CIS 6333: Notes on Faults and Fault-tolerances

Recall that, in the absence of faults, a program satisfies its safety and liveness specification. We prove this satisfaction by exhibiting an invariant predicate such that, in the absence of faults, the program is always at a state where the invariant predicate is true.

Faults. The faults that a distributed/network program is subject to may be categorized in a variety of ways:

– *Type:* e.g., the faults are stuck-at, fail-stop, crash, omission, timing, performance, or Byzantine.

– *Duration:* e.g., the faults are permanent, intermittent, or transient.

– *Observability:* e.g., the faults are detectable or not.

– *Repair:* e.g. the faults are correctable or not.

To reason about faults in a simple and uniform manner, we adopt the following thesis:

Faults are systematically represented by actions whose execution perturbs the program state.

Definition (*Fault-class*). A fault-class for a program p is a set of actions over the variables of p. □

Consider, for example, a fault that corrupts the state of a wire. The wire itself is represented by the following program action over two bit variables in and out:

$$out \neq in \;\; \rightarrow \;\; out := in \; .$$

The fault that corrupts the state of the wire is represented by the fault action:

$$out \neq in \;\; \rightarrow \;\; out :=? \; ,$$

where ? denotes a nondeterministically chosen binary-value.

For this representation to capture all of the categories mentioned above sometimes requires the use of auxiliary state. For example, consider the fault by which the wire is stuck-at-low-voltage. In this case, the correct behavior of the wire is represented by using an auxiliary boolean variable *broken* and the program action:

$$out \neq in \; \wedge \; \neg broken \;\; \rightarrow \;\; out := in \; .$$

If a fault occurs, the incorrect behavior of the wire is represented by the program action that sets out to 0 provided that the state of the wire is *broken*:

$$broken \;\; \rightarrow \;\; out := 0 \; .$$

The stuck-at-low-voltage fault is represented by the fault action:

$$\neg broken \;\; \rightarrow \;\; broken := true \; .$$

Continuing along these lines, consider process crashes. The crash of a process is represented by introducing an auxiliary variable up for that process, as follows. Each action of that process is to be executed only if up is true. The crash itself is modeled as the occurrence of a fault that corrupts up, by setting it to false.

Similarly, the Byzantine behavior of a process can be captured by introducing an auxiliary variable *good*, as follows: If the variable *good* is true, then the process executes its normal actions. When a fault action corrupts *good* to false, the process executes actions whose behavior is nondeterministic.

Tolerances. We are now ready to define what it means for a program p with an invariant S to tolerate a fault-class F.

Definition (*Fault-span*). Let S be an invariant of a program p and F be a fault-class.
T is an F-span of p from S

iff

$S \Rightarrow T$,
T is closed in p, and
each action of F preserves T. □

Definition(F-*tolerant for* $SPEC$ *from* S). p is F-tolerant for $SPEC$ from S iff there exists a state predicate T that satisfies the following three conditions:

- At any state where S is true, T is also true. (In other words, $S \Rightarrow T$.)
- Starting from any state where T is true, if any action in p or F is executed, the resulting state is also one where T is true. (In other words, T is closed in p and T is closed in F.)
- Starting from any state where T is true, every computation of p alone eventually reaches a state where S is true. (In other words, T leads to S in p.) □

This definition may be understood as follows. The state predicate T is an F-span of p from S— a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of faults in F. If faults in F continue to occur, the state of p remains within this boundary. When faults in F stop occurring, p converges from this boundary to the stricter boundary in the state space where the invariant S is true.

It is important to note that there may be multiple such state predicates T from which p meets the above three requirements. Each of these multiple T state predicates captures a (potentially different) type of fault-tolerance of p.

Types of Tolerances. We now proceed to classify three types of fault-tolerances that a program can exhibit, namely *masking, nonmasking*, and *fail-safe* tolerance.

1. In the presence of faults, a *masking tolerant* program always satisfies its safety specification, and the execution of p after execution of actions in F yields a computation that is in both the safety and liveness specification of p, i.e., the computation is in the problem specification of p.

Definition (*masking tolerant*). p is *masking tolerant* to F for $SPEC$ from S iff p is F-tolerant for $SPEC$ from S, and S is closed in F. (In other words, if a fault in F occurs in a state where S is true, p continues to be in a state where S is true.) □

We prove this tolerance by exhibiting an invariant predicate such that even in the presence of faults the program is always at a state where the invariant predicate is true.

2. *Nonmasking tolerance* is less strict than masking tolerance: in the presence of faults, the program need not satisfy its safety specification but, when faults stop occurring, the program eventually satisfies both its safety and liveness specification; i.e., the computation has a suffix that is in the problem specification.

Definition (*nonmasking tolerant*). p is *nonmasking tolerant* to F for $SPEC$ from S iff p is F-tolerant for $SPEC$ from S, and S is not closed in F. (In other words, if a fault in F occurs in a state where S

is true, p may be perturbed to a state where S is violated. However, p then recovers to a state where S is true.) □

We prove this tolerance by exhibiting an invariant predicate such that when faults stop occurring the computation eventually reaches (recovers to) a state where the invariant predicate is true. More specifically, this would involve calculating a fault-span predicate, and showing that:

T leads-to S in p

We distinguish a special case of nonmasking tolerance: p is *stabilizing tolerant* to F iff p is nonmasking tolerant to F, and *true* converges to S in p. (In other words, stabilizing tolerant programs recover from any state in the program state space to S.)

3. *Fail-safe tolerance* is also less strict than masking: in the presence of faults, the program satisfies its safety specification but, when faults stop occurring, the program need not satisfy its liveness specification; i.e., the computation is in the safety specification –but not necessarily in the liveness specification.

Definition (*fail-safe tolerant*). Let $SSPEC$ be the minimal safety specification that contains $SPEC$.

p is *fail-safe tolerant* to F for $SPEC$ from S iff there exists a state predicate R such that p is F-tolerant for $SSPEC$ from $S \vee R$, $S \vee R$ is closed in p and in F. (In other words, if a fault in F occurs in a state where S is true, p may be perturbed to a state where S or R is true. In the latter case, the subsequent execution of p yields a computation that is in $SSPEC$ but not necessarily in $SPEC$.) □

We prove this satisfaction by exhibiting an invariant predicate and a safe predicate such that when faults occur the program is always at a state where the invariant predicate is true or at least the safe predicate is true.

Examples of Types of Tolerances. Consider the critical section problem: Its safety specification is *mutual exclusion* —multiple processes cannot simultaneously be in the critical section— and its liveness specification is *freedom from deadlock* —if some process requests critical section access then eventually some process accesses its critical section.

For the critical section problem, a masking fault-tolerant solution would preserve both mutual exclusion in the presence of the faults and satisfy freedom from deadlock if only finitely many faults occurred. A nonmasking fault-tolerant solution would eventually satisfy both mutual exclusion and freedom from deadlock if only finitely many faults occurred. Observe that this is equivalent to saying that the solution would satisfy freedom from deadlock and eventually satisfy mutual exclution if only finitely many faults occurred. A failsafe fault-tolerant solution would satisfy mutual exclusion in the presence of faults, but not necessarily freedom from deadlock.

Next, we give an example in the use of double/triple modular redundancy: The problem is to assign the value of an input variable into the variable *out*. Sensors named x, y, z contain the value of the input variable. Faults may corrupt the sensor values values of at most one of the sensors.

Fault-intolerant program IR. Program IR consists of a single action that copies the value of x into *out*. The value $\perp$ of *out* denotes that *out* has not been assigned. Thus, the action of IR is as follows:

$$IR :: \quad out = \perp \quad \longrightarrow \quad out := x$$

IR satisfies the specification in the absence of one sensor corruption but not in its presence.

Fail-safe fault-tolerant program SR. To preserve safety in the presence of one corrupted sensor, we use another sensor y thus obtaining double modular redundancy:

$$SR :: \quad out = \bot \ \wedge \ x = y \quad \longrightarrow \quad out := x$$

SR does not satisfy its liveness specification in the presence of one sensor corruption.

Nonmasking fault-tolerant program NR. To restore safety in the presence of one corrupted sensor, while preserving liveness, we use yet another sensor z thus obtaining triple modular redundancy:

$$\begin{array}{lll} NR1 :: & out = \bot & \longrightarrow \quad out := x \\ NR2 :: & out = x \ \wedge \ (x \neq y \ \wedge \ x \neq z) & \longrightarrow \quad out := y \text{ or } out := z \end{array}$$

MR satisfies the livenss specification and eventually satisfies the safety specification in the presence of one sensor corruption.

Masking fault-tolerant program MR. In fact, triple modular redundancy suffices to preserve both safety and liveness in the presence of a sensor corruption:

$$\begin{array}{lll} MR1 :: & out = \bot \ \wedge \ (x = y \ \vee \ x = z) & \longrightarrow \quad out := x \\ MR2 :: & out = \bot \ \wedge \ (y = x \ \vee \ y = z) & \longrightarrow \quad out := y \\ MR3 :: & out = \bot \ \wedge \ (z = y \ \vee \ z = x) & \longrightarrow \quad out := z \end{array}$$

MR satisfies the specification in the presence of one sensor corruption.

Remarks.

"In the absence of faults" means that each computation consists of program actions only.

"In the presence of faults" means that each computation is an interleaving of program and fault actions.

"When faults stop occurring" means that the computation has only finitely many occurrences of fault actions.

A computation "eventually satisfies" a property means that the computation has a suffix that satisfies the property.

For design and engineering purposes, it is important to characterize the classes of faults that the program is subject to. This characterization involves analyzing the environment of the program — the environment includes other program with which this interacts. In some cases, exhaustively characterizing the fault classes is difficult. In such cases, one should choose some fault-class that is large enough to accommodate all possible faults. It is often for this reason that designers choose weak fault-models such as transient state failures (where the state may be perturbed arbitrarily) or Byzantine failure (where the program may behave arbitrarily).

We have made an assumption in this discussion: execution of any fault action in F always maintains the problem specification, i.e., if a prefix σ maintains a problem specification and σs is the extended prefix obtained by execution of a fault action in F (where s is a state and σs is the concatenation of σ and s), then σs also maintains the problem specification.

Fail-Safe/Masking: Atomic Commitment Protocol

Specification

Each process casts one of two votes, Yes or No, then reaches one of two decisions, Commit or Abort, such that a process reaches a Commit decision iff all processes voted Yes.
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,
- $v.j$ be the vote of j; $v.j$ is *true* iff the vote is Yes,
- $d.j$ be the decision of j; $d.j$ is *true* iff the vote is Yes,
- $up.j$ be the current status of j; $up.j$ is *true* iff j is executing.

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;
$c : X$;
var ph : array X of 0..2;
up, v, d : array X of *boolean*;
process $j : X$;
parameter $k : X$;
begin

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

end

---

faults F

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

Program *Two-phase* is F-tolerant for S, where

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

Observe that processes decide to Commit iff each process votes Yes, in all computations of program *Two-phase* that start in a state where each process is yet to vote.

Stabilizing: Minimum Spanning Tree Construction

The specification is to continually maintain a rooted minumum spanning tree, with stabilizing tolerance to faults that change the set of up processes or the adjacency relation. In the solution described below, we accommodate such changes by ensuring that the reconfiguration program performs its task irrespective of which state it starts from.

In our solution, the rooted spanning tree is represented by a "father" relation between the processes. Each $tree.i$ process maintains a variable $f.i$ whose value denotes the index of the current father of process $P.i$. Since the layer can start in any state, the initial graph of the father relation (induced by the initial values of the $f.i$ variables) may be arbitrary. In particular, the initial graph may be a forest of rooted trees or it may contain cycles.

For the case where the initial graph is a forest of rooted trees, all trees are collapsed into a single tree by giving precedence to the tree whose root has the highest index. This is achieved as follows. Each $tree.i$ process maintains a variable $root.i$ whose value denotes the index of the current root process of $P.i$. If $root.i$ is lower than $root.j$ for some adjacent process $P.j$ then $tree.i$ sets $root.i$ to $root.j$ and makes $P.j$ the father of $P.i$.

For the case where the initial graph has cycles, each cycle is detected and removed by using a bound on the length of the path from each process to its root process in the spanning tree. This is achieved as follows. Each $tree.i$ process maintains a variable $d.i$ whose value denotes the length of a shortest path from $P.i$ to $P.(root.i)$. To detect a cycle, $tree.i$ sets $d.i$ to be $d.(f.i)+1$ whenever $f.i \in N.i$ and $d.i < K$. The net effect of executing this action is that if a cycle exists then the $d.i$ value of each process $P.i$ in the cycle gets "bumped up" repeatedly. Eventually, some $d.i$ exceeds $K-1$, where K is the maximum possible number of up processes. Since the length of each path in the adjacency graph is bounded by $K-1$, the cycle is detected. To remove a cycle that it has detected, $tree.i$ makes $P.i$ its own father.

process $tree.i\ (i : 1\ ..\ K)$
var $root.i, f.i : 1\ ..\ K;$
$d.i$: **integer**;
parameter $j : 1\ ..\ K;$

begin

$(root.i < i)\ \vee$
$(f.i = i\ \wedge\ (root.i \neq i\ \vee\ d.i \neq 0))\ \vee$
$(f.i \notin (N.i \cup \{i\})\ \vee\ d.i \geq K) \longrightarrow root.i, f.i, d.i := i, i, 0$

[]

$f.i = j\ \wedge\ j \in N.i\ \wedge\ d.i < K\ \wedge$
$(root.i \neq root.j\ \vee\ d.i \neq d.j+1) \longrightarrow root.i, d.i := root.j, d.j+1$

[]

$(root.i < root.j\ \wedge\ j \in N.i\ \wedge\ d.j < K)\ \vee$
$(root.i = root.j\ \wedge\ j \in N.i\ \wedge\ d.j+1 < d.i) \longrightarrow root.i, f.i, d.i := root.j, j, d.j+1$

end

Figure 1: Process $tree.i$

This program satisfies *true* leads to G, where

$$
\begin{aligned}
G \ \equiv\ & (k = \underline{max}\,\{i \mid P.i \text{ is up}\})\ \wedge \\
& (\forall i : P.i \text{ is up} : \\
& \quad (i = k\ \Rightarrow\ (root.i = i\ \wedge\ f.i = i\ \wedge\ d.i = 0))\ \wedge \\
& \quad (i \neq k\ \Rightarrow\ (root.i = k\ \wedge\ (\exists j : j \in N.i : f.i = j \wedge d.i = d.j+1 \wedge d.j = \underline{min}\{d.j' | j' \in N.i\}))))
\end{aligned}
$$

At each state in G, for each process $P.i$, $root.i$ equals the highest index among all up processes, $f.i$ is such that some shortest path between process $P.i$ and the root process $P.(root.i)$ passes through the father process $P.(f.i)$, and $d.i$ equals the length of this path. Therefore, a rooted spanning tree exists. Also, note that each state in G is a fixed-point; i.e., once the $tree.i$ processs reach a state in G, no action in any of the $tree.i$ modules is enabled.