closure actions are the actions that perform the intended computation of the program when all of the constraints are satisfied. Convergence actions are the actions that reestablish the constraints when they have been violated. Sufficient conditions for the validation of closure and convergence actions are formalized in terms of a constraint graph. These conditions are illustrated by designing nonmasking fault-tolerant programs for diffusing computations, atomic actions, and token rings.

The rest of the paper proceeds as follows. In Section 2, we define formally our programming notation. In Section 3, we describe our method for designing nonmasking fault-tolerant programs, based on closure and convergence actions. In Section 4, we formalize the notion of the constraint graph of a program. We employ this notion, in Section 5, to present a sufficient condition for validating the closure and convergence actions of programs whose constraint graph is an out-tree. We illustrate the condition by designing a nonmasking fault-tolerant programs for diffusing computations. We then present more general sufficient conditions, in Section 6, for programs with self-looping constraint graphs and, in Section 7, for programs with cyclic constraint graphs. We illustrate these more general conditions by designing nonmasking fault-tolerant programs for atomic actions and token-rings. Finally, we make concluding remarks in Section 8.

2 Programs and Computations

A *program* is a finite set of variables and a finite set of actions. Each variable has a predefined nonempty domain, and each action has the form :

$$\langle \text{guard} \rangle \quad \rightarrow \quad \langle \text{statement} \rangle$$

A guard is a boolean expression over program variables. A statement updates zero or more of program variables and always terminates upon execution.

Let p be a program. A *state* of p is defined by a value for each variable of p (chosen from the 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 the guard of the action holds at that state.

An action of p *preserves* a state predicate R iff starting from any state where R holds and the action is enabled, executing the action yields a state where R holds. A state predicate R of p is *closed* iff each action of p preserves R.

A *computation* of a set of actions of p is a fair, maximal sequence of steps; in every step, some action in the set that is enabled in the current state is executed. Fairness of the sequence means that each action in the set that is continuously enabled along the sequence is eventually executed. Maximality of the sequence means that if the sequence is finite then no action in the set is enabled in the final state.

3 Designing Nonmasking Fault-Tolerant Programs

To motivate our method, let us first observe that the input-output relation of (both sequential and concurrent) programs can be characterized by a state predicate that is true throughout the program execution [6]. Such an *invariant* predicate serves two purposes. First, it identifies the set of "fault-free" states of the program; these are the states starting from which every computation of the program is guaranteed to meet the specification of the program, i.e., the safety and progress properties required of the program. Second, an invariant predicate constrains the design of program actions by requiring that the set of fault-free states be kept closed under the execution of program actions. Examples showing how to design program invariants appear in [7].

We next observe that the input-output relation of programs in the presence of faults can also be characterized by a state predicate that is true throughout program execution [4, 5]. Such a state predicate identifies the *fault-span* of the program; i.e., the set of states that the program can reach in the presence of faults. (Note that the fault-span includes the fault-free states of the program.) Examples showing how to design fault-span predicates appear in [5]. These examples employ the view that all classes of faults can be represented as actions that change the program state [5, 8]. As a result of this view, a program fault-span identifies a set of states that is kept closed under the execution of program actions as well as fault actions.

We are now ready to give a formal definition of fault-tolerance [4, 5]. Let p be a program, S be the invariant of p, and T be the fault-span of p. (Recall that $S \Rightarrow T$.) We say that *p is T-tolerant for S* iff the following two requirements hold:

- Closure: Both S and T are closed in p.
- Convergence: Every computation of p that starts at any state where T holds, reaches a state where S holds.

The definition above yields a formal classification of masking and nonmasking fault-tolerance [4, 5]. Let p be T-tolerant for S. If $S = T$, we say that p is masking fault-tolerant. Else ($S \neq T$), we say that p is nonmasking fault-tolerant.

Furthermore, the definition above suggests that nonmasking fault-tolerant programs can be designed ideally by separately designing two classes of program actions: closure actions and convergence actions. Closure actions are actions that perform the intended computation of the program when the program state satisfies the program invariant S. Convergence actions are actions that restore the program from a state where the program fault-span T holds but S does not to a state where S holds.

The design problem. Based on the distinction between closure actions and convergence, we formulate next the problem of designing nonmasking fault-tolerant programs:

> Given a "candidate" triple (p, S, T)
> where p consists solely of closure actions that preserve S and T,
> design a set of convergence actions $\{ca.1, ca.2, \ldots ca.n\}$
> such that the augmented program $p \cup \{ca.1, ca.2, \ldots ca.n\}$ is T-tolerant for S.

This design problem is readily solved in the special case where we can design actions that check whether $\neg S$ holds and establish S: in this case, we simply augment the candidate program with the action:

$$\neg S \quad \rightarrow \quad \text{`` establish } S \text{ ''}$$

Since the action is enabled only in states where $\neg S$ holds, it trivially preserves S; since it yields upon execution a state where S holds, it also preserves T; hence, the augmented program satisfies the closure requirement. Moreover, since each continuously enabled action is eventually executed due to the fairness of computation, the action guarantees that every computation of the augmented program that starts from any state where T holds reaches a state where S holds; hence, the augmented program satisfies the convergence requirement. It follows that the augmented program is T-tolerant for S.

In general, however, program actions can access and update only a limited part of the program state. (This is especially true for concurrent programs.) In other words, typically, program actions cannot independently check whether S holds or establish states where S holds. It follows that closure actions may execute in states where $T \wedge \neg S$ holds and convergence actions may execute in states where S holds. As a result, the design problem is complicated by possible interference between the executions of closure actions and convergence actions. Furthermore, if the convergence actions may not establish in one step a state where S holds, the design problem is complicated by possible interference between the executions of the convergence actions themselves.

Our method. We account for the general limitation that program actions access and update only a limited part of the program state, as follows.

First, we partition the invariant S into a set of predicates, which we call the *constraints* in S, that can each be independently checked and established by some program action. The constraints in S are chosen such that their conjunction together with T equivales S. It follows that if every constraint is satisfied in a state where T holds, then S holds in that state.

Next, for each constraint c in S, we design one convergence action of the form:

$$\neg c \quad \rightarrow \quad \text{`` establish } c \text{ while preserving } T \text{ ''}$$

In other words, for each constraint c in S we design a convergence action that independently checks c and, if need be, establishes c while preserving T. Since convergence actions are enabled only when $\neg S$ holds, they trivially preserve S; by design, they also preserve T; it follows that the closure requirement of the augmented program is valid.

To complete the design, it only remains to validate the convergence requirement. Recalling the first of the two interference complications for convergence validation discussed above, we see that convergence actions are enabled only in states where $\neg S$ holds; hence, they do not interfere with the execution of closure action in states where S holds. Closure actions may, however, be enabled in states where $\neg S$ holds; hence, their execution may interfere with that of convergence actions, by violating the constraints already established by some convergence actions. Also, the second interference complication is possible, as convergence actions for some constraints may violate some other constraints of the program invariant upon execution.

Convergence validation is a nontrivial task. Therefore, in the rest of this paper, we focus on the problem of convergence validation. More specifically, we formulate a set of sufficient conditions under which convergence validation is made feasible. Towards this end, we define the notion of a constraint graph in the next section.

4 Constraint Graphs

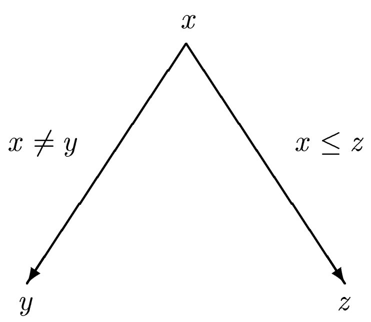
Let q be a set of convergence actions. A *constraint graph* of q is a directed graph that has one edge for each action in q, such that

1. Each node of the graph is labeled with a set of variables that appear in actions in q. Labels of nodes are mutually exclusive; thus, a variable appears in the label of only one node.
2. Each edge of the graph is labeled with an action of q. If the action ac od q labels the edge from node v to node w, then all variables read in ac are in the union of the labels of v and w and all variables written in ac are in the label of w.

Since there is a bijection between constraints and convergence actions, for convenience, we sometimes ambiguously refer to the corresponding constraint as being the label of the edge.

Example : Consider a program that has three integer variable x, y, and z. Consider further that S is the conjunction of the constraints $x \neq y$ and $x \leq z$. Now, if a convergence action satisfies the first constraint by changing x if $x = y$ holds, it can violate the second constraint.

Alternatively, consider for the first constraint a convergence action that changes y if x equals y, and for the second constraint a convergence action that changes z to be at least x if x exceeds z. For these two convergence actions, the constraint graph is:



□

5 Designing Programs with Out-tree Constraint Graphs

In this section, we first identify in Theorem 1 a sufficient condition for the convergence validation of nonmasking fault-tolerant programs. The condition applies to programs whose convergence actions have an out-tree constraint graph. (An out-tree is a weakly connected directed graph one of whose nodes has indegree zero and the remaining of whose nodes have indegree one. See, for example, the constraint graph depicted above.) We then illustrate the condition by designing a nonmasking fault-tolerant program for diffusing computations.

Theorem 1 :

Let (p, S, T) be a candidate triple and q be a set of convergence actions.

If • every closure action of p preserves each constraint in S, and

• the constraint graph of q is an out-tree,

then $p \cup q$ is T-tolerant for S.

Proof : We define the rank of node j in the constraint graph of q to be the nonnegative integer 0, if j is a source node, and $1 + max$ $\{$ rank of node k $|$ there is an edge from k to j and $k \neq j$ $\}$, if j is not a source node. (A source node is one that has no edge adjacent to it from any other node.) Further, we define the rank of a convergence action to be the rank of the target node of the edge that is labeled by the convergence action.

Let C be an arbitrary computation of $p \cup q$ starting from any state where T holds. We show by induction on rank r that if the constraint graph of q is an out-tree, then C has a suffix where the convergence actions of rank $\leq r$ do not execute. It follows that C has a suffix where convergence actions do not execute; i.e., a suffix where each state satisfies S.

Base case $(r = 0)$: Each out-tree edge is of rank > 0, hence C trivially has a suffix where convergence actions of rank ≤ 0 do not execute.

Induction case $(r > 0)$: By the definition of constraint graph, convergence actions update the variables only of the target node of their edge. Hence, the convergence actions of rank $> r$ preserve the constraint of any edge of rank $\leq r$. Also, by assumption, each closure action in p preserves each constraint in S. Thirdly, in any computation of $p \cup q$ where the convergence actions of rank $< r$ do not execute, any convergence action of rank r executes at most once. From these three facts and the induction hypothesis, it follows that C has a suffix where the convergence actions of rank $\leq r$ do not execute.

Recall that in our design method, every closure action (in p) and convergence action (in q) preserves S and T. It follows that $p \cup q$ is T-tolerant for S. □

Note that if the second antecedent in Theorem 1 is generalized to "the constraint graph of q is an out-forest", the theorem is still true.

We illustrate Theorem 1 by designing a "stabilizing" program that maintains a diffusing computation. A stabilizing program [9, 10, 11] is one that exhibits an extreme form of nonmasking fault-tolerance: regardless of the state the program is started in, execution of the program converges to a state from where S holds. It follows that stabilizing programs can tolerate faults whose effect is to somehow corrupt the program state. Note that for stabilizing programs, the program fault-span T is the state predicate $true$; hence, each action of a stabilizing program trivially preserves T.

5.1 Stabilizing Diffusing Computations

Diffusing computations are used commonly in distributed systems to perform tasks that involve accessing or modifying the collective system state. Applications of diffusing computations include, for example, global state snapshot, termination detection, deadlock detection, and distributed reset. Typically, a distinguished process in the system periodically initiates a diffusing computation, which then propagates across the system to perform some subtask at each process. Having completely spanned the system, the computation then collapses back to the distinguished process.

Below, we specify and design an abstract version of a diffusing computation, that is independent of any specific application. (The resulting program is a simplified version of a program in [12].)

Specification :

Consider a finite, rooted tree. Desired is a program in which, starting from a state where all tree nodes are colored green, the root node initiates a diffusing computation. The diffusing computation then propagates from the root to the leaves, coloring the tree nodes red. Upon reaching the leaves, the diffusing computation is reflected back towards the root, coloring the tree nodes green. And the cycle repeats. The program should tolerate faults that arbitrarily corrupt the state of any number of nodes.

Design :

First, we characterize S. Let $c.j$ be the color of node j, and let $sn.j$ be a boolean session number that is used to distinguish "j has not started participating in the current diffusing computation" from "j has completed participating in the current diffusing computation".

Also, let $P.j$ be the parent node of j in the tree (hence if j is the root then $P.j$ is j, else $P.j$ is the unique node from which there is an edge to j in the tree).

We postulate that when all j are colored green, all j have the same session number. Hence, to distinguish "j has not started participating in the current diffusing computation" from "j has completed participating in the current diffusing computation", it suffices that j toggles the value of $sn.j$ whenever j starts participating in a new diffusing computation.

We can now characterize S as follows : in the current diffusing computation, each j satisfies one of the following four conditions.

1. j and $P.j$ have both started participating, and hence
   $c.j = c.(P.j) \ \wedge \ sn.j \equiv sn.(P.j)$,
2. j and $P.j$ have both completed participating, and hence
   $c.j = c.(P.j) \ \wedge \ sn.j \equiv sn.(P.j)$,
3. j has not started participating whereas $P.j$ has, and hence
   $c.j = green \ \wedge \ c.(P.j) = red \ \wedge \ sn.j \not\equiv sn.(P.j)$, or
4. j has completed participating whereas $P.j$ has not, and hence
   $c.j = green \ \wedge \ c.(P.j) = red \ \wedge \ sn.j \equiv sn.(P.j)$.

That is,

$$\begin{aligned} S \ &= \ (\forall j :: R.j) \text{ , where} \\ R.j \ &= \ (c.j = c.(P.j) \ \wedge \ sn.j \equiv sn.(P.j)) \vee (c.j = green \wedge c.(P.j) = red) \ . \end{aligned}$$

Each state predicate $R.j$ can be independently checked and satisfied by the node j, hence we consider each $R.j$ to be a separate constraint of S. We now proceed to design the closure actions.

For initiating a diffusing computation at the root node, we consider

$$c.j = green \ \wedge \ P.j = j \quad \rightarrow \quad c.j \, , sn.j := red \, , \neg sn.j$$

For propagating a diffusing computation from $P.j$ to j, we consider

$$c.j = green \ \wedge \ c.(P.j) = red \ \wedge \ sn.j \not\equiv sn.(P.j) \quad \rightarrow \quad c.j \, , sn.j := c.(P.j) \, , sn.(P.j)$$

For reflecting the diffusing computation from the children of j to j, we consider

$$c.j = red \ \wedge \ (\forall k :: \ P.k = j \ \Rightarrow \ (c.k = green \ \wedge \ sn.j \equiv sn.k)) \quad \rightarrow \quad c.j := green$$

We observe that each of these closure actions preserves each constraint in S and, hence, also preserves S.

Finally, let us design for each constraint $R.j$ a convergence action of the form

$$\neg R.j \quad \rightarrow \quad \text{“establish } R.j\text{”}$$

We propose to establish $R.j$ by updating the variables associated with node j. As a result, the constraint graph edge associated with $R.j$ will be from node $P.j$ to j and, hence, the constraint graph will be an out-tree. From Theorem 1, it follows that the resulting program will be *true*-tolerant for S. In other words, the resulting program will be stabilizing fault-tolerant.

We note that there are several statements that establish $R.j$ as proposed above. For instance, “$c.j, sn.j := c.(P.j), sn.(P.j)$” could be used or “if $c.(P.j) = red$ then $c.j := green$ else $c.j, sn.j := green, sn.(P.j)$” could be used. We prefer the former statement, since it is identical to the statement of the propagation closure action. With this choice, these two actions can be combined to yield the action

$$\begin{array}{l} \neg R.j \ \vee \\ (c.j = green \ \wedge \ c.(P.j) = red \ \wedge \ sn.j \not\equiv sn.(P.j)) \ \rightarrow \ c.j\,,\, sn.j := c.(P.j)\,,\, sn.(P.j) \end{array}$$

which is equivalent to the action

$$sn.j \not\equiv sn.(P.j) \ \vee \ (c.j = red \ \wedge \ c.(P.j) = green) \ \rightarrow \ c.j\,,\, sn.j := c.(P.j)\,,\, sn.(P.j)$$

Thus, our design yields the following stabilizing fault-tolerant program for diffusing computations:

---

program *Diffusing-computation*
process $j : 1..N$;
var $c.j : \{green, red\}$;
      $sn.j$: **boolean** ;
begin

$$c.j = green \ \wedge \ P.j = j \qquad \rightarrow \ c.j\,,\, sn.j := red\,,\, \neg sn.j$$

$\|$

$$sn.j \not\equiv sn.(P.j) \ \vee \ (c.j = red \ \wedge \ c.(P.j) = green) \ \rightarrow \ c.j\,,\, sn.j := c.(P.j)\,,\, sn.(P.j)$$

$\|$

$$\begin{array}{l} c.j = red \ \wedge \\ (\forall k :: \ P.k = j \ \Rightarrow \ (c.k = green \ \wedge \ sn.j \equiv sn.k)) \ \rightarrow \ c.j := green \end{array}$$

end

---

6 Designing Programs with Self-looping Constraint Graphs

In this section, we first identify in Theorem 2 a more general sufficient condition for the convergence validation of nonmasking fault-tolerant programs. The condition applies to programs whose convergence actions have constraint graphs with no cycles of length greater than 1. In other words, their constraint graph is either acyclic or every cycle in the graph is a self-loop. We refer to such constraint graphs as *self-looping* constraint graphs. Later in the section, we illustrate the condition by designing a nonmasking fault-tolerant program that implements an atomic action.

The basic problem with convergence validation when the constraint graph is self-looping is that if the edges in the constraint graph corresponding to two constraints have the same target node, then executing the convergence action of one of the constraints may violate the other constraint, and vice versa.

Example : For the constraint set $\{x \neq y \ , \ x \leq z\}$, consider a convergence action that changes x if x equals y, and a convergence action that changes x to be at most z if x exceeds z. For these convergence actions, it is possible that executing one can violate the constraint of the other, then executing the other can violate the constraint of the one, and so on.

In contrast to the above, consider for $x \neq y$ a convergence action that decreases x if x equals y, and for $x \leq z$ a convergence action that changes x to be at most z if x exceeds z. The first action preserves the constraint of the second action, and hence every computation of these two convergence actions is finite. □

Theorem 2 :

Let (p, S, T) be a candidate triple and q be a set of convergence actions.

If
- every closure action of p preserves each constraint in S,
- the constraint graph of q is self-looping, and
- for each node j of the constraint graph of q, the convergence actions of edges adjacent to node j can be linearly ordered so that each action in the order preserves the constraints of the preceding actions in the order,

then $p \cup q$ is T-tolerant for S.

Proof : First, we claim that for each node j in the constraint graph, every computation involving only the convergence actions of edges adjacent to node j is finite. The claim follows from an easy induction on the number of edges adjacent to j; the induction step uses the the third antecedent of the theorem (namely the existence of linear ordering of edges adjacent to j) to assert that the constraint of the first edge in the linear ordering is preserved by the convergence actions of the other edges.

Let C be an arbitrary computation of $p \cup q$. We show by induction on the rank r of the convergence actions (in q) that C has a suffix where convergence actions of rank $\leq r$ do not execute. It follows that C has a suffix where convergence actions do not execute. Hence, C has a suffix where S always holds.

Base case $(r=0)$: Observe that each of the convergence actions of rank >0 preserve the constraint of any convergence action of rank $=0$. Also, every computation involving only the convergence actions of rank $= 0$ is finite. And, by assumption, each closure action preserves each constraint in S. From these three observations, it follows that C has a suffix where convergence actions of rank ≤ 0 do not execute.

Induction case $(r > 0)$: Observe that the convergence actions of rank $> r$ preserve the constraint of any convergence action of rank $\leq r$. Also, every computation of the convergence actions of rank $= r$ is finite. And, by assumption, each closure action preserves each constraint in S. From these three observations, it follows that in any computation of $p \cup q$ where the convergence actions of rank $< r$ do not execute, there is a suffix where the convergence actions of rank $\leq r$ do not execute. From the induction hypothesis, it now follows that C has a suffix where convergence actions of rank $\leq r$ do not execute.

Recall that in our design method, every closure action (in p) and convergence action (in q) preserves S and T. It follows that $p \cup q$ is T-tolerant for S. $\Box$

We illustrate Theorem 2 by designing a program that implements an atomic action. The program is motivated by a method due to Schlichting and Schneider [2], that shows how to design certain fault-tolerant programs using a sequence of "restartable" phases. A restartable phase is one that serves as its own recovery protocol: if execution of the phase is perturbed due to the occurrence of a fault, then recovery is achieved by executing the same phase all over again. Hence, if a computation of a sequence of restartable phases is perturbed due to the occurrence of a fault, then recovery of the computation is achieved by executing the sequence of phases starting from the phase whose execution was perturbed. Typically, to implement such a sequence of restartable actions, some form of stable storage is used that is unaffected by the faults.

6.1 Atomic Action

Specification :
Desired is a program *Atomic* that satisfies the Hoare-triple:

$$\{x=M \ \wedge \ y=N\} \quad Atomic \quad \{x=h.M \ \wedge \ y=h.N\}\ ,$$

where x and y are integer variables maintained in stable storage, M and N are integer constants, and h is a function from integers to integers. Program execution is to occur in a sequence of phases, and actions in each phase may each update at most one variable in stable storage. Faults may corrupt the volatile storage of the program, but not the stable storage or the phase number.

Design :
Let us design program *Atomic* to consist of two phases. In phase zero ($ph = 0$), $h.x$ and $h.y$ are computed and respectively assigned to fresh variables u and v, that are also maintained in stable storage. In phase one ($ph=1$), u and v are respectively assigned to x and y.

Since each action updates at most one variable in shared storage, we propose that in phase zero, first, $h.x$ is assigned to u; and second, if $u=h.x$, then $h.y$ is assigned to v. Similarly, in phase one, first, u is assigned to x; and second, if $u=x$, then v is assigned to y. Let pc denote the program counter in each phase; $pc=0$ at the start of each phase.

We can now characterize the program invariant of *Atomic* as

$$\begin{aligned} S = \quad & ((ph=0 \ \Rightarrow \ x=M \wedge y=N) && \wedge \\ & (ph=1 \ \Rightarrow \ u=h.M \wedge v=h.N) && \wedge \\ & (ph=2 \ \Rightarrow \ x=h.M \wedge y=h.N) && \wedge \\ & (ph=0 \wedge pc=1 \ \Rightarrow \ u=h.x) && \wedge \\ & (ph=1 \wedge pc=1 \ \Rightarrow \ u=x)) && \end{aligned}$$

Since faults corrupt only the volatile storage and pc is the only program variable maintained in volatile storage, only the last two conjuncts of S can be violated when faults occur. Hence, we can characterize the program fault-span of *Atomic* as

$$\begin{aligned} T = \quad & ((ph=0 \Rightarrow x=M \wedge y=N) && \wedge \\ & (ph=1 \Rightarrow u=h.M \wedge v=h.N) && \wedge \\ & (ph=2 \Rightarrow x=h.M \wedge y=h.N)) && \end{aligned}$$

The last two conjuncts of S can both be independently checked and satisfied, and so we consider them to be separate constraints of S.

Following the two-phase description above, the closure actions are immediately suggested. For the first step of phase 0, we consider

$$ph=0 \ \wedge \ pc=0 \quad \rightarrow \quad u,pc := h.x, 1$$

For the second step of phase 0, we consider

$$ph=0 \ \wedge \ pc=1 \ \wedge \ u=h.x \quad \rightarrow \quad v,ph,pc := h.y, 1, 0$$

For the first step of phase 1, we consider

$$ph=1 \ \wedge \ pc=0 \quad \rightarrow \quad x,pc := u, 1$$

For the second step of phase 1, we consider

$$ph=1 \ \wedge \ pc=1 \ \wedge \ x=u \quad \rightarrow \quad y,ph := v, 2$$

We observe that each of the closure actions preserves both constraints as well as T.

Finally, we design the convergence actions for the two constraints, as follows (note that both constraints can be established by assigning the value 0 to pc, which corresponds to restarting the current phase of the program):

$$\begin{array}{lll} ph=0 \ \wedge \ pc=1 \ \wedge \ u \neq h.x & \rightarrow & pc := 0 \ , \\ ph=1 \ \wedge \ pc=1 \ \wedge \ x \neq u & \rightarrow & pc := 0 \end{array}$$

We observe that each of the convergence actions preserves T. Also, the constraint graph for the convergence actions is a single node with a self-loop. Moreover, both convergence actions preserve each other's constraint. From Theorem 2, it follows that the resulting program is T-tolerant for S.

Thus, our design yields the following program for atomic actions:

---

program *Atomic*

var $ph : 0..2;$

$pc : 0..1;$

x, y, u, v : **integer** ;

begin

$$\begin{array}{lllll} & ph=0 \ \wedge \ pc=0 & & \rightarrow & u,pc := h.x, 1 \\ [] & ph=0 \ \wedge \ pc=1 & \wedge \ u=h.x & \rightarrow & v,ph,pc := h.y, 1, 0 \\ [] & ph=0 \ \wedge \ pc=1 & \wedge \ u \neq h.x & \rightarrow & pc := 0 \\ & & & & \\ [] & ph=1 \ \wedge \ pc=0 & & \rightarrow & x,pc := u, 1 \\ [] & ph=1 \ \wedge \ pc=1 & \wedge \ x=u & \rightarrow & y,ph := v, 2 \\ [] & ph=1 \ \wedge \ pc=1 & \wedge \ x \neq u & \rightarrow & pc := 0 \end{array}$$

end

---

7 Designing Programs with Cyclic Constraint Graphs

In this section, we first identify in Theorem 3 yet another sufficient condition for the convergence validation of nonmasking fault-tolerant programs. The condition applies to programs whose convergence actions have constraint graphs with cycles of length greater than 1. Later in the section, we illustrate the condition by designing a nonmasking fault-tolerant program that implements a token ring.

Let us begin by observing that our definition of a constraint graph is sometimes "coarser" than need be: A cyclic constraint graph may become self-looping when its definition is refined to take into account (1) certain subsets of states or (2) certain subsets of convergence actions.

Consider a subset of states R of the program. If a constraint is true at each state in R, then in reasoning about states in R, the corresponding edge in the constraint graph can be ignored. We describe next three ways to use the notion of a constraint graph refined thus.

One possibility is that even if the constraint graph is cyclic, its restriction to those states where the fault-span predicate T holds may be self-looping, thereby enabling use of Theorem 2. A second possibility is that T can be partitioned into a finite number of closed subsets of states. If the constraint graph for each of these partitions is self-looping, we may use Theorem 2 with respect to each of these partitions to validate the program actions. A third possibility is that all computations converge from T to S in two stages. Let R be a closed state predicate such that $T \Leftarrow R$ and $R \Leftarrow S$. In stage one, starting from any state where T holds each computation reaches a state where R holds. In stage two, starting from any state where R holds each computation reaches a state where S holds. (Gouda and Multari call this a convergence stair of height two [13].) In this case, the constraint graph for R may be self-looping, whereas the constraint graph for T is cyclic. As a result, convergence validation may be carried out in two corresponding stages, and Theorem 2 may be used for stage two.

An alternative approach to convergence validation when the constraint graphs are cyclic is to partition the convergence actions in a hierarchical manner. If for each layer in the hierarchy the constraint graph is self-looping when it is refined to take into account only the convergence actions in that layer, then convergence validation is made feasible as follows.

Let q' be a subset of a set q of convergence actions. A *constraint graph* of q' is a directed graph that has one edge for each action in q', such that

1. Each node of the graph is labeled with a set of variables that appear in q'. Labels of nodes are mutually exclusive; thus, a variable appears in the label of only one node.
2. Each edge of the graph is labeled with an action of q'. If the action ac of q' labels the edge from node v to node w, then all variables read in ac are in the union of the labels of v and w and all variables written in ac are in the label of w.

Theorem 3 :

Let (p, S, T) be a candidate triple and q be a set of convergence actions that is partitioned into subsets numbered 0, 1, ... , $M-1$.

If
- for each partition, each closure action of p preserves each constraint in that partition whenever all constraints in lower numbered partitions hold,
- for each partition, each convergence action in higher numbered partitions preserves each constraint in that partition whenever all constraints in lower numbered partitions hold,
- for each partition, the constraint graph is self-looping, and
- for each partition, the convergence actions of edges adjacent to each node in the constraint graph of that partition can be linearly ordered so that each action in the order preserves the constraints of the preceding actions in the order,

then $p \cup q$ is T-tolerant for S. □

We illustrate Theorem 3 by designing a stabilizing distributed program that circulates the token around a ring of nodes. The program is due to Dijkstra [9].

7.1 Token ring

Cooperation between processes of a distributed system, for say resource sharing, can be achieved by passing a token around the processes of the system. In such a token passing program, the process possessing the token has the privilege to access the shared resource.

Below, we specify and design a token passing program. We choose to design closure and convergence actions that update the state of a process based on the state of that process and at most one neighboring process. Refinement of this program into one where the neighboring processes communicate via actions that either update the state of a process or read the state of neighboring process –but not both– is left as an exercise to the reader.

Specification :

Consider N+1 nodes numbered 0 through N that are organized in a ring. The successor of node j is the node $j+1$ *modulo* $N+1$. At every state, each node is either privileged or unprivileged. Desired is a program such that:

1. Exactly one node is privileged at any state.
2. Each privileged node is eventually yields its privilege to its successor in the ring.

The program should tolerate faults whereby nodes spontaneously become privileged or unprivileged.

Design :

Consider a path where node $j+1$ is adjacent from node j, $0 \leq j < N$. One way of designing a token ring program is to assign an integer value, $x.j$, to each node j so that the sequence of x values along the path from 0 to N is non-increasing and has at most one decrease in value.

We can therefore characterize the program invariant S as denoting a non-increasing sequence of x values with at most one decrease in value. In other words, $S = (\forall j :: x.j \geq x.(j{+}1)) \wedge (x.0{=}x.N \vee x.0{=}x.N+1)$. Recall that we are designing a stabilizing fault-tolerant program; hence, the program fault-span T is *true*.

We partition S into its two conjuncts. The first conjunct of S comprises the first layer of constraints, and the second conjunct comprises the second layer of constraints.

We next design the closure actions. Observe that at all S states either $x.0 = x.N$ holds —and thus all x values are equal— or $x.0{=}x.N+1$ holds —and thus the sequence of x values decreases exactly once. The first case uniquely distinguishes node 0 : we let node 0 be privileged when $x.0 = x.N$. The second case uniquely distinguishes some other node $j{+}1$: we let node $j{+}1$ be privileged when $x.j > x.(j{+}1)$.

The closure actions are immediately suggested. For node 0, we consider passing the token to node 1 with

$$x.0 = x.N \quad \rightarrow \quad x.0 := x.0 + 1$$

For node j, we consider passing the token to node $j{+}1$ with

$$x.j > x.(j{+}1) \quad \rightarrow \quad x.(j{+}1) := x.j$$

Observe that the closure actions preserve each constraint in the first conjunct of S. Also, the first closure action is not enabled when the first conjunct holds but the second does not; hence it preserves each constraint in the second conjunct when the first conjunct holds but the second does not.

Finally, we design the convergence actions. The first conjunct is established using convergence actions, as follows. For each constraint $x.j \geq x.(j+1)$ consider the convergence action

$$x.j < x.(j+1) \quad \rightarrow \quad \text{“establish } x.j \geq x.(j+1)\text{”}$$

Observe that there are several ways to establish $x.j \geq x.(j+1)$, for example by executing the statement “$x.(j+1) := x.j$”. Before we commit to any particular statement, let us design the remaining convergence actions.

Our goal is to design convergence actions for the second conjunct that preserves each of the constraints in the first layer of constraints. Unfortunately, the second conjunct of S cannot be established using convergence actions that write $x.0$ or $x.N$, since executing such actions would violate some constraint of the first layer. We therefore propose to satisfy the second conjunct by satisfying the constraints $x.j = x.(j+1)$ for each $j < N$. To establish these constraints, we consider for each j the convergence action

$$x.j > x.(j+1) \quad \rightarrow \quad x.(j+1) = x.j$$

Note that in this example the second closure action is identical to the convergence action of the second layer; hence execution of the one has the same effect as that of the other. Moreover, for both layers of constraints, the constraint graph is the path from j to $j+1$, for $0 \leq j \wedge j < N$. From Theorem 3, it follows that the resulting program is $true$-tolerant for S. In other words, the resulting program is stabilizing fault-tolerant for S.

We conclude this design with the note that if we use the statement “$x.(j+1) := x.j$” to establish $x.j \geq x.(j+1)$ in the first convergence action of node $j+1$, then the first convergence action will have the same statement as does the closure action of node $j+1$. These two actions can then be combined to yield the action

$$x.j \neq x.(j+1) \quad \rightarrow \quad x.(j+1) := x.j$$

Thus, our design yields the following program for stabilizing token rings:

---

program *Token-ring*
parameter $j : 0..N-1$;
var $x.j$: **integer** ;
begin

$$x.0 = x.N \quad \rightarrow \quad x.0 := x.0 + 1$$

$[\!]$

$$x.j \neq x.(j+1) \quad \rightarrow \quad x.(j+1) := x.j$$

end

---

8 Concluding Remarks

In this paper, we have developed a method for the design of nonmasking fault-tolerant programs. The method distinguishes two types of program actions and exploits the dependencies between the constraints, that in conjunction with the program fault-span equivale the program invariant, so as to satisfy the closure and convergence properties necessary for nonmasking fault-tolerance.

Closure properties are a special class of the safety properties of programs. Safety properties are used in design methods to constrain the design of program actions. For example, an invariant predicate constrains the design of program actions so that the program specification can be satisfied. In this paper, we have shown how a fault-span predicate constrains the design of program actions so that nonmasking fault-tolerance can be achieved.

Convergence properties are a special class of the progress properties of programs. The standard approach for proving that computations of a program progress towards satisfying some state predicate is to exhibit a "variant" function. A variant function is a mapping from the program state space to a set that is wellfounded under a relation $<$, such that in each step of the computation the variant function value does not increase (with respect to $<$) and eventually decreases, until the desired state predicate is satisfied. Exhibiting variant functions is a nontrivial task, especially for concurrent programs. In this paper, we have shown how to simplify the problem of exhibiting variant functions, in terms of the sufficient conditions on the closure and convergence actions and the constraint graph.

To further develop the methodology, one issue that we are studying is the role of fairness. The fairness requirement on program computations is often unnecessary. (In fact, each of the programs derived in this paper is correct even when the fairness requirement is ignored; to see this, observe that each computation of the closure actions is either finite or has a state where S holds [the argument for the token-ring program requires an additional similar requirement for the convergence actions of each layer].) Hence, it will be useful to characterize sufficient conditions under which program actions guarantee convergence to S without requiring fairness.

Finally, we note that it will also be useful to develop systematic methods of refining programs that preserve the property of convergence to fault-free states. For instance, recall that one of the closure actions in the stabilizing diffusing computation involves accessing the state of a node and all its children nodes in the out-tree. This action has high atomicity and may therefore be unsuitable for a distributed implementation. In [4], we present a refinement of this system that yields actions with low atomicity and preserves the property of convergence. We study refinement issues in a companion paper.

Acknowledgements. We thank Edsger Dijkstra, Jayadev Misra, members of the Austin Tuesday Afternoon Club, Ernie Cohen, and Boaz Patt for their comments on earlier drafts of this paper.

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journal of the ACM*, 32(2) (1985), pp. 374-382.

[2] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems", *ACM Transactions on Computers* (1983), pp. 222-238.

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall (1976).

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

[5] A. Arora, "A foundation of fault-tolerant computing", *Ph.D. Dissertation*, The University of Texas at Austin (1992).

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

[7] D. Gries, *The Science of Programming*, Springer-Verlag (1981).

[8] F. Cristian, "A rigorous approach to fault-tolerant programming", *IEEE Transactions on Software Engg.* 11(1), (1985).

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

[10] F. B. Bastani, I.-L. Yen, and I.-R. Chen, "A class of inherently fault-tolerant distributed programs", *IEEE Transactions on Software Engg.* 14(10) (1988), pp. 1431-1442.

[11] J. E. Burns and J. Pachl, "Uniform stabilizing rings," *ACM Transactions on Programming Languages and Systems* 11(2) (1989), pp. 330-344.

[12] A. Arora and M. G. Gouda, "Distributed reset", *IEEE Transactions on Computers* (1994), to appear.

[13] M. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers* 40(4) (1991), pp. 448-458.

[14] G. Varghese, "Self-stabilization by local checking and correction", *Ph.D. Dissertation*, Massachusetts Institute of Technology (1992).