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:

$$out \neq in \land \neg broken \rightarrow out := in .$$

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:

$$T \text{ leads-to } S \text{ in } p$$

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 $S \lor R$, $S \lor R$ 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 exclusion if only finitely many faults occurred. A fail-safe 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 $\bot$ of $out$ denotes that $out$ has not been assigned. Thus, the action of $IR$ is as follows:

$$IR :: \quad out=\bot \quad \rightarrow \quad out := x$$

$IR$ 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:

\[
SR :: \quad \text{out} = \bot \land x = y \quad \rightarrow \quad \text{out} := x
\]

\( SR \) 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:

\[
NR1 :: \quad \text{out} = \bot \quad \rightarrow \quad \text{out} := x
\]

\[
NR2 :: \quad \text{out} = x \land (x \neq y \land x \neq z) \quad \rightarrow \quad \text{out} := y \text{ or } \text{out} := z
\]

\( MR \) satisfies the liveness 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:

\[
MR1 :: \quad \text{out} = \bot \land (x = y \lor x = z) \quad \rightarrow \quad \text{out} := x
\]

\[
MR2 :: \quad \text{out} = x \land (y = x \lor y = z) \quad \rightarrow \quad \text{out} := y
\]

\[
MR3 :: \quad \text{out} = \bot \land (z = y \lor z = x) \quad \rightarrow \quad \text{out} := z
\]

\( MR \) 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 \( \sigma \) maintains a problem specification and \( \sigma s \) is the extended prefix obtained by execution of a fault action in \( F \) (where \( s \) is a state and \( \sigma s \) is the concatenation of \( \sigma \) and \( s \)), then \( \sigma s \) also maintains the problem specification.