Closure and Convergence: A Foundation of Fault-Tolerant Computing

Anish ARORA

Department of Computer Science
The Ohio State Univ. at Columbus
2036 Neil Avenue Mall, OH 43210
614-292-1836, Fax: 614-292-2911
anish@cis.ohio-state.edu

Mohamed GOUDA

Department of Computer Sciences
The Univ. of Texas at Austin
2.128 Taylor Hall, TX 78712
512-471-9532, Fax: 512-471-8885
gouda@cs.utexas.edu

Abstract

We give a formal definition of what it means for a system to "tolerate" a class of "faults". The definition consists of two conditions: One, if a fault occurs when the system state is within a set of "legal" states, the resulting state is within some larger set and, if faults continue occurring, the system state remains within that larger set (*Closure*). And two, if faults stop occurring, the system eventually reaches a state within the legal set (*Convergence*). We demonstrate the applicability of our definition for specifying and verifying the fault-tolerance properties of a variety of digital and computer systems. Further, using the definition, we obtain a simple classification of fault-tolerant systems and discuss methods for their systematic design.

Keywords: Fault-tolerance, Reliability, Algorithms, Verification, Design.

Additional Keywords: Masking, Stabilizing, Closure, Convergence.

1 Introduction

Fault-tolerant computing has traditionally been studied in the context of specific technologies, architectures, and applications. One consequence of this tradition is that several subdisciplines of fault-tolerant computing have emerged that are apparently unrelated to each other. These subdisciplines deal with specific classes of faults, employ distinct models and design methods, and have their own terminology and classification [10, 26, 39]. As a result, the discipline itself appears to be fragmented.

Another consequence of this tradition is that verification of fault-tolerant systems is often based on implementation-specific artifacts—such as stable storage, timeouts, and shadow registers—without explicitly specifying what properties of these artifacts are necessary. Such verification is imprecise and hence unsuitable, especially for safety-critical systems.

To redress the problems described above, some efforts have been made in the last decade. These efforts have mainly focussed on classifying fault-tolerant systems, and have yielded two noteworthy classifications. One is based on a distinction between the notions of faults, errors, and failures: faults in a physical domain can cause errors in an information domain, whereas errors in an information domain can cause failures in an external domain [1, 6, 27]. (Unfortunately, these notions are subjective: "what one person calls a failure, a second person calls a fault, and a third person might call an error" [13].) The other is based on what type of fault is tolerated, for example, stuck-at, crash, fail-stop, omission, timing, or byzantine faults [19, 30, 32, 36].

A few efforts have also been made to formally define and verify system fault-tolerance [14, 29, 30, 32], albeit with limited scope. More specifically, these efforts have considered systems that recover from the occurrence of faults and terminate properly. In other words, they have considered systems whose input-output relation masks faults. Alternative forms of fault-tolerance that do not always mask faults have rarely been considered. Such forms of fault-tolerance ensure the continued availability of systems by repairing faulty system parts or by correctly restoring the system state whenever the system exhibits incorrect behavior due to the occurrence of faults.

One form of fault-tolerance that does not always mask faults is self-stabilization [33]. While self-stabilization was first studied in computing science in 1973 [15], and its application to fault-tolerance was strongly endorsed in 1983 [25], it is only in the last few years that concerted efforts have been made to relate self-stabilization to fault-tolerance

[7, 9, 11]. Even so, self-stabilizing systems are mainly being designed to tolerate arbitrary transient faults, whereas they can be designed to tolerate a variety of fault types [4, 24, 40, 41].

In summary, a survey of the literature reveals that there is a well-defined need for (i) a uniform definition of fault-tolerance, and (ii) methods for designing and verifying system fault-tolerance independent of technology, architecture, and application.

1.1 Overview

In this paper, we first give a uniform definition of what it means for a system to tolerate a class of faults. Our definition consists of two conditions: one of closure and another of convergence.

To motivate the closure condition, let us observe that a well-established method for verifying fault-free systems is to exhibit a predicate that is true throughout system execution [16, 22]. Such an "invariant" predicate identifies the "legal" system states and asserts that the set of legal states is closed under system execution. Following this method, we require that for each fault-tolerant system there exists a predicate S that is invariant throughout fault-free system execution.

Next, we observe that faults—be they stuck-at, crash, fail-stop, omission, timing, or byzantine—can be systematically represented as actions that upon execution perturb the system state [14]. In other words, even when the effect of a fault is not transient, but is permanent or intermittent, the fault can be represented as an action. For example, consider a wire that can be permanently stuck at low voltage. Such a wire can be represented by the following program. Let *in* and *out* be two variables that range over $\{0,1\}$, and let *broken* be a boolean variable. The correct behavior of the wire can be described by a program action that sets *out* to *in* provided that $out \neq in$ holds and the state of the wire is $\neg broken$. That is,

$$out \neq in \ \wedge \ \neg broken \ \rightarrow \quad out := in$$

If a fault occurs, the incorrect behavior of the wire can be described by the program action that sets *out* to 0 provided that the state of the wire is *broken*. That is,

$$broken \qquad\qquad \rightarrow \quad out := 0$$

For this two-action program, the predicate S is $\neg broken$ and the stuck-at-low-voltage fault can be represented by the fault action

$$\neg broken \quad \rightarrow \quad broken := true$$

Now, if the wire can also be "unstuck" then, in addition to the fault action above, we need to consider the fault action

$$broken \quad \rightarrow \quad broken := false$$

Further, if the wire can also exhibit byzantine behavior (that is, it can intermittently and nondeterministically sets *out* to 0 or 1) after being stuck then, in addition to the program actions above, we need to consider the program action

$$broken \quad \rightarrow \quad out := 1$$

Having thus represented faults as actions, we next characterize what happens when a system is perturbed into an illegal state due to the execution of a fault action. We require that for each fault-tolerant system there exists a predicate T that is weaker than S and is invariant under the execution of system and fault actions. In other words, we require that once fault actions start executing, the system state necessarily satisfies T. Thus, T defines the extent to which fault actions can perturb the legal states during system execution.

The requirement that predicates S and T exist constitutes the closure condition. We are now ready to motivate the convergence condition. Once fault actions stop executing, the system can achieve progress only if it is restored to a state where S holds. Therefore, we require that every fault-free system execution, upon starting from any state where T holds, eventually reaches a state where S holds. This requirement constitutes the convergence condition.

We define fault-tolerance formally in the next section. We then go on to show how the fault-tolerance properties of digital and computing systems can be specified, verified, and designed independent of technology, architecture, or application. In particular, the issues we consider include how to use our definition to:

- classify the fault-tolerance of a system,
- verify that a system is fault-tolerant,
- verify that a system is fault-intolerant,
- prove there is no system that both meets a specification and is fault-tolerant,
- design a system that both meets a specification and is fault-tolerant.

We emphasize that reported here are only a few of the applications that we have developed over the last three years. A detailed report of these applications appears in [2].

We proceed as follows. In Section 2, we give a formal definition of what it means for a program to be fault-tolerant and present a formal classification of fault-tolerant programs. Using the definition, we illustrate: in Section 3, how to verify that a program is fault-tolerant; in Section 4, how to verify that a program is fault-intolerant; in Section 5, how to prove that there is no fault-tolerant program that meets a given specification; and in Section 6, how to design programs to be fault-tolerant. Finally, we discuss some questions raised by our approach in Section 7 and make concluding remarks in Section 8.

2 Defining Fault-Tolerance

Towards giving a formal definition of what it means for a program to tolerate a set of faults, we first discuss the notion of a program and define two program properties: closure and convergence.

2.1 Closure and Convergence in Programs

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

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

where the guard is a boolean expression over program variables, and the statement updates zero or more 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. If a state predicate evaluates to *true* at some state, we say the state predicate *holds* at that state. An action is *enabled* at a state iff its guard holds at that state. A process is *enabled* at a state iff some action in the process is enabled at that state.

Definition 1:

Let S be a state predicate of p.

S is *closed* in p iff for each action $B \to st$ in each process of p, executing st starting from a state where $B \wedge S$ holds results in a state where S holds.

We assume nondeterministic interleaving semantics. A *computation* of p is a sequence of states that satisfies the following three conditions: (i) for each consecutive pair of states c followed by d in the sequence, there exists an action $B \to st$ in some process of p such that B holds at c and executing st starting from c results in d; (ii) the sequence is maximal, i.e., the sequence is either infinite or (it is finite and) no action is enabled in the last state; and (iii) the sequence is process-fair, i.e., if any process j of p is continuously enabled along the sequence, then eventually some action of j is chosen for execution.

Definition 2:

Let S and T be state predicates of p.

T *converges* to S in p iff

- S is closed in p,
- T is closed in p, and
- in each computation of p starting at any state where T holds, there exists a state where S holds.

2.2 Fault-Tolerance

Recall from Section 1.1 that in defining fault-tolerance we can represent the faults that affect a program p by a set of actions F over the variables of p.

Definition 3:

Let S be a closed state predicate of p, and F be a set of actions over variables of p.

p is *F-tolerant* for S iff there exists a state predicate T of p such that

- T holds at every state where S holds; i.e., $S \Rightarrow T$,
- for each action $B \to st$ in F, executing st starting from a state where $B \wedge T$ holds results in a state where T holds, and
- T converges to S in p.

Definition 3 can be understood as follows. Let c be any state of p where S holds. Since S is closed, executing any enabled action in p starting from c yields a state where S holds.

However, executing any enabled action in F starting from c may yield a state where $\neg S$ holds. In this case, Definition 3 guarantees three facts about the resulting state: (i) some state predicate T holds, (ii) subsequent execution of actions in p and F yields states where T holds, and (iii) subsequent execution of actions in p alone eventually yields a state where S holds.

Thus, Definition 3 states that if the intended domain of execution of p is all states where S holds then p tolerates the fault actions in F as follows. Once fault actions in F stop executing, execution of actions in p alone yields a state where S holds, and from this point the program resumes its intended execution.

2.3 Extremal Solutions

Observe that there may exist several state predicates T of a program p that satisfy the three conditions of Definition 3. We now show that if there exists at least one such state predicate, then there exists a strongest one Ts and a weakest one Tw.

Existence of Ts : Let Ts be the conjunction of all state predicates T of p that satisfy the three conditions in Definition 3. We show that Ts also satisfies the three conditions.

$$\begin{array}{ll}
 & S \Rightarrow Ts \\
= & \{ \text{ definition of } Ts \ \} \\
 & S \Rightarrow (\forall T : T) \\
= & \{ \text{ predicate calculus } \} \\
 & (\forall T : S \Rightarrow T) \\
= & \{ \ S \Rightarrow T, \text{ for all } T \ \} \\
 & true
\end{array}$$

$$\begin{array}{ll}
 & (\forall\, B \rightarrow st \text{ in } F \ : \ \text{executing } st \text{ in a state where } B \wedge Ts \text{ holds preserves } Ts \ ) \\
= & \{ \text{ definition of weakest precondition of action } B \rightarrow st \text{ for state predicate } Ts, \\
 & wp.(B \rightarrow st).Ts, \text{ where the infix operator ``.'' denotes function application } \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ Ts \Rightarrow wp.(B \rightarrow st).Ts) \\
= & \{ \text{ definition of } Ts \ \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ (\forall T : T) \Rightarrow wp.(B \rightarrow st).(\forall T : T)) \\
= & \{ \ wp.(B \rightarrow st) \text{ is universally conjunctive if } st \text{ always terminates } \} \\
 & (\forall\, B \rightarrow st \text{ in } F \ : \ (\forall T : T) \Rightarrow (\forall T : wp.(B \rightarrow st).T)) \\
\Leftarrow & \{ \text{ Leibniz [18] } \}
\end{array}$$

$$
\begin{array}{cl}
 & (\forall\, B \rightarrow st \text{ in } F\ :\ (\forall T : T \Rightarrow wp.(B \rightarrow st).T)) \\
= & \{\ T \text{ is closed in } F, \text{ for all } T\ \} \\
 & true
\end{array}
$$

Ts converges to S in p since: (i) S is closed in p, (ii) Ts is closed in p (replace F with p in the proof above), and (iii) at each state where Ts holds some T holds (in fact, every T holds), and that T converges to S.

Existence of Tw : Let Tw be the disjunction of all state predicates T of p that satisfy the three conditions in Definition 3. We show that Tw also satisfies the three conditions.

$$
\begin{array}{cl}
 & S \Rightarrow Tw \\
= & \{\ \text{definition of } Tw\ \} \\
 & S \Rightarrow (\exists T : T) \\
= & \{\ \text{there is at least one such } T, \text{ predicate calculus}\ \} \\
 & (\exists T : S \Rightarrow T) \\
= & \{\ S \Rightarrow T, \text{ for all } T\ \} \\
 & true
\end{array}
$$

$$
\begin{array}{cl}
 & (\forall\, B \rightarrow st \text{ in } F\ :\ \text{executing } st \text{ in a state where } B \wedge Tw \text{ holds preserves } Tw\ ) \\
= & \{\ \text{definition of weakest preconditions [18]}\ \} \\
 & (\forall\, B \rightarrow st \text{ in } F\ :\ Tw \Rightarrow wp.(B \rightarrow st).Tw) \\
\Leftarrow & \{\ T \Rightarrow Tw \text{ for all } T,\ wp.(B \rightarrow st) \text{ is monotonic [18], predicate calculus}\ \} \\
 & (\forall\, B \rightarrow st \text{ in } F\ :\ Tw \Rightarrow (\exists T : wp.(B \rightarrow st).T)) \\
= & \{\ \text{definition of } Tw\ \} \\
 & (\forall\, B \rightarrow st \text{ in } F\ :\ (\exists T : T) \Rightarrow (\exists T : wp.(B \rightarrow st).T)) \\
= & \{\ \text{predicate calculus}\ \} \\
 & (\forall\, B \rightarrow st \text{ in } F\ :\ (\exists T : T \Rightarrow wp.(B \rightarrow st).T)) \\
= & \{\ T \text{ is closed in } p, \text{ for all } T\ \} \\
 & true
\end{array}
$$

Tw converges to S in p since: (i) S is closed in p, (ii) Tw is closed in p (replace F with p in the proof above), and (iii) at each state where Tw holds some T holds, and that T converges to S.

Observe that Ts characterizes the largest set of states that are reachable by executing actions in p and F upon starting from states where S holds. In other words, Ts characterizes the extent to which the program state can be perturbed due to occurrence of fault actions. In contrast, Tw characterizes the largest set of states from which convergence to S is guaranteed.

Two situations where the extremal solutions are easily computed deserve mention here.

- When S is closed in F, S satisfies the three conditions in Definition 3 and, hence, $Ts = S$.
  In this situation, regardless of how actions in F perturb the program state, an illegal state is never observed.
- When $true$ converges to S in p, $true$ satisfies the three conditions in Definition 3 and, hence, $Tw = true$.
  In this situation, regardless of how actions in F perturb the program state, subsequent execution of actions in p is guaranteed to eventually reach a legal state again.

2.4 A Classification

Based on these two situations where the extremal solutions Ts and Tw are easily calculated, we introduce the following terminology for describing the fault-tolerance of p relative to S:

> If $Ts = S$
> then p has *Masking* fault-tolerance
> else p has *Nonmasking* fault-tolerance.
>
> If $Tw = true$
> then p has *Global Stabilizing* fault-tolerance
> else p has *Local Stabilizing* fault-tolerance.

The following four classes of fault-tolerant programs are immediately suggested:

- Masking and Global Stabilizing
- Masking and Local Stabilizing
- Nonmasking and Global Stabilizing
- Nonmasking and Local Stabilizing.

We present in the following sections examples of programs that belong to each class.

3 Verifying Fault-Tolerance

In this section, we present three examples that illustrate how our definition can be used to verify whether a program is tolerant of a set of faults. The first example deals with faults whose effect is of a permanent nature; the second deals with faults whose effect is of a transient nature; and the third deals with faults whose effect is of an intermittent nature. We have presented several other examples in [2] and [3].

3.1 Example: Atomic Commitment Protocol

Specification [8]

Each process casts one of two votes, Yes or No, then reaches one of two decisions, Commit or Abort, such that:

1. If no faults occur and all processes vote Yes, all processes reach a Commit decision.
2. A process reaches a Commit decision only when all processes voted Yes.
3. All processes that reach a decision reach the same decision.

Faults may stop or restart processes.

Two-Phase Commit Protocol

As its name suggests, this protocol consists of two phases. In the first phase, each process casts its vote and sends the vote to a distinguished "coordinator" process c. In the second phase, the coordinator reaches a decision based on the votes received, and broadcasts the decision to all processes.

Process c has three actions. In the first action, c casts its vote, enters the second phase, and starts waiting for the votes of other processes. In the second action, c detects that all processes have voted Yes, and reaches a Commit decision. In the third action, c detects that some process has voted No or has stopped, and reaches an Abort decision.

Each process j other than the coordinator has three actions. In the first action, j detects that c has voted and casts its vote. In the second action, j detects that c has stopped and reaches an Abort decision. In the third action, j detects that some process has completed its second phase and reaches the same decision as that process has.

For each process j, let

- $ph.j$ be the current phase of j; $ph.j$ is 0 initially, 1 after j has cast its vote, and 2 after j has reached a decision,
- $d.j$ be (depending upon the current phase) the vote or the decision of j; $d.j$ is $true$ if the vote is Yes or the decision is Commit and $false$ if the vote is No or the decision is Abort,
- $up.j$ be the current status of j; $up.j$ is $true$ if j is executing and $false$ if j is stopped.

Remark on programming notation:

We use "?" to denote nondeterministic choice. Thus, "x :=?" means that x is assigned a nondeterministically chosen value from its domain.

Also, we use parameters to abbreviate a set of actions as one parameterized action. For example, let m be a parameter whose value is 0, 1 or 2; then the parameterized action $act.m$ abbreviates the following set of three actions.

$$act.(m := 0) \;[\!]\; act.(m := 1) \;[\!]\; act.(m := 2)$$

The domain of each parameter is finite.

We adopt the following binding power of logical connectives (in decreasing order):

$$\neg$$
$$\vee \,,\, \wedge$$
$$\Rightarrow \,,\, \Leftarrow$$
$$\equiv \,,\, \not\equiv$$

Thus, $p{\vee}q{\Rightarrow}\neg r$ equivales $(p{\vee}q) \Rightarrow (\neg r)$. We sometimes use blank spaces as delimiters. Thus, $p \;\vee\; q{\Rightarrow}\neg r$ equivales $p \vee (q{\Rightarrow}(\neg r))$. (End of Remark)

The Two-phase protocol is described formally in the following program, along with the set of faults it tolerates.

program *Two-phase*
constant X : *set* of ID;
$\quad\quad c : X$;
var $\quad ph$: array X of 0..2;
$\quad\quad up$: array X of $boolean$;
$\quad\quad d$: array X of $boolean$;
process $j\ :\ X$;
parameter $k : X$;
begin

$$
\begin{array}{lll}
 & j=c \ \wedge\ up.j\ \wedge\ ph.j=0 & \rightarrow\ ph.j\,,\, d.j := 1\,,\,?\\
[\!] & j=c \ \wedge\ up.j\ \wedge\ ph.j=1\ \wedge\ (\forall l \in X : up.l\ \wedge\ ph.l=1\ \wedge\ d.l) & \rightarrow\ ph.j\,,\, d.j := 2\,,\, true\\
[\!] & j=c \ \wedge\ up.j\ \wedge\ ph.j=1\ \wedge\ (\exists l \in X : \neg up.l \vee (ph.l \geq 1\ \wedge \neg d.l)) & \rightarrow\ ph.j\,,\, d.j := 2\,,\, false\\
 & & \\
[\!] & j\neq c \ \wedge\ up.j\ \wedge\ ph.j=0\ \wedge\ (up.c\ \wedge\ ph.c=1) & \rightarrow\ ph.j\,,\, d.j := 1\,,\,?\\
[\!] & j\neq c \ \wedge\ up.j\ \wedge\ ph.j=0\ \wedge\ \neg up.c & \rightarrow\ ph.j\,,\, d.j := 2\,,\, false\\
[\!] & j\neq c \ \wedge\ up.j\ \wedge\ ph.j<ph.k\ \wedge\ (up.k\ \wedge\ ph.k=2) & \rightarrow\ ph.j\,,\, d.j := 2\,,\, d.k
\end{array}
$$

end

---

faults F

$$\{true \qquad \rightarrow\ up.j := \neg up.j\}$$

We show that program *Two-phase* is F-tolerant for S, where

$$
\begin{array}{rlll}
S\ = & & ph.c=0 & \Rightarrow\ (\forall j :\ ph.j=0\ \vee\ (ph.j=2\ \wedge\ \neg d.j))\\
 & \wedge & ph.c=1 & \Rightarrow\ (\forall j :\ ph.j\neq 2\ \vee\ \neg d.j)\\
 & \wedge & ph.c=2\ \wedge\ d.c & \Rightarrow\ (\forall j :\ ph.j\neq 0\ \wedge\ d.j)\\
 & \wedge & ph.c=2\ \wedge\ \neg d.c & \Rightarrow\ (\forall j :\ ph.j\neq 2\ \vee\ \neg d.j\ )
\end{array}
$$

Informally, S states that the domain of execution of program *Two-phase* satisfies the following four conditions. (i) If c has not voted ($ph.c=0$), then each process has either not voted or (detected that c had stopped and) reached an Abort decision. (ii) If c has voted but not reached a decision ($ph.c=1$), then each process has either not reached a decision or (detected that c had stopped and) reached an Abort decision. (iii) If c has reached a Commit decision ($ph.c=2\ \wedge\ d.c$), then each process has either voted Yes (and not reached a decision) or reached a Commit decision. (iv) If c has reached an Abort decision ($ph.c=2 \wedge \neg d.c$), then each process has either not reached a decision or reached an Abort decision.

It can be shown that each computation of program *Two-phase* that starts at a state where S holds satisfies the atomic commitment specification. (Details appear in [2].)

Proof

To show that program *Two-phase* is F-tolerant for S, we are required to exhibit a state predicate T that satisfies the three conditions in Definition 3. In this case, we let T to be S itself. It therefore remains to show that S is closed in *Two-phase* as well as in F.

S is closed in Two-phase :
For arbitrary j, we show that each conjunct of S is preserved under execution of program actions starting from a state where S holds.

The first conjunct of S is preserved: by executing the first three actions, since they falsify $ph.c=0$; by executing the fourth action, since it is not enabled when $ph.c=0$; and by executing the fifth and the sixth action, since they truthify $ph.j=2 \wedge \neg d.j$.

The second conjunct of S is preserved: by executing the first action, since it truthifies $(\forall j : ph.j \neq 2 \vee \neg d.j)$; by executing the next two actions, since they falsify $ph.c=1$; by executing the fourth action, since it truthifies $ph.j \neq 2$; and by executing the last two actions, since they truthify $\neg d.j$.

The third conjunct of S is preserved: by executing the first action, since it is not enabled when $ph.c=2$ nor does it establish $ph.c=2$; by executing the second action, since it truthifies $(\forall j : ph.j \neq 0 \wedge d.j)$; by executing the third action, since it truthifies $\neg d.c$; by executing the next two actions, since they are not enabled when $ph.c=2$; and by executing the sixth action, since it truthifies $ph.j \neq 0 \wedge d.j$.

The last conjunct of S is preserved: by executing the first action, since it is not enabled when $ph.c=2$ nor does it establish $ph.c=2$; by executing the second action, since it truthifies $d.c$; by executing the third action, since it preserves $(\forall j : ph.j \neq 2 \vee \neg d.j)$; by executing the fourth action, since it is not enabled when $ph.c=2$; and by executing the last two actions since they truthify $\neg d.j$. $\square$

S is closed in F :
S does not name any *up* variables; hence S is closed in F. $\square$

Since the predicate T is S, the strongest solution Ts is S and, hence, *Two-phase* is masking fault-tolerant. Also, it is straightforward to show that *true* does not converge to S and, hence, that *Two-phase* is local stabilizing fault-tolerant. $\Box$

Remarks

Existing two-phase commit protocols require three modes of execution: a "normal" mode is used when faults do not occur, a "termination" mode is used when the coordinator stops, and a "recovery" mode is used when a process restarts. In contrast, our protocol does not require different modes of operation.

Proofs of correctness of existing protocols rely heavily on implementation details, such as stable storage (nonvolatile memory) and timeouts. In contrast, the proof of our protocol does not rely on implementation details.

Not relying on implementation details does not mean that our protocol is unsuitable for studying implementation issues. For example, how would we implement that S is closed in F? Clearly, one way would be to ensure that the ph and d variables are not corrupted when fault actions occur; this is readily achieved if the ph and d variables are kept in stable storage. As another example, how would we implement the detection of $up.c$? One way to detect $up.c$ would be to receive a message from c and, likewise, one way to detect that $\neg up.c$ holds is to use a timer and to timeout if no message from c is received.

Finally, the guards of some actions in our protocol access variables that are updated by more than one process. Furthermore, it is assumed that, for each action, the evaluation of its guard and the execution of its assignment statement is instantaneous. These "high atomicity" assumptions are not necessary: the program remains fault-tolerant even if (i) the variables of different processes are accessed separately during the evaluation of the guards, and (ii) the evaluation of the guards is done separately from the execution of the assignment statements.

3.2 Example: Data Transfer Protocol

Specification

An infinite input array is to be copied to an infinite output array. Items from the input array are to be sent by a *sender* process to a *receiver* process via a bidirectional channel. Faults may lose channel messages.

Sliding-window Protocol

In the sliding-window protocol, the *sender* process associates an identifier with each item it sends. When an item is received by the *receiver* process, it is accepted provided its identifier is the one expected; if accepted, an acknowledgement of the item is sent to *sender*. There can be at most $W-1$ unacknowledged items at any time, hence a $log\, W$-bit identifier suffices.

Process *sender* has three actions. In the first action, *sender* sends an item provided it has sent less than $W-1$ items that are currently unacknowledged. In the second action, *sender* receives an acknowledgement and prepares to send the next item. In the third action, *sender* detects the loss of messages and resends all the items that are currently unacknowledged.

Process *receiver* has two actions. In the first action, *receiver* sends an acknowledgement for the item last received and starts waiting for the next item. In the second action, *receiver* receives an item, and accepts it if the identifier is the one expected.

Let

- cs be the channel from *sender* to *receiver*,
- cr be the channel from *receiver* to *sender*,
- ns be the number of items sent by *sender*,
- nr be the number of items received by *receiver*,
- na be the number of items whose acknowledgement has been received by *sender*,
- bs be the $log\, W$-bit identifier of the item to be sent next,
- br be the $log\, W$-bit identifier of the item to be received next,
- ba be the $log\, W$-bit identifier of the item to be acknowledged next,
- rr be a binary valued control variable maintained by the receiver,
- $\oplus$ and $\ominus$ be, respectively, addition and subtraction modulo W, and
- & be sequence composition.

The sliding-window protocol is described formally in the following program, along with the set of faults it tolerates.

---

program *Sliding-window*
var cs, cr : **sequence** of **integer** ;
$rr : 0..1$;
ns, na, nr : **integer** ;
$bs, ba, br : 0..W-1$;

process *sender*
begin

$$
\begin{array}{lll}
 & ns < na+(W-1) & \to \quad ns, bs, cs := ns+1\,,\ bs \oplus 1\,,\ cs \& bs \\
[\!] & cr \neq \langle\rangle & \to \quad \textbf{if } head.cr \in ba..bs \ominus 1 \\
 & & \qquad \textbf{then } na, ba := na+(head.cr \ominus ba)+1\,,\ head.cr \oplus 1 \ \textbf{fi}\ ; \\
 & & \qquad cr := tail.cr \\
[\!] & cs = \langle\rangle \ \wedge\ cr = \langle\rangle\ \wedge & \\
 & rr = 0 \ \wedge\ ns > na & \to \quad cs := cs \& (ba..bs \ominus 1)
\end{array}
$$

end

process *receiver*
begin

$$
\begin{array}{lll}
 & rr = 1 & \to \quad rr, cr := 0, cr \& br \ominus 1 \\
[\!] & cs \neq \langle\rangle & \to \quad \textbf{if } head.cs = br \ \textbf{then } nr, br, rr := nr+1\,,\ br \oplus 1\,,\ 1 \ \textbf{fi}\ ; \\
 & & \qquad cs := tail.cs
\end{array}
$$

end

---

faults F

$$
\begin{array}{lll}
\{ & cs \neq \langle\rangle & \to \quad cs := tail.cs\,, \\
 & cr \neq \langle\rangle & \to \quad cr := tail.cr \ \}
\end{array}
$$

We show that program *Sliding-window* is F-tolerant for S, where

$$
\begin{array}{lllll}
S \ = & cs = br..bs \ominus 1 & \wedge\ cr \text{ is a subsequence of } ba..br \ominus rr \ominus 1 & & \wedge \\
 & na \le nr & \wedge\ nr \le ns & \wedge\ ns \le na+(W-1) & \wedge \\
 & bs = (ns \ mod \ W) & \wedge\ br = (nr \ mod \ W) & \wedge\ ba = (na \ mod \ W) &
\end{array}
$$

Informally, S states that the domain of execution of program *Sliding-window* satisfies the following five conditions. (i) Channel cs contains the in-order sequence of items that have been sent but not yet received. (ii) Channel cr contains an in-order sequence of acknowledgments that have been sent but not yet received. (iii) The number of acknowledgements received is at most the number of items received, which in turn is at most the number of items sent. (iv) The number of unacknowledged items is at most $W-1$. (v) The three $log\,W$-bit identifiers are correct.

It is straightforward to show that each computation of program *Sliding-window* that starts at a state where S holds satisfies the data transfer specification. (Details appear in [2].)

Proof

To show that program *Sliding-window* is F-tolerant for S, we are required to exhibit a state predicate T that satisfies the three conditions in Definition 3. In this case, we let T to be

$$
\begin{array}{llll}
T\ = & cr\&cs \text{ is a subsequence of } ba..br\ominus rr\ominus 1\&br..bs\ominus 1 & & \wedge \\
& cr \text{ is a subsequence of } ba..br\ominus rr\ominus 1 & & \wedge \\
& na\leq nr \quad\quad\quad\ \wedge\ nr\leq ns & \wedge\, ns\leq na+(W-1) & \wedge \\
& bs=(ns\ mod\ W)\ \wedge\ br=(nr\ mod\ W) & \wedge ba=(na\ mod\ W) &
\end{array}
$$

It remains to show that S is closed in *Sliding-window*, T is closed in *Sliding-window* as well as in F, and T converges to S in *Sliding-window.*

S is closed in Sliding-window :
Executing the first action of *sender* preserves $cs = br..bs\ominus 1$, $bs = (ns\ mod\ W)$, and $nr\leq ns \wedge ns\leq na{+}(W{-}1)$, and does not modify the variables in the remaining conjuncts. Executing the second action of *sender* preserves the second conjunct, $ba=(na\ mod\ W)$, and $na\leq nr\ \wedge\ ns\leq na+(W-1)$, and does not modify the variables in the remaining conjuncts. The third action of *sender* is not enabled at any state where S holds.

Executing the first action of *receiver* preserves the second conjunct, and does not modify the variables in the remaining conjuncts. Executing the second action of *receiver* preserves $cs=br..bs\ominus 1$, $br=(nr\ mod\ W)$, and $na\leq nr\ \wedge\ nr\leq ns$, and does not modify the variables in the remaining conjuncts.

T is closed in Sliding-window :
Similar to the proof of S is closed in *Sliding-Window*.

T is closed in F :
Actions in F do not add new messages in cs or cr nor do they update any other variable.

T converges to S in Sliding-window :
Consider an arbitrary state where T holds. We consider three cases for this state: (a) $cs = br..bs \ominus 1$, (b) some item in cs has identifier less than br, and (c) no item in cs has identifier less than br, but some item in $br..bs \ominus 1$ is missing in cs.

Case (a). S holds at the state.

Case (b). Due to fair execution of actions of *receiver*, all items in cs with identifier less than br will be received by receiver, thereby yielding a state where case (a) or (c) apply.

Case (c). Due to fair execution of actions of *receiver*, br will eventually be the identifier of the first item missing in cs. Subsequently, as long as an item is missing in cs, br and nr will not be updated and items received from cs will not be accepted. Since $na \leq nr$, eventually na will no longer be updated and the first action of *sender* will no longer be enabled. Hence, eventually cs will be empty, thereafter rr will be 0, and cr will be empty. Therefore, the third action of *sender* will be executed, yielding a state where $cs = ba..bs \ominus 1$ and $cr = \langle\rangle$ holds. Due to fair execution of actions of *receiver*, cs will eventually be the sequence $br..bs \ominus 1$, yielding a state where case (a) applies.

Since S is not closed in F, the strongest solution Ts is weaker than S and, hence, *Sliding-window* is nonmasking fault-tolerant. Also, it is straightforward to show that *true* does not converge to S and, hence, that *Sliding-window* is local stabilizing fault-tolerant. □

Remarks

The guard of the third action of *sender* involves detecting the global state of the system. One way to implement this detection is to use a timer and to timeout if no acknowledgement is received within the time taken to append W messages to cs plus the maximum roundtrip delay of a message.

3.3 Example: Byzantine Agreement

Specification

Each process is either Reliable or Unreliable. Each Reliable process reaches one of two decisions, $false$ or $true$. One process g is distinguished, and has associated with it a boolean value B. It is required that:

1. If g is Reliable, the decision value of each Reliable process is B.
2. All Reliable processes eventually reach the same decision.

Faults may make Reliable processes Unreliable.

Program [12, 37]

We assume authenticated communication: messages sent by Reliable processes are correctly received by Reliable processes, and Unreliable processes cannot forge messages on behalf of Reliable processes.

Agreement is reached within $N+1$ rounds of communication, where N is the maximum number of processes that can be Unreliable. In each round r, where $r \leq N$, every Reliable process j that has not yet reached a decision of $true$ checks whether g and at least $r-1$ other processes have reached a decision of $true$. If the check is successful, j reaches a decision of $true$. If j does not reach a decision of $true$ in the first N rounds, it reaches a decision of $false$ in round $N+1$.

Let $d^r.k$ be a boolean value denoting process k's tentative decision up to round r, $c^r.k.l$ be a boolean value that is $true$ iff in round r process k knows that process l has reached a decision of $true$, and $b.k$ be a boolean value that is true iff k is Reliable. Note that since we assume authenticated communication, an Unreliable k cannot for Reliable l set $c^r.k.l$ to $true$ unless $d^{r-1}.l$ is $true$.

Let $c^r.j.* = (\underline{sum}\, k : c^r.j.k : 1)$.

The byzantine agreement algorithm is described formally in the following program, along with the set of faults it tolerates.

program *Byzantine*

constant $N : integer;$
$X : set \text{ of } ID;$
$g : X;$

parameter $j, k, l : X;$
$q : 0..N+1;$

var $r : 0..N+1;$
$b.j : boolean;$
$d^q.j : boolean;$
$c^q.j.k : boolean;$

begin

$$
\begin{array}{lll}
r<N & \rightarrow & r := r+1 \\
 & & ;\ \langle \| j,k : \\
 & & \quad\ \ true \quad \rightarrow \quad c^r.j.k := d^{r-1}.k \vee (\exists l : c^{r-1}.l.k) \\
 & & \quad [\!]\ \neg b.j \wedge b.k \quad \rightarrow \quad c^r.j.k := false \\
 & & \quad [\!]\ \neg b.j \wedge \neg b.k \quad \rightarrow \quad c^r.j.k := ? \\
 & & \ \rangle \\
 & & ;\ \langle \| j : \\
 & & \quad\ \ true \quad \rightarrow \quad d^r.j := d^{r-1}.j \vee (c^r.j.* \geq r \ \wedge \ c^r.j.g) \\
 & & \quad [\!]\ \neg b.j \quad \rightarrow \quad d^r.j := ? \\
 & & \ \rangle
\end{array}
$$

end

---

faults F

$$\{(\underline{sum}\ k : \neg b.k : 1) < N \ \wedge \ b.j \quad \rightarrow \quad b.j := \ false\}$$

We show that program *Byzantine* is F-tolerant for S, where

$$
\begin{array}{rl}
S \ = & (sum\, j : \neg b.j : 1) \leq N \\
\wedge & (\forall j, k, q : \\
 & \qquad (b.j \quad \Rightarrow \quad (j = g \Rightarrow d^0.j = B) \ \wedge \ (j \neq g \Rightarrow \neg d^0.j) \ \wedge \ \neg c^0.j.k) \\
 & \quad \wedge \ \ (b.j \wedge 0 < q \leq r \quad \Rightarrow \quad d^q.j \equiv (d^{q-1}.j \vee (c^q.j.* \geq q \wedge c^q.j.g))) \\
 & \quad \wedge \ \ (b.j \wedge 0 < q \leq r \quad \Rightarrow \quad c^q.k.j \Rightarrow d^q.j) \\
 & \quad \wedge \ \ (b.j \wedge b.k \wedge \neg d^{r-1}.j \wedge 0 < q \leq r \quad \Rightarrow \quad c^q.j.k \equiv (d^{q-1}.k \vee (\exists l : c^{q-1}.l.k))) \ )
\end{array}
$$

Informally, S states that the domain of execution of program *Byzantine* satisfies the following four conditions. (i) The number of Unreliable processes is at most N. (ii) Before the first round, the tentative decision of each Reliable process j is *false*, and each $c.j$ item is *false*. (iii) In each round q, the tentative decision of each Reliable process j is set to *true* iff its previous tentative decision is true or $(c^q.j.* \geq q \ \wedge \ c^q.j.g)$ holds, and $c^q.k.j$ of each other process k is set to *true* only if $d^q.j$ is *true*. (iv) In each round q, for all Reliable processes j and k, if the current tentative decision of j is $false$ then $c^q.j.k$ is *true* iff $d^{q-1}.k \ \vee \ (\exists l : c^{q-1}.l.k)$ is *true*.

It is straightforward to show each computation of program *Byzantine* that starts at a state where S holds satisfies the byzantine agreement specification. (Details appear in [2].)

Proof

To show that program *Byzantine* is F-tolerant for S, we are required to exhibit a state predicate T that satisfies the three conditions in Definition 3. In this case, we let T to be S itself. It remains to show that S is closed in *Byzantine* as well as in F.

S is closed in Byzantine :
Upon execution of program actions,

- the first conjunct of S is trivially preserved since program actions do not update any b value,
- the first clause of the second conjunct is preserved since program actions do not update any d^0 or c^0 value,
- the second clause of the second conjunct is preserved since $d^r.j$ is set to $d^{r-1}.j \vee (c^r.j.* \geq r \ \wedge \ c^r.j.g)$,
- the third clause of the second conjunct is preserved since if $c^r.k.j$ is set to $true$, then $d^{r-1}.j$ holds and thus $d^r.j$ is set to $true$, and
- the last clause of the second conjunct is preserved, since $c^q.j.k$ is set to $d^{q-1}.k \vee (\exists l : c^{q-1}.l.k)$. □

S is closed in F :
Only the first conjunct in S names the b variables and the first conjunct is preserved upon execution of an action in F; hence, S is closed in F. □

Since the predicate T is S, the strongest solution Ts is S and, hence, *Byzantine* is masking fault-tolerant. Also, it is straightforward to show that *true* does not converge to S and, hence, that *Byzantine* is local stabilizing fault-tolerant.

Remarks

Observe that in each round r each Reliable process updates its c^r and d^r variables based only on the variables c^{r-1} and d^{r-1}. Hence, in implementing *Byzantine*, it is not necessary that each Reliable process store c^r and d^r for all r. Instead, if the state of each Reliable process is broadcast after every round, then each Reliable process needs to store only one c and one d variable.

A further optimization is made possible by the observation that once a Reliable processes j sets $d.j$ to $true$ and broadcasts its state, then in the subsequent rounds $d.j$ and each $c.k.j$ remain true. Hence, j no longer needs to participate in the computation.

4 Verifying Fault-Intolerance

In this section, we illustrate how fault-intolerance can be formally verified using our definition.

Let p be a program, S be the intended domain of execution of p, and F be a set of actions. To verify that p is not F-tolerant for S, we are obliged to show that for each state predicate T one or more of the following conditions hold.

1. T does not hold at every state where S holds,
2. T is not closed under execution of actions in F, or
3. T does not converge to S in p.

One way of meeting the above obligation is to exhibit three "witnesses":

- a state b where S holds,
- a state c reachable from b by executing actions in F, and
- a computation of p that starts at c and has no state where S holds.

To see that this method of witnesses meets the above obligation, note that for each T either conditions 1 or 2 hold or (conditions 1 and 2 are false) and, since T holds at the witness state c and the witness computation starts at c, condition 3 holds.

This method of witnesses can be simplified when verifying special kinds of fault-tolerance such as masking or global stabilizing fault-tolerance. Observe that for verifying that a program is not masking fault-tolerant, it suffices to exhibit the witness states b and c, and to show that S does not hold at c. Likewise, for verifying that a program is not stabilizing fault-tolerant, it suffices to exhibit a witness computation that has no suffix where S holds.

4.1 Example: A Delay-Insensitive Circuit

In this example, we consider circuit timing faults that are caused by delays in signal propagation. We first verify that a delay-insensitive circuit, the Muller C-element, tolerates timing faults in the arrival of its input signals [34]. We then exhibit an implementation of the C-element that uses a 3-input majority function and verify the well-known fact that the implementation is masking fault-tolerant for one type of timing fault but not masking fault-tolerant for another type.

Specification [32]

A C-element with boolean inputs x and y and a boolean output z is specified as follows: (i) Input x (respectively, y) changes only if $x \equiv z$ (respectively, $y \equiv z$) holds ; (ii) Output z becomes $true$ only if $x \wedge y$ holds, and becomes $false$ only if $\neg x \wedge \neg y$ holds ; (iii) Starting from a state where $x \equiv y$ holds, eventually a state is reached where z is set to the same value that both x and y have.

Ideally, both x and y change simultaneously. Faults may delay changing either x or y.

Program

Changing both inputs simultaneously is represented by the program action

$$x \equiv z \ \wedge \ y \equiv z \quad \rightarrow \quad x, y := \neg x, \neg y$$

If a delay occurs in the arrival of an input, then one input is changed after the other is. Changing x late is represented by the program action

$$x \equiv z \ \wedge \ y \not\equiv z \quad \rightarrow \quad x := \neg x$$

Similarly, changing y late is represented by the program action

$$x \not\equiv z \ \wedge \ y \equiv z \quad \rightarrow \quad y := \neg y$$

Lastly, if both inputs have arrived, the output can be changed. Changing the output is represented by the action

$$x \not\equiv z \ \wedge \ y \not\equiv z \quad \rightarrow \quad z := \neg z$$

The C-element is described formally in the following program, along with the set of faults it tolerates.

---

program *C-element*
var x, y, z : **boolean** ;
begin

$$
\begin{array}{llcl}
 & x \equiv z \ \wedge \ y \equiv z & \rightarrow & x, y := \neg x, \neg y \\
[\!] & x \equiv z \ \wedge \ y \not\equiv z & \rightarrow & x := \neg x \\
[\!] & x \not\equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y \\
[\!] & x \not\equiv z \ \wedge \ y \not\equiv z & \rightarrow & z := \neg z
\end{array}
$$

end

---

faults F

$$
\begin{array}{lcl}
\{ \ x \equiv z \ \wedge \ y \equiv z & \rightarrow & x := \neg x \,, \\
\ \ \ x \equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y \ \}
\end{array}
$$

We show that program *C-element* is F-tolerant for S, where S is *true*. It is straightforward to show that program *C-element* satisfies its specification for S. (We observe: First, program *C-element* satisfies the specification properties (i) and (ii) at every state. Second, every computation of *C-element*, upon starting from any state, eventually reaches a state where z is set to the value that both x and y have; thus, *C-element* also satisfies property (iii).)

Since every state is legal, the closure and convergence conditions are trivially met and, hence, program *C-element* is F-tolerant for S. In particular, it is both global stabilizing and masking fault-tolerant.

Implementation

Consider a majority circuit with three boolean inputs x, y, and u and one boolean output v. To implement the C-element using this majority circuit, it suffices to connect v to z and feedback v to u [34]. This corresponds to replacing the last action of program *C-element* with the following two actions

$$
\begin{array}{llcl}
 & v \not\equiv majority\,(u, x, y) & \rightarrow & v := majority\,(u, x, y) \\
[\!] & z \not\equiv v \ \vee \ u \not\equiv v & \rightarrow & z, u := v, v
\end{array}
$$

thereby yielding the following program:

program *C-maj-element*
var u, v, x, y, z : **boolean** ;
begin

$$
\begin{array}{llll}
 & x \equiv z \ \wedge \ y \equiv z & \rightarrow & x, y := \neg x, \neg y \\
[\!] & x \equiv z \ \wedge \ y \not\equiv z & \rightarrow & x := \neg x \\
[\!] & x \not\equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y \\
[\!] & v \not\equiv majority\,(u, x, y) & \rightarrow & v := majority\,(u, x, y) \\
[\!] & z \not\equiv v \ \vee \ u \not\equiv v & \rightarrow & z, u := v, v
\end{array}
$$

end

Faults

Program *C-maj-element* can tolerate delays in the signal from v to z, but cannot tolerate delays in the signal from v to u. To verify this fact, we consider two classes of fault actions: in $F1$, delays in the signal from v to z are allowed, thus the signal from v can change u early; in $F2$, delays in the signal from v to u are allowed, thus the signal from v can change z early. That is,

$$
\begin{array}{llll}
F1 = & \{x \equiv z \ \wedge \ y \equiv z & \rightarrow & x := \neg x\,, \\
 & \ \ x \equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y\,, \\
 & \ \ u \not\equiv v & \rightarrow & u := v \ \ \}
\end{array}
$$

and

$$
\begin{array}{llll}
F2 = & \{x \equiv z \ \wedge \ y \equiv z & \rightarrow & x := \neg x\,, \\
 & \ \ x \equiv z \ \wedge \ y \equiv z & \rightarrow & y := \neg y\,, \\
 & \ \ z \not\equiv v & \rightarrow & z := v \ \ \}
\end{array}
$$

We show that *C-maj-element* is masking $F1$-tolerant for a specific set of states, but is not masking $F2$-tolerant for any plausible set of states.

Proof

• Let $S \ = \ ((z \not\equiv v) \Rightarrow (x \not\equiv z \ \wedge \ y \not\equiv z)) \wedge ((u \not\equiv v) \Rightarrow (x \not\equiv z \ \wedge \ y \not\equiv z \ \wedge \ u \equiv z))$. We show that *C-maj-element* is masking $F1$-tolerant for S.

We observe: First, specification properties (i) and (ii) are satisfied at every state in S. Second, every computation of *C-maj-element* that starts at a state in S eventually reaches a state in S where z is set to the value that both x and y have; thus, *C-maj-*

element also satisfies property (iii). And third, S is closed under the execution of actions in *C-maj-element* and $F1$. Hence, *C-maj-element* is masking $F1$-tolerant for S.

• Let S' be any non-empty set of states that is closed in *C-maj-element* and each of whose states satisfy specification properties (i), (ii) and (iii). We show, using the method of witnesses introduced at the beginning of this section, that *C-maj-element* is not masking $F2$-tolerant for S'. In particular, we exhibit a state b that satisfies S' and a state c that does not satisfy S' and that is reachable from b by executing actions in $F2$.

We observe: Since (iii) is satisfied at every state in S', there exists a state d in S' where $x \equiv z \ \wedge \ y \equiv z$ holds. Let e be the state reached from d by executing the first action, then the fourth action, and then the fifth action of *C-maj-element.* In e, the variables u, v, x, y, and z all have the same value. Let b be the state reached from e by executing the first action and the fourth action of *C-maj-element* starting from e. Since S' is closed under execution of actions in *C-maj-element*, it follows that b satisfies S'.

Let c be the state reached from b by executing the third action and then the first action of $F2$. In c, $u \equiv x \ \wedge \ x \not\equiv v \ \wedge \ v \equiv y \ \wedge \ y \equiv z$ holds. Now, if the fourth action and then the fifth action of *C-maj-element* are executed starting from c, (ii) is violated. Since each state in S' satisfies (ii) and S' is closed under execution of actions in *C-maj-element*, it follows that c does not satisfy S'. Hence, *C-maj-element* is not masking $F2$-tolerant for S'.

5 Proving Impossibility of Fault-Tolerance

In this section, we illustrate how our definition can be used to prove that for a given specification and a given class of faults there is no program that both satisfies that specification and tolerates that class of faults.

In keeping with the method of witnesses presented in the previous section, we observe that to prove that there is no program that both satisfies some specification SP and tolerates F, it suffices to exhibit three witnesses:

- a state b that is in the domain of execution of all programs satisfying SP,
- a state c that is reachable from b by executing actions in F, and
- a computation of every program satisfying SP that starts at c and has no suffix satisfying SP.

Several results in the literature on impossibility of fault-tolerance [28] can be proven using this method, including the well-known impossibility of distributed consensus with one faulty process [20]. Some of these impossibility results involve special kinds of fault-tolerance such as masking or global stabilizing fault-tolerance. Observe that for proving impossibility of masking fault-tolerance, it suffices to exhibit the states b and c, and to show that c does not satisfy SP . Likewise, for proving impossibility of global stabilizing fault-tolerance, it suffices to exhibit a witness computation that has no suffix satisfying SP .

5.1 Example: Mutual Exclusion

We prove a new impossibility result using the method outlined above. Our impossibility result concerns programs for mutual exclusion which exhibit the following fault-tolerance property: upon starting from an illegal state, their execution necessarily reaches a state where no further execution is possible. Such programs for mutual exclusion are desirable because their "bad behavior" does not persist indefinitely.

More formally, consider a program p whose intended domain of execution is S. We say that p *halts on failure* iff the following two conditions hold.

- p has global stabilizing fault-tolerance with respect to $S \vee TR$, and
- $\neg S$ converges to TR,

where TR is the state predicate denoting all states of p where no further execution is possible.

Consider, further, programs whose variables can be partitioned so that variables in each partition are written by actions in one process only. We say: an action in process j is a *read* action iff it reads a variable that is written in some action of a process other than j; an action in process j is a *write* action iff it writes a variable of j that is read in some action of a process other than j. Program p is *read–write* iff none of its process actions is both a read and write action.

Theorem: No read–write program for mutual exclusion halts on failure.

Proof: Let p be an arbitrary read–write program for mutual exclusion, and let S be the intended domain of execution of p. That is, S is a closed state predicate of p such that all computations of p starting in S satisfy the following two properties [17].

- *Safety* : at most one process is "privileged" at each state in the computation, and

- *Deadlock-Freedom* : if the computation starts at a state where some process has requested the privilege, then there exists a subsequent state in the computation where some process that previously requested the privilege is privileged.

Our obligation is to show that p does not halt on failure. We meet this obligation by exhibiting a state transition from a state e where $\neg S$ holds to a state f where S holds. Such a state transition violates the second condition in the definition of p halts on failure, since there is no state where both S and TR hold.

Since processes of p communicate only via variables, no process in p can yield the privilege without executing some write action. (Else, deadlock freedom cannot be satisfied.) Also, notice that guards of write actions in a process of a read-write program can only access variables of that process. Hence, based on the guards of write actions that are involved in yielding the privilege, there exists for each process j a state predicate $LC.j$ over the variables of j for which at each state in S, if $LC.j$ holds then j is privileged.

Consider an infinite computation that starts at a state where some process k is privileged and some process other than k has requested the privilege. By deadlock-freedom, there exists a state transition from a state d to a state f in the computation by which k yields its privilege. Consider, further, that k performs no actions after yielding the privilege for the first time.

We claim that f results from executing a write action of k. For if f results from executing a non-write action of k, then if that action is significantly delayed from executing, it is possible for the other processes to execute the same sequence of actions that they executed after state f in the given computation and thereby violate safety.

State e can now be constructed as follows. In e, let the values of k's variables be the same as in d, and the values of the variables of other processes be the same as in f. Since $LC.k$ holds at d and since $LC.k$ depends only on k's variables, our construction ensures that $LC.k$ holds at e. Also, our construction ensures that k is not privileged at e. It follows that S does hold at e (recall that at each state in S, if $LC.k$ holds then k is privileged). Finally, we observe that the write action that updated d to yield f in the chosen computation can be executed in e to yield f. $\square$

6 Designing Fault-Tolerance

In this section, we illustrate that our definition can be used to design programs to be fault-tolerant.

Let us begin by observing that according to our definition, fault-tolerant programs meet the following two requirements: (a) their domain of execution S is closed under program execution, and (b) whenever faults perturb program execution from a state where S holds to a state where $T \wedge \neg S$ holds, subsequent program execution reaches a state where S holds.

Requirements (a) and (b) suggest that fault-tolerant programs can ideally be designed by separately designing two classes of actions: "closure" actions and "convergence" actions. Closure actions are executed only in states where the system invariant S holds, and upon execution yield states where S continues to hold. Convergence actions are executed only in states where S does not hold (but T does), and upon execution yield states where S holds.

The above classification of actions is, however, based on the assumption that it is feasible to design actions that can independently check whether S holds or can instantaneously satisfy S upon execution. This assumption is sometimes inappropriate: such actions can have large "atomicity", and thus be unsuitable for certain applications. Therefore, we relax the restriction on closure and convergence actions as follows.

- Closure actions may execute in states where S does not hold provided their execution does not prevent the convergence actions from yielding states where S holds.
- Execution of convergence actions need not establish S in one step, but in some finite number of steps.

One approach to designing such closure and convergence actions is to first characterize S in terms of a finite set of constraints, each of which can be individually established by some convergence action. Then, show that all convergence actions when executed together eventually satisfy all of the constraints (and hence also satisfy S). Finally, design closure actions that preserve each of the constraints in S. The net result is that each computation of the closure and convergence actions eventually reaches a state where S holds. For more details on this approach, we refer the reader to [5].

6.1 Example: Diffusing Computations

Consider a finite, rooted tree. It is required to design a stabilizing 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.

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. (i) j and $P.j$ have both started participating , (ii) j and $P.j$ have both completed participating, (iii) j has not started participating whereas $P.j$ has , or (iv) j has completed participating whereas $P.j$ has not. That is,
$S \;\equiv\; (\forall j :: R.j)$, where
$R.j \;\equiv\; (c.j = c.(P.j) \;\wedge\; sn.j \equiv sn.(P.j)) \vee (c.j = green \;\wedge\; c.(P.j) = red)$.

Let us consider each $R.j$ as a separate constraint in S and design the following convergence action to reestablish $R.j$ if it is violated:

$$\neg R.j \qquad\qquad \rightarrow \quad \text{“update } c.j \text{ and } sn.j \text{ to establish } R.j\text{”}$$

Observe that since the nodes are organized in a tree, every computation of the convergence actions eventually reaches a state where S holds. Observe also that there is more than one way to update $c.j$ and $sn.j$ so as to establish $R.j$. For example, the statement "$c.j, sn.j := c.(P.j), sn.(P.j)$" could be used or the statement "if $c.(P.j) = red$ then $c.j := green$ else $c.j, sn.j := green, sn.(P.j)$" could be used. Before we commit to any of these choices, let us design closure actions that preserve each $R.j$ constraint.

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

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

For propagating a diffusing computation from $p.j$ to j, we consider the closure action

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

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

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

Each of these closure actions upon execution preserve each of the $R.j$ constraints. It follows that the closure actions do not prevent the convergence actions from satisfying their constraints. Also, as noted above, every computation of the convergence actions eventually reaches a state where S holds. Hence, every computation of the closure and convergence actions eventually reaches a state from where S continues to hold.

Lastly, we observe that the statement of the propagation closure action of node j is one way of establishing $R.j$; hence, this action can be combined with the convergence action to yield the action

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

Hence, our design yields the following program [4].

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

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

$[\!]$

$$\neg R.j \;\vee$$
$$c.j = green \;\wedge\; c.(P.j) = red \;\wedge\; sn.j \not\equiv sn.(P.j) \quad \to \quad c.j\,,\, sn.j := c.(P.j)\,,\, sn.(P.j)$$

$[\!]$

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

end

7 Discussion

Any broad-based methodology such as ours raises several questions. Below, we answer some of the questions that our methodology has raised and discuss the rationale for some of the design decisions that we made in the course of this work.

While our definition of fault-tolerance specifies that all executions of a fault-tolerant program eventually reach a legal state, it does not specify how quickly the executions reach a legal state. Is our definition therefore too weak to be useful?

In defining fault-tolerance, we have deliberately chosen to separate the concerns of correctness and efficiency. To this end, our definition specifies correctness —viz, that convergence to legal states occurs in finite time— but does not specify efficiency —viz, the rate at which convergence to legal states occurs.

Nonetheless, the rate of convergence can be deduced from the proof of convergence. For example, letting a round denote a minimal sequence of steps where each process executes a step, and observing that the total number of items in *cs* and *cr* cannot exceed W, we can deduce from our proof of T converges to S in *Sliding-window* that, starting from a state where T holds, a state where S holds is reached within $3 \times W$ rounds.

Is it necessary that execution of program actions be fair?

The programs presented in this paper are correct even if the execution of program actions is not fair. More specifically, the programs are correct under the assumption of minimal progress; i.e., if there exists an enabled action, then some enabled action is executed.

We have nonetheless assumed fairness for two reasons. First, some useful programs require fairness to satisfy our definition of fault-tolerance. And second, proofs of convergence are sometimes simplified by assuming fairness, as is the case for our proof of T converges to S in *Sliding-window* (see Section 3.2).

Since faults actions can only perturb program state, how can we capture permanent faults? intermittent faults? faults some number of which can be tolerated, but more cannot?

Consider, for example, our discussion of the Byzantine Agreement problem in Section 3.3. In that discussion, executing a fault action causes a process to permanently change its mode of operation from *Reliable* to *Unreliable*. Thus, even though the fault actions by themselves only cause state perturbations, the effect of those state perturbations on the behavior of processes is permanent. (A similar argument holds for intermittent faults.)

Furthermore, in the same discussion, we show that program *Byzantine* can tolerate up to N faults —but no more— by restricting the guards of the fault actions so that the fault actions can execute at most N times.

Is our definition of fault-tolerance applicable to probabilistic programs?

Yes, provided we replace the convergence requirement with a probabilistic convergence requirement; i.e., a requirement which ensures that all program executions upon starting from a perturbed state eventually reach a legal state with probability one.

How can we reason about the fault-tolerance of program interfaces?

A program interface specifies the program behavior that is observable by some environment. This specification consists of a set of program variables and a set of constraints on how these variables may be updated [38].

In our approach, reasoning about interfaces is simple: Associated with each interface of a program p is some state predicate R that is closed under program execution. An interface is fault-tolerant with respect to some set of fault actions F iff p is F-tolerant for R.

Since only some of the program variables may be observed by the environment, it is often the case that the state predicate R (corresponding to the interface) is weaker than the state predicate S (corresponding to the intended domain of the execution). Thus, it is often the case that while p is not masking fault-tolerant with respect to S, p offers an interface R that is masking fault-tolerant.

8 Conclusions

In this paper, we have given a formal definition of what it means for a system to be fault-tolerant. The definition consists of a safety requirement, closure, and a progress requirement, convergence. It is both general (in that it expresses the fault-tolerance properties of digital and computing systems) and uniform (in that it does not depend on the type of fault considered).

In addition, we have developed a formal framework for reasoning about fault-tolerant systems. The framework comprises methods for specifying, classifying, verifying and designing system fault-tolerance. Due to its formal nature, the framework enables reasoning that is independent of technology, architecture, and application considerations.

In future work, we plan to further develop the framework along the following lines: (i) To illustrate how to augment a program to make it fault-tolerant; (ii) To illustrate how to implement a program while preserving its fault-tolerance; (iii) To develop methods for reasoning about the fault-tolerance of real-time programs; and (iv) To replace the nondeterministic interleaving semantics considered here with more general program semantics.

Acknowledgements
It is a pleasure to thank the anonymous referees for their suggestions.

References

[1] T. Anderson and P. Lee, "Fault tolerance terminology proposals", *Proceedings of FTCS-12*, pp. 29-33, 1982.

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

[3] A. Arora and M. Gouda, "Closure and convergence: A formulation of fault-tolerant computing", preliminary version in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pp. 396-403, 1992.

[4] A. Arora and M. Gouda, "Distributed reset", revised for *IEEE Transactions on Computers* ; extended abstract in *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science 472*, Springer-Verlag, pp. 316-331, 1990.

[5] A. Arora, M. Gouda, and G. Varghese, "Distributed constraint satisfaction", submitted for publication.

[6] A. Avizienis, "The four-universe information system model for the study of fault tolerance", *Proceedings of 12th International Symposium on Fault-Tolerant Computing*, pp. 6-13, 1982.

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

[8] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Chapter 7, Addison-Wesley, 1987.

[9] G. Brown, and Y. Afek, "Self-stabilization of the Alternating-Bit protocol", *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pp. 80-83, 1989.

[10] M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

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

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

[13] F. Cristian, "Understanding fault-tolerant distributed systems", *Communications of the ACM*, 34(2), pp. 56-78, 1991.

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

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

[16] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[17] E. Dijkstra, "Solution of a problem in concurrent programming control", *Communications of the ACM*, 17(11), pp. 569, 1965.

[18] E. Dijkstra and C. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

[19] P. Ezhilchelvan and S. Shrivastava, "A characterization of faults in systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

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

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

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

[23] B. Johnson, *The Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.

[24] S. Katz and K. Perry, "Self-stabilizing extensions for message-passing systems", *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pp. 91-101, 1990.

[25] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency", invited talk, *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pp. 1-11, 1984.

[26] B. Lampson and H. Sturgis, "Crash recovery in a distributed storage system", *Xerox Park Tech. Report*, Xerox Palo Alto Research Center, 1979.

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

[28] N. Lynch, "A hundred impossibility proofs for distributed computing" invited talk, *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 1-29, 1989.

[29] A. Mili, *An Introduction to Program Fault-Tolerance*, Prentice-Hall, 1990.

[30] C. Mohan, R. Strong, and S. Finkelstein, "Methods for distributed transaction commit and recovery using byzantine agreement within clusters of processes", *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pp. 29-43, 1983.

[31] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, Princeton University Press, pp. 43-98, 1956.

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

[33] M. Schneider, "Self-Stabilization", *ACM Computing Surveys*, 25(1), pp. 45-67, 1993.

[34] C. Seitz, "System timing", in *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[35] D. Siewiorek, "Architecture of fault-tolerant computers", in *Fault-Tolerant Computing: Volume II*, Prentice-Hall, 1986.

[36] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system", *IEEE Transactions on Software Engg.*, pp. 219-228, 1983.

[37] T. Srikanth and S. Toeug, "Simulating authenticated broadcast to derive simple fault tolerant algorithms", *Distributed Computing*, 2(2), pp. 80-94, 1987.

[38] B. Randell, "System structure for software fault tolerance", *IEEE Transactions on Software Engg.*, pp. 220-232, 1975.

[39] A. Tanenbaum, *Computer Networks*, Prentice-Hall, 1981.

[40] I.-L. Yen, F. Bastani, and E. Leiss, "An inherently fault-tolerant sorting algorithm", *Proceedings of the 5th International Parallel Processing Symposium*, pp. 37-42, 1991.

[41] Y. Zhao and F. Bastani, "A self-stabilizing algorithm for byzantine agreement", *University of Houston Tech. Rep. UH-CS-87-6*, 1987.