Detectors and Correctors: A Theory of Fault-Tolerance Components[1]

Anish Arora *Sandeep S. Kulkarni*

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA

Abstract

In this paper, we show that two types of tolerance components, namely detectors and correctors, appear in a rich class of fault-tolerant systems. This class includes systems designed using the wellknown techniques of encapsulation and refinement, as well as systems designed using extant fault-tolerance methods such as replication and the state-machine approach. Our demonstration is via a theory of detectors and correctors, which characterizes the particular role of these components in achieving various types of fault-tolerance. Based on this theory and on our experience with using these components in designs, we suggest that detectors and correctors provide a powerful basis for efficient, component-based design of fault-tolerance.

Keywords : Composition, Fault environment, Tolerance components, Tolerance design

[1] A preliminary version of this paper appeared as [6].
Email: {anish,kulkarni}@cis.ohio-state.edu ; Web: http://www.cis.ohio-state.edu/{~anish,~kulkarni };
Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ; Research supported in part by
an Ameritech Faculty Fellowship, NSA Grant MDA904-96-1-0111, NSF Grant CCR-93-08640, and
OSU Grant 221506

1 Introduction

The thesis of this paper is : *A fault-tolerant system consists of a fault-intolerant system and a set of fault-tolerance components.* We illustrate this thesis by exhibiting the two primitive components, namely detectors and correctors, that provide a basis for achieving the different types of fault-tolerance properties for a rich class of computing systems.

Intuitively, a detector is a system component that "detects" whether some state predicate is true at the system state. Well-known examples of detectors include comparators, error detection codes, consistency checkers, watchdog programs, snoopers, alarms, snapshot procedures, acceptance tests, and exception conditions.

Likewise, a corrector is a system component that detects whether some state predicate is true at the system state and that "corrects" the system state in order to truthify that state predicate whenever it is false. Well-known examples of correctors include voters, error correction codes, reset procedures, rollback recovery, rollforward recovery, constraint (re)satisfaction, exception handlers, and alternate procedures in recovery blocks.

To justify that detectors and correctors form a basis set of components for achieving fault-tolerance properties, we consider the following questions.

1. Given a fault-tolerant system, does it contain detectors and correctors components?
2. Given a fault-intolerant system, do there exist detectors and correctors components whose composition with the system yields a fault-tolerant version of the system?

In previous work [4], we have answered Question 2 affirmatively, by presenting methods for transforming a fault-intolerant system into a fault-tolerant one. Given a fault-intolerant program, these methods show how to calculate the components required for achieving fault-tolerance, how to construct them hierarchically and distributively, and how to compose them with the given fault-intolerant program. We have applied these methods in designing fault-tolerant programs for various problems such as barrier computations, repetitive Byzantine agreement, leader election, mutual exclusion, tree maintenance, distributed reset termination detection and bounded-space network management [10, 11, 5]. (See `http://www.cis.ohio-state.edu/~anish` for additional references.) These designs are distinguished from existing solutions in multiple ways: in most cases, they are the first solutions that satisfy multiple fault-tolerance properties for the respective problems, all of them are at least as efficient as the existing solutions, and they are more efficient —in terms of time-complexity and/or space-complexity— than existing solutions.

In this paper, we focus our attention on Question 1, and provide two sorts of results. Our first set of results identify a rich class of fault-tolerant systems that contain detectors and correctors. More specifically, these results provide a theory of detectors and correctors that characterizes their respective roles in achieving fault-tolerance. The main results of this theory are as follows: (i) If a program satisfies a "safety" specification, then it contains detectors. A consequence of this result is that "fail-safe" tolerant programs contain fail-safe tolerant detectors. (ii) If a program eventually satisfies both the safety and the "liveness" of a problem specification, then it contains correctors. A consequence of this result is that "nonmasking" tolerant programs

contain nonmasking correctors. (iii) "Masking" tolerant programs contain masking tolerant detectors and correctors. (Each of the terms within quotes is defined formally in the next section.)

Our second set of results pertain to extant methods for designing fault-tolerance (e.g., replication based voting, Schneider's state machine approach [14], Randell's recovery blocks [13], checkpointing and recovery, and exception handling). More specifically, they demonstrate (a) that systems designed using replication-based voting and Schneider's state machine approach contain detectors and correctors and (b) how the same systems can be alternatively designed with the direct use of detectors and correctors. We conclude that the use of detectors and correctors generalizes extant methods, and also argue that their use offers the potential for designing more efficient fault-tolerant systems and systems that tolerate multiple types of faults.

The rest of the paper is organized as follows: In Section 2, we give formal definitions of the concepts we need for formulating our results. We develop, in Section 3, the theory of detectors, and show their role in the design of fail-safe tolerance. We continue, in Section 4, with the theory of correctors, showing their role in the design of nonmasking tolerance. Then, in Section 5, we show the role of both detectors and correctors in the design of masking tolerance. Moving on to extant design methods, in Section 6, we give illustrative examples that show that programs designed using extant design methods contain detectors and correctors. Finally, we make concluding remarks in Section 7.

2 Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerances. The formalization of programs is a standard one and borrows from work by Chandy and Misra [8], that of specifications is adapted from Alpern and Schneider [2], that of faults is adapted from earlier work of the first author with Mohamed Gouda [3], and that of fault-tolerance specifications is original.

2.1 Programs

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

$$\langle \text{name} \rangle \ :: \ \langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$$

The guard of each action is a boolean expression over the program variables. The statement of each action is such that its execution atomically and instantaneously updates zero or more program variables. □

Let p, q and p' be programs.

Definition (*State*). A state of p is defined by a value for each variable of p, chosen from the predefined domain of the variable. □

Definition (*State predicate*). A state predicate of p is a boolean expression over the variables of p . □

Note that a state predicate may be characterized by the set of all states in which its boolean expression is true. We therefore use sets of states and state predicates interchangeably. Thus, conjunction, disjunction and negation of sets is the same as the conjunction, disjunction and

negation of the respective state predicates.

Definition (*Enabled*). An action of p is enabled in a state iff its guard is true in that state. □

2.1.1 Program Compositions

Definition ($[]$ *Composition*). The parallel composition of p and q, denoted as $p \,[]\, q$, is a program whose actions are the union of the actions of p and that of q .

Definition ($\wedge$ *Composition*). Let Z be a state predicate of p. The restriction of p by Z, denoted as $Z \wedge p$, is a program whose actions are of the form $Z \wedge g \longrightarrow st$, for each action $g \longrightarrow st$ of p .

Definition (; *Composition*). Let Z be a state predicate of $p \,[]\, q$. The sequential composition of p and q with respect to Z, denoted as $p \;;_Z\; q$, is $p \,[]\, (Z \wedge q)$.

Notation. We use $\wedge$ composition for actions as well, i.e., if ac is an action of the form $g \longrightarrow st$, then $Z \wedge ac$ denotes the action $Z \wedge g \longrightarrow st$. Also, in a '; composition' when the predicate Z is clear from the context, we write $p; q$ to mean $p \;;_Z\; q$.

Definition (*Computation*). A computation of p is a fair, maximal sequences of states $s_0, s_1, ...$ such that for each j, $j > 0$, s_j is obtained from state s_{j-1} by executing an action of p that is enabled in the state s_{j-1}. Fairness of the sequence means that each action in p that is continuously enabled along the states in the sequence is eventually chosen for execution. Maximality of the sequence means that if the sequence is finite then the guard of each action in p is false in the final state. □

Let S be a state predicate.

Definition (S-*computations*). The S-computations of p, denoted as $p \mid S$, is the set of computations of p that start in a state where S is true.

Definition (*Encapsulates*). p' encapsulates p iff each action in p' that updates variables of p is of the form $g \wedge g' \longrightarrow st||st'$, where $g \longrightarrow st$ is an action of p *and* st' does not update variables of p, and an action of the form $g \wedge g' \longrightarrow st||st'$, is executed only when its guard, $g \wedge g'$, is true, and to execute this action st and st' are atomically executed. (Note that st' may read variables used in st; in this case, the values of the variables in the initial state are used in the execution of st' .)

2.2 Problem Specification

Definition. A problem specification is a set of sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and x is a program state. □

Note that the state sequences in a problem specification may be finite or infinite. Following Alpern and Schneider [2], it can be shown that any problem specification is the intersection of some "safety" specification that is suffix closed and fusion closed and some "liveness" specification, defined next.

Definition (*Safety*). A safety specification is a set of state sequences that meets the following condition: for each state sequence σ not in that set, there exists a prefix α of σ, such that for all state sequences β, $\alpha\beta$ is not in that set (where $\alpha\beta$ denotes the concatenation of α and β). □

Definition (*Liveness*). A liveness specification is a set of state sequences that meets the following condition: for each finite state sequence α there exists a state sequence β such that $\alpha\beta$ is in that set. □

Defined below are some examples of problem specifications, namely, generalized pairs, closures, and converges to. For these examples, let S and R be state predicates.

Definition (*Generalized Pairs*). The generalized pair $(\{S\}, \{R\})$ is a set of state sequences, $s_0, s_1, ...$ such that for each $j, j \geq 0$, if S is true at s_j then R is true at s_{j+1}. □

Definition (*Closure*). The closure of S, $cl(S)$, is the set of all state sequences $s_0, s_1, ...$ where for each $j, j \geq 0$, if S is true at s_j then S is true at each $k, k \geq j$. □

Definition (*Converges to*). S converges to R is the set of all state sequences $s_0, s_1, ...$ in the intersection of $cl(S)$ and $cl(R)$ such that if there exists $i, i \geq 0$, for which S is true at s_i then there exists k, $k \geq i$, for which R is true at s_k. □

Note that $(\{S\}, \{S\}) = cl(S) =$ S converges to S.

Notation. We use S^* to denote a finite sequence of states where S is true in each state. Thus, $(true)^*$ denotes an arbitrary finite sequence of states. Also, if α is a finite sequence of states *and* β is a sequence of states, then $\alpha\beta$ is concatenation of α and β . And, if Γ is a set of finite sequence of states *and* Δ is a set of sequence of states, then $\Gamma\Delta = \{\ \alpha\beta :\ \alpha \in \Gamma$ and $\beta \in \Delta\ \}$.

2.2.1 Program Correctness with respect to a Problem Specification

Let $SPEC$ be a problem specification.

Definition (*Projection*). The projection of a state of p' on p (respectively $SPEC$) is a state obtained by considering only the variables of p (respectively $SPEC$).

Definition (*Projection*). The projection of a computation of p' on p (respectively $SPEC$) is a sequence of states obtained by projecting each state in that computation on p (respectively $SPEC$) .

Definition (*Refines*). p' refines p (respectively $SPEC$) from S iff the following two conditions hold:

- S is closed in p', and
- For every computation of p' that starts in a state where S is true, the projection of that computation on p (respectively $SPEC$) is a computation of p (respectively $SPEC$).

Notation. We use 'a computation of p is in $SPEC$' to mean the projection of that computation on $SPEC$ is in $SPEC$. Also, if c is a computation of p, we use the term '$c \in SPEC$' to mean that the projection of c on $SPEC$ is in $SPEC$.

Definition (*Violates*). p violates $SPEC$ from S iff it is not the case that p refines $SPEC$ from S . □

Definition (*Maintains*). Let α be a prefix of a computation of p. The prefix α maintains $SPEC$ iff there exists a sequence of states β such that $\alpha\beta \in SPEC$. □

For convenience in reasoning about programs that refine special cases of problem specifications, we introduce the following notational abbreviations.

Definition (*Generalized Hoare-triples*). $\{S\}\ p\ \{R\}$ iff p refines the generalized pair $(\{S\}, \{R\})$ from *true*. □

Definition (*Closed in* p). S is closed in p iff p refines $cl(S)$ from *true*. □

Note that it is trivially true that the state predicates *true* and *false* are closed in p.

Definition (*Converges to in* p). S converges to R in p iff p refines S converges to R from *true*. □

Informally speaking, proving the correctness of p with respect to $SPEC$ involves showing that p refines $SPEC$ from some state predicate S. (Of course, to be useful, the predicate S should not be *false*.) We call such a state predicate S an invariant of p. Invariants enable proofs of program correctness that eschew operational arguments about long (sub)sequences of states, and are thus methodologically advantageous.

Definition (*Invariant*). S is an invariant of p for $SPEC$ iff p refines $SPEC$ from S. □

One way to calculate an invariant of p is to characterize the set of states that are reachable under execution of p starting from some designated "initial" states. Experience shows, however, that for ease of proofs of program correctness one may prefer to use invariants of p that properly include such a reachable set of states. This is a key reason why we have not included initial states in the definition of programs.

Notation. Henceforth, whenever the problem specification is clear from the context, we will omit it; thus, "S is an invariant of p" abbreviates "S is an invariant of p for $SPEC$".

2.3 Faults

The faults that a program is subject to are systematically represented by actions whose execution perturbs the program state. We emphasize that such representation is possible notwithstanding the type of the faults (be they stuck-at, crash, fail-stop, omission, timing, performance, or Byzantine), the nature of the faults (be they permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults (be they detectable or undetectable).

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

Let $SPEC$ be a problem specification, T be a state predicate, S an invariant of p, and F a fault-class for p.

Definition. (*Computation in the presence of faults*). A computation of p in the presence of F is a sequence of states $s_0, s_1, ...$ that is p-fair and p-maximal such that for each $j, j > 0$, s_j is obtained from s_{j-1} by executing an action of p or an action of F that is enabled in s_{j-1}, and $|\{j : s_j$ is obtained from s_{j-1} by executing an action of $F\}|$ is finite. By p-fairness, we mean that for each action of p that is continuously enabled along the states in the sequence is eventually chosen for execution. And, by p-maximality, we mean that if the sequence is finite then

the guard of each action in p is false in the final state.

Notation. We overload $[\!]$ for combining programs and faults. More specifically, we use the notation $p \,[\!]\, F$ to mean the union of actions of p and F . However, a computation of $p \,[\!]\, F$ is only p-fair and p-maximal.

Definition (*Preserves*). An action ac preserves a state predicate T iff execution of ac in any state where T is true results in a state where T is true. □

Definition (*Fault-span*). 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. □

Thus, at each state where an invariant S of p is true, an F-span T of p from S is also true. Also, like S, T is also closed in p. Moreover, if any action in F is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where S is true, T is 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 the actions in F.

Notation. Henceforth, we will ambiguously abbreviate the phrase "each action in F preserves T" by "T is closed in F". And, whenever the program p is clear from the context, we will omit it; thus, "S is an invariant" abbreviates "S is an invariant of p" and "F is a fault-class" abbreviates "F is a fault-class for p".

2.4 Fault-Tolerance Specifications

In the absence of faults, a program should refine its problem specification. In the presence of faults, however, it may not refine its specification, it may refine some (possibly) weaker 'tolerance specification'. Below, we define some tolerance specifications that occur often in practice.

Definition (*Masking tolerance specification of* $SPEC$). The masking tolerance specification of $SPEC$ is $SPEC$.

Definition (*Fail-safe tolerance specification of* $SPEC$). The fail-safe tolerance specification of $SPEC$ is the smallest safety specification containing $SPEC$.

Definition (*Nonmasking tolerance specification of* $SPEC$). The nonmasking tolerance specification of $SPEC$ is $(true)^* SPEC$.

Using these definitions, we are now ready to define what it means for a program to tolerate a fault-class F. With the intuition that a program is F-tolerant to $SPEC$ if it refines $SPEC$ in the absence of faults *and* it refines a tolerance specification of $SPEC$ in the presence of F, we define 'F-tolerant to $SPEC$ from S' as follows:

Definition (F-*tolerant for* $SPEC$ *from* S). p is masking F-tolerant to $SPEC$ from S (respectively nonmasking F-tolerant to $SPEC$ from S or fail-safe F-tolerant to $SPEC$ from S) iff the following two conditions hold:

- p refines $SPEC$ from S, and
- there exists T such that $T \Leftarrow S$ *and* $p \,[\!]\, F$ refines the masking tolerance specification of $SPEC$ from T (respectively the nonmasking tolerance specification of $SPEC$ from T or the fail-safe tolerance specification of $SPEC$ from T) .

The type of tolerance characterizes extent to which the program refines $SPEC$ in the presence of faults. Of the three, masking is the strictest type of tolerance: computations of the program in the presence of faults are always in $SPEC$. Fail-safe is less strict than masking: computations of the program in the presence of faults are in the minimal safety specification that contains $SPEC$. Nonmasking is also less strict than masking: computations of the program in the presence of faults have a suffix in $SPEC$.

Notation. In the sequel, whenever the specification $SPEC$ and the invariant S are clear from the context, we omit them; thus, "masking F-tolerant" abbreviates "masking F-tolerant for $SPEC$ from S", and so on.

2.5 A Note on Assumptions

For the reader's convenience, we reiterate and justify the assumptions made in this paper and provide an argument for their non-restrictiveness.

Assumption 1 : Problem specifications are suffix closed and fusion closed.

Suffix closure allows nonmasking tolerance to capture the intuition that the execution of a nonmasking tolerant program has a suffix in the problem specification. Without this assumption, to achieve nonmasking tolerance, the program would have to be restored to initial states in the problem specification, and restoring the program to an initial state may not always be desirable. Suffix closure and fusion closure also simplify the presentation of detectors and correctors. More specifically, they are used to show the existence of *detection* predicates used in detectors and of *invariant* predicates used in the correctors.

This assumption is not restrictive in the following sense: Given a set of sequences L that is not suffix closed and/or fusion closed, it is possible (by adding "history" variables) to construct a set L' such that for a program p, all computations of p that start at some "initial states" are in L iff p refines L' from some state predicate. Thus, if the given specification is not suffix closed or fusion closed it is still possible to determine the detection predicates and the invariant predicates, although they may depend on the added history variables.

Assumption 2 : The number of fault occurrences in a computation is finite.

This assumption shows up in the definition of 'a computation in the presence of faults'. The motivation behind this assumption is that it is in general impossible to guarantee liveness if faults occur forever. The results from our theory are applicable if eventually faults stop for a long enough time for the program to make progress.

This assumption is not restrictive in the following sense: If a fault happens infinitely often and the liveness condition at hand can still be satisfied then we can get around this assumption by treating the fault actions as program actions.

3 Detectors and Their Role in Fail-Safe Tolerance

In this section, we introduce the first of the two tolerance components, *detectors*. Below, we define detectors formally and develop their theory. Subsequently, we present a simple memory access example to illustrate an instance of detectors. (As mentioned in the introduction,

methods for the hierarchical and distributed construction of detectors and methods for adding detectors to a fault-intolerant program are presented in [4].)

3.1 Definition

Let X and Z be state predicates. Let 'Z detects X' be the problem specification that is the set of all sequences, $s_0, s_1, ...$ satisfying the following three conditions:

- (*Safeness*) For each $i, i \geq 0$, if Z is true at s_i then X is also true at s_i. (In other words, $Z \Rightarrow X$ at s_i.)
- (*Progress*) For each $i, i \geq 0$, if X is true at s_i then there exists $k, k \geq i$, such that Z is true at s_k or X is false at s_k.
- (*Stability*) For each $i, i \geq 0$, if Z is true at s_i then Z is true at s_{i+1} or X is false at s_{i+1}. (In other words, $(\{Z\}, \{Z \vee \neg X\})$ is true.) □

Definition (*detector*). Z detects X in d from U iff d refines 'Z detects X' from U. □

A detector d is used to check whether its "detection predicate", X, is true. Since d satisfies *Progress* from U, it follows that if $U \wedge X$ is true continuously, d eventually detects this fact and truthifies Z. Since d satisfies *Safeness* from U, it follows that d never lets Z witness X incorrectly. Moreover, since d satisfies *Stability* from U, it follows that once Z is truthified, it continues to be true unless X is falsified, i.e., $\{U \wedge Z\}\ d\ \{Z \vee \neg X\}$.

Definition (*tolerant detector*). d is a fail-safe (respectively nonmasking or masking) tolerant detector for 'Z detects X' from U iff d refines the fail-safe (respectively nonmasking or masking) tolerance specification of Z detects X from U .

Remark. If the detection predicate X is closed in d, our definition of the detects relation reduces to one given by Chandy and Misra [8]. We have considered this more general definition to accommodate the case —which occurs for instance in nonmasking tolerance— where X denotes that "something bad has happened"; in this case, X is not supposed to be closed since it has to be subsequently corrected. *(End of Remark.)*

3.2 Theory of Detectors

We show (1) if a program refines a safety specification then it contains detectors, and (2) if a program is fail-safe F-tolerant then it contains fail-safe tolerant detectors.

Our proof is organized as follows: first, Lemma 3.1 shows that if two prefixes of a computation maintain a safety specification then so does their concatenation. Then, Lemma 3.2 shows that the violation of a safety specification can be detected from the current state, independent of how that state is reached. Subsequently, Theorem 3.3 shows that for each action of the program, there exists a set of states from where execution of that action maintains the given safety specification. Finally, using Theorem 3.3, Theorems 3.4 and 3.6 respectively show (1) and (2) .

Throughout this section, let p be a program, $SPEC$ be a problem specification, $SSPEC$ be the minimal safety specification that contains $SPEC$, σ be a prefix of a computation, β be a finite suffix of a computation, s and s' be states, and X be a state predicate.

Lemma 3.1

If

- σs maintains $SPEC$, and
- $s\beta$ maintains $SPEC$

then

- $\sigma s\beta$ maintains $SPEC$.

Proof.

σs maintains $SPEC \quad \wedge \quad s\beta$ maintains $SPEC$
$=$ { by definition of maintains }
$(\exists \delta : \sigma s\delta \in SPEC) \quad \wedge \quad (\exists \delta' : s\beta\delta' \in SPEC)$
$\Rightarrow$ { by fusion closure of SPEC }
$(\exists \delta' : \sigma s\beta\delta' \in SPEC)$
$=$ { by definition of maintains }
$\sigma s\beta$ maintains $SPEC$ □

Lemma 3.2

If

- σs maintains $SPEC$

then

- $\sigma ss'$ maintains $SPEC$ iff ss' maintains $SPEC$.

Proof. *If part:*

$\sigma ss'$ maintains $SPEC$
$=$ { by definition of maintains }
$(\exists \beta : \sigma ss'\beta \in SPEC)$
$\Rightarrow$ { by suffix closure of $SPEC$ }
$(\exists \beta : ss'\beta \in SPEC)$
$=$ { by definition of maintains }
ss' maintains $SPEC$

Only if part:

ss' maintains $SPEC \quad \wedge \; \sigma s$ maintains $SPEC$
$\Rightarrow$ { by Lemma 3.1 }
$\sigma ss'$ maintains $SPEC$ □

Theorem 3.3 For each action ac of p there exists a predicate such that execution of ac in a state where that predicate is true maintains $SPEC$.

Proof. Consider a prefix of a computation, say σs, that maintains $SPEC$. Execution of ac maintains $SPEC$ iff the extended prefix $\sigma ss'$, after execution of ac maintains $SPEC$. In other words, there exists a set of prefixes of computation, say $PREF$, from which execution of ac maintains $SPEC$.

From Lemma 3.2, it follows that the extended prefix $\sigma ss'$ maintains $SPEC$ iff ss' maintains $SPEC$. Thus, the execution of ac maintains $SPEC$ iff it executes in a state that is in the set $\{s : \exists \sigma : \sigma s \in PREF \}$. The predicate characterized by this set of states suffices as a witness for the theorem. □

Definition (*detection predicate*). We say that X is a *detection predicate* of action ac for $SPEC$ iff execution of ac in any state where X is true maintains $SPEC$. □

Note that the existence of detection predicates follows from Theorem 3.3, and that an action may have multiple detection predicates. Also, if sf is a detection predicate of ac for $SPEC$ *and* $X \Rightarrow sf$, then X is also a detection predicate of ac for $SPEC$. And, if $sf1$ and $sf2$ are detection predicates of ac for $SPEC$ then so is $sf1 \vee sf2$. Thus, there exists a weakest safe predicate for each action.

Using the definition of detection predicates, we are now ready to show that if a program refines a safety specification then it contains detectors. The intuition is that if program p' is designed by transforming p so as to satisfy $SSPEC$, then the transformation must have added a detector for each action of p, i.e., p' must contain a detector for each action of p . We formulate this, in Theorem 3.4, for the case where the transformation uses encapsulation and refinement.

Typically, the detector components used in p' will be smaller (in terms of actions/state transitions, etc.) than p'. However, for p' to refine $SSPEC$ the components used in p' must not interfere with each other. If a component of p' refines the detector specification and the other components in p' do not interfere with it then p' will also refine the detector specification. Therefore, in Theorem 3.4 we show that p' itself refines the specification of a detector.

Theorem 3.4 (Programs that refine a safety specification contain detectors).

If

- p' refines p from S,
- p' encapsulates p, and
- p' refines $SSPEC$ from S

then

- $(\forall ac : ac$ is an action of $p : p'$ is a detector of a detection predicate of $ac)$.

Proof. Let sf be the weakest detection predicate for ac . Since p' encapsulates p, if ac is of the form $g \longrightarrow st$, p'contains an action, say ac', of the form $g \wedge g' \longrightarrow st||st'$.

Let $Z = g \wedge g'$, and let
$X = g \wedge sf \wedge$
$(\neg\{s : s$ is a state of $p' :$ Z is false in state s, $g \wedge sf$ is true in state s, and
there exists a transition $(s0, s)$ of p' such that Z is true in state $s0$ $\}) \wedge$
$(\neg\{s : s$ is a state of $p' :$ Z is false in state s, g is true in state s,
there exists another action, say $ac1$, of p and states, say $s0, s1$ of p' such that
$(s, s0)$ is a transition of ac, $(s, s1)$ is a transition of $ac1$, and
the projection of $s0$ and $s1$ on p is same. $\})$.

Since $X \Rightarrow sf$, whenever X is true, execution of ac maintains $SSPEC$. It follows that X is a safe predicate of ac .

We now show that p' refines Z detects X from S .

By definition of Z, $Z \Rightarrow g$. Since p' refines $SSPEC$ from S, whenever ac is executed in a state where S is true, its execution is safe. Since sf is the weakest detection predicate of ac, $S \wedge Z \Rightarrow sf$. Also, Z implies the remaining two predicates in X . Thus, Safeness is satisfied.

Consider any computation, say c', of p' which starts in a state where S is true and X is true in each state in c' : By definition of X, g is true in each state in c' . Now, consider the computation, say c, obtained by projecting c' on p : Since p' refines p from S, c is a computation of p . In c, g is continuously true. Therefore, by fairness, action ac must eventually execute. Let s denote the state where action ac executes in c, and let s' denote the corresponding state in c' . Consider the action executed by p' in state s': it is either ac' or an action $ac1'$ which is based on action $ac1$ of p such that executing ac and $ac1$ have the same effect on variables of p from state s . In the former case, Z is true in state s' . And, in the latter case, either Z is true in the state s' or the fourth conjunct in X is false in the state s' . Thus, Progress is satisfied.

Starting from a state where Z is true, if p' has a transition to a state where Z is false, then in that state the third conjunct in X is false. It follows that Stability is satisfied. □

Remark. Henceforth, to show that p' contains detectors (respectively correctors), we will show that p' itself refines the corresponding detector (respectively corrector) specifications.

Observe that in Theorem 3.4 'p' refines p from S' is used only to show the Progress of the detector. It follows that if only encapsulation is used then p' continues to satisfy Safeness and Stability. Thus, we have

Lemma 3.5

If

- p' encapsulates p, and
- p' refines $SSPEC$ from S

then

- $(\forall\ ac : ac$ is an action of $p : p'$ is a fail-safe tolerant detector of a detection predicate of $ac)$.

Proof. We use the same definition of Z and X as in the proof of Theorem 3.4, and show that p' refines the fail-safe tolerance specification of 'Z detects X' from S . We leave it to the reader to verify that the proof of Safeness and Stability in Theorem 3.4 can be used, verbatim, to prove that p' satisfies Safeness and Stability. □

We now use Theorem 3.4 and Lemma 3.5 to show that if a fail-safe F-tolerant program p' is designed by using encapsulation and refinement from program p then p' contains a fail-safe tolerant detector for each action of p .

Theorem 3.6 (Fail-safe F-tolerant programs contain fail-safe tolerant detectors).

If

- p refines $SPEC$ from S,
- p' refines p from R, where $R \Rightarrow S$
- p' encapsulates p, and
- $p' [] F$ refines $SSPEC$ from T, where $T \Leftarrow R$

then

- p' is fail-safe F-tolerant for $SPEC$ from R, and
- $(\forall\ ac : ac$ is an action of $p : p'$ is a fail-safe F-tolerant detector of a detection predicate of $ac)$.

Proof. *Part 1: fail-safe F-tolerance to SPEC .* Since p' refines p from R, R is closed

in p' and for every computation of p' that starts in a state where R is true, the projection of that computation on p is a computation of p. Also, since p refines $SPEC$ from S *and* $R \Rightarrow S$, for every computation of p that starts in a state where R is true, the projection of that computation on $SPEC$ is in $SPEC$. It follows that for every computation of p' that starts in a state where R is true, the projection of that computation on $SPEC$ is in $SPEC$. Thus, p' refines $SPEC$ from R .

Since $R \Rightarrow T$ and T is closed in $p'[]F$, in the presence of F, p' is perturbed only to states where T is true. From these states, p' refines the safety specification of $SPEC$, namely $SSPEC$. It follows that p' is fail-safe F-tolerant for $SPEC$ from R .

Part 2: detector. Let sf be the weakest detection predicate for ac . Since p' encapsulates p, if ac is of the form $g \longrightarrow st$, p'contains an action, say ac', of the form $g \wedge g' \longrightarrow st||st'$.
Let $Z = g \wedge g'$, and let
$X = g \wedge sf \wedge$
($\neg\{s : s$ is a state of $p' :$ Z is false in state s, $g \wedge sf$ is true in state s, and
there exists a transition $(s0, s)$ of p' or **F** such that Z is true in state $s0$ $\}) \wedge$
($\neg\{s : s$ is a state of $p' :$ Z is false in state s, g is true in state s,
there exists another action, say $ac1$, of p and states, say $s0, s1$ of p' such that
$(s, s0)$ is a transition of ac, $(s, s1)$ is a transition of $ac1$, and
the projection of $s0$ and $s1$ on p is same. $\}$).

Since $X \Rightarrow sf$, whenever X is true, execution of ac maintains $SSPEC$. It follows that X is a safe predicate of ac .

We now show that p' is fail-safe F-tolerant for Z detects X from R and the F-span of p' is T . To this end, we first show that p' refines Z detects X from R . Then, we show that $p'[]F$ refines the fail-safe tolerance specification of Z detects X from T .

For the first part, since $R \Rightarrow T$, we observe that p' refines $SSPEC$ from R . Therefore, by Theorem 3.4, it follows that p' refines Z detects X from R .

For the second part, we need to show that a computation of $p'[]F$ satisfies Safeness and Stability. This proof is identical to the proof of Safeness and Stability in Theorem 3.4 . □

3.3 Example : Memory Access

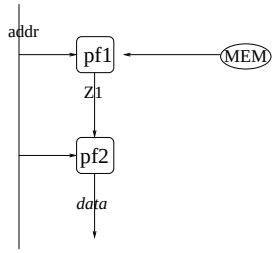
Let us consider a simple memory access program that obtains the value stored at a given address in the memory. For ease of exposition, we will allow access to only one memory location, $addr$. Thus, an intolerant program for memory access, p, is as follows (where MEM contains the set of objects of the form $\langle addr, value\rangle$, and if MEM does not contain an object of the form $\langle addr, -\rangle$, $(val|\langle addr, val\rangle \in MEM)$ returns an arbitrary value):

$p ::$ $\quad true \quad \longrightarrow \quad data := (val|\langle addr, val\rangle \in MEM)$

The fault-class we consider is a page fault whereby $addr$ and its value are initially removed from the memory. In the presence of this fault, fail-safe tolerance can be achieved by the following program, say pf .

$pf1 ::$ $\quad (\exists\ val :: \langle addr, val\rangle \in MEM) \ \wedge \ \neg Z1 \longrightarrow \quad Z1 := true$
$pf2 ::$ $\quad Z1 \wedge true \quad \longrightarrow \quad data := (val|\langle addr, val\rangle \in MEM)$

Program pf contains two actions: the first action detects whether $addr$ is in the memory. If this detection is successful, it sets $Z1$ to true. The data is accessed only when $Z1$ is true. (cf. Figure 1: the predicate $X1$ denotes that $addr$ is currently in the memory, and $U1$ denotes that the predicate $Z1$ is truthified only when the predicate $X1$ is true.)



$$X1 \equiv (\exists val :: \langle addr, val \rangle \in MEM)$$
$$U1 \equiv (Z1 \Rightarrow X1)$$

Figure 1: **Memory access**

Program pf is fail-safe 'page-fault'-tolerant in the sense that it refines the specification of the memory transfer program, $SPEC_{mem}$ in the absence of faults, and it refines the fail-safe tolerance specification of $SPEC_{mem}$ in the presence of a page fault. (Intuitively, $SPEC_{mem}$ requires that the $data$ is eventually set to the correct value, and it is never set to an incorrect value.) More specifically, if no fault occurs, i.e., a tuple $\langle addr, val \rangle$ exists in the memory, it eventually sets $data$ to be equal to val , and it never sets $data$ to any other value. In the presence of a page fault, it never sets the value of $data$ incorrectly, although, it may not assign a value to $data$. We use the theory of detectors developed in the previous subsection to show that pf is fail-safe 'page-fault'-tolerant.

Let $S := U1 \wedge X1$, $T := U1$, and $F :=$ 'page fault'. Now, observe that p refines $SPEC_{mem}$ from S, pf refines p from S, pf encapsulates p, and $pf[]F$ refines the safety specification of $SPEC_{mem}$ from T. Therefore, by Theorem 3.6, we have

pf is fail-safe 'page fault'-tolerant for $SPEC_{mem}$ from S, and
pf is a fail-safe 'page fault'-tolerant detector of a detection predicate of p .

The alert reader will note that the detection predicate of pf is $X1$ and the witness predicate of pf is $Z1$. Also, this detector is implemented by action $pf1$ in program pf .

4 Correctors and Their Role in Nonmasking Tolerance

In this section, we introduce the second of the two tolerance components, *correctors*. Below, we define correctors formally and develop their theory. Subsequently, we build upon our memory access example to illustrate an instance of correctors. (As mentioned in the introduction, methods for the hierarchical and distributed construction of correctors and methods for adding correctors to a fault-intolerant program are presented in [4].)

4.1 Definition

Let X and Z be state predicates. Let 'Z corrects X' be the problem specification that is the set of all state sequences, $s_0, s_1, ...$ satisfying the following three conditions:

- (*Convergence*) There exists $i, i \geq 0$, such that for each $j, j \geq i$, X is true at s_j, and for each $k, k \geq 0$, if X is true at s_k then X is also true at s_{k+1}.
- (*Safeness*) For each $i, i \geq 0$, if Z is true at s_i then X is also true at s_i. (In other words, $Z \Rightarrow X$.)
- (*Progress*) For each $i, i \geq 0$, if X is true at s_i then there exists $k, k \geq i$, such that Z is true at s_k or X is false at s_k.
- (*Stability*) For each $i, i \geq 0$, if Z is true at s_i then Z is true at s_{i+1} or X is false at s_{i+1}. (In other words, $(\{Z\}, \{Z \vee \neg X\})$.) □

Definition (*corrector*). Z corrects X in c from U iff c refines 'Z corrects X' from U. □

Since c satisfies *Convergence* from U, it follows that eventually c reaches a state where X is truthified and X continues to be true thereafter. Moreover, since c satisfies *Safeness* from U, it follows that a corrector never lets the predicate Z witness the correction predicate X incorrectly. Since c satisfies *Progress* from U, it follows that Z is eventually truthified. And, finally, since c satisfies *Stability* from U, it follows that Z is never falsified.

Definition (*tolerant corrector*). c is a nonmasking (respectively fail-safe or masking) tolerant corrector for 'Z corrects X' from U iff c refines the nonmasking (respectively fail-safe or masking) tolerance specification of Z corrects X from U .

Remark. If the witness predicate Z is identical to the correction predicate X, our definition of the corrects relation reduces to one given by Arora and Gouda [3]. We have considered this more general definition to accommodate the case —which occurs for instance in masking tolerance— where the witness predicate Z can be checked atomically but the correction predicate X cannot. *(End of Remark.)*

4.2 Theory of Correctors

We show (1) if a program eventually refines a specification then it contains correctors, and (2) if a program is nonmasking F-tolerant then it contains nonmasking tolerant correctors.

Throughout this section, let p be a program, α be a prefix of a computation, β be a suffix of a computation, $SPEC$ be a problem specification, and s be a state.

Let p be a program that refines $SPEC$ from S. In Theorem 4.1, we show that if p' is designed such that it eventually behaves like p and, thus, has a suffix in $SPEC$, then p' contains a corrector of an invariant predicate of p . As discussed in Section 3.2, we prove Theorem 4.1 by showing that p' itself refines the required corrector specification.

Theorem 4.1 (Programs that eventually refine a specification contain correctors).

If

- p refines $SPEC$ from S,
- p' refines p from S, and
- p' refines $(true)^*(p' \mid S)$ from T

then

- p' is a corrector of an invariant predicate of p .

Proof.
Let $X = S$, and
$Z = S \wedge \{s : s$ is a state of $p' : s$ is reached in some computation of p' starting from $T \}$.

Since p refines $SPEC$ from S, it follows that X is an invariant predicate of p for $SPEC$. Now, we show that p' refines 'Z corrects X' from T .

By definition of Z, in any state where Z is true, S is true. In other words, in any state where Z is true, X is also true. Thus, Safeness is satisfied.

Since p' refines $(true)^*(p' \mid S)$ from T, every computation of p' starting from T will reach a state where S is true. By definition of Z, Z is true in this state. Thus, Progress is satisfied.

Since p' refines p from S, it follows that S is closed in p' . Also, the second conjunct in Z is closed in p' . Thus, Z is closed in p' . Thus, Stability is satisfied.

Since p' refines $(true)^*(p' \mid S)$ from T, every computation of p' starting from T will reach a state where S is true. And, S is closed in p' . Thus, Convergence is satisfied. □

The next lemma generalizes Theorem 4.1 . In general, given a program p that refines $SPEC$ from S, p' may not behave like p from each state in S but only from a subset of S, say R . This may happen, for example, if p' contains additional variables and p' behaves like p only after the values of these additional variables are restored. Lemma 4.2 shows that in such a case, p' contains a nonmasking corrector of an invariant predicate of p . (The corrector is nonmasking in the sense that the correction predicate is preserved only after p' reaches a state where R is true.)

Lemma 4.2

If

- p refines $SPEC$ from S,
- p' refines p from R, where $R \Rightarrow S$, and
- p' refines $(true)^*(p' \mid R)$ from T

then

- p' is a nonmasking corrector of an invariant predicate of p .

Proof.

Let $X = S$, and
$Z = R$.

We show that p' refines the nonmasking tolerance specification of Z corrects X from T . In particular, we first show that a computation of p' starting from a state where T is true

eventually reaches a state where R is true. Then, we show that starting from this state p' refines the Z corrects X .

For the first part, since p' refines $(true)^*(p' \mid R)$, it follows that p' eventually reaches a state where R is true.

For the second part, we show that starting from this state, p' satisfies Safeness, Progress, Stability and Convergence. $R \Rightarrow S$ is trivially true, thus, Safeness is satisfied. In a state where R is true, Progress is satisfied. Since p' refines p from R, R is closed in p', Stability is satisfied. Finally, in a computation starting from a state where R is true, S is true at all states and, thus, Convergence is satisfied. □

We now use Theorem 4.1 and Lemma 4.2 to show that if a nonmasking F-tolerant program p' is designed from p using refinement then p' contains a nonmasking corrector for an invariant of p .

Theorem 4.3 (Nonmasking F-tolerant programs contain nonmasking tolerant correctors).

If

- p refines $SPEC$ from S,
- p' refines p from R, where $R \Rightarrow S$ and
- $p' [\!] F$ refines $(true)^*(p' \mid R)$ from T, where $T \Leftarrow R$

then

- p' is nonmasking F-tolerant for $SPEC$ from R, and
- p' is a nonmasking F-tolerant corrector of an invariant predicate of p .

Proof. *Part 1: nonmasking F-tolerance to $SPEC$.* Since p' refines p from R, R is closed in p' and for every computation of p' that starts in a state where R is true, the projection of that computation on p is a computation of p. Also, since p refines $SPEC$ from S *and* $R \Rightarrow S$, for every computation of p that starts in a state where R is true, the projection of that computation on $SPEC$ is in $SPEC$. It follows that for every computation of p' that starts in a state where R is true, the projection of that computation on $SPEC$ is in $SPEC$. Thus, p' refines $SPEC$ from R .

Since $R \Rightarrow T$ and T is closed in $p' [\!] F$, in the presence of F, p' is perturbed only to states where T is true. From these states p' eventually reaches a state where R is true, and from that state a computation of p' is in $SPEC$. It follows that in the presence of F, p' refines nonmasking tolerance specification of $SPEC$. Thus, p' is nonmasking F-tolerant to SPEC from R .

Part 2: corrector. We use the definition of Z and X given in the proof of Lemma 4.2 and show that p' is nonmasking F-tolerant to Z corrects X from R and the F-span of p' is T . To this end, we first show that p' refines Z corrects X from R, and then show that $p' [\!] F$ refines the nonmasking tolerance specification of Z detects X from T .

In Lemma 4.2, we have shown that starting from any state in R, every computation of p' satisfies Safeness, Progress, Stability, and Convergence. It follows that p' refines Z corrects X from R .

In Lemma 4.2, we have also shown that p' refines the nonmasking tolerance specification of Z corrects X from T . In the presence of F, this specification may be violated. However,

after faults stop occurring (by Assumption 2, number of faults in a computation are finite), p' eventually reaches a state where R is true. And, from this state, p' refines Z corrects X . Thus, p' is nonmasking F-tolerant to Z detects X from R.

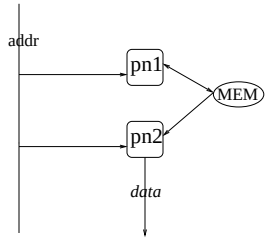
4.3 Example : Memory Access (continued)

Continuing with the example in Section 3.3, consider the case where the given address not in the memory. In this case, an object of the form $\langle addr, - \rangle$ has to be added to the memory. (This object may be obtained from a disk, from a remote memory, or from a network; but we ignore these details.) Thus, nonmasking 'page fault'-tolerance can be achieved by the following program, say pn.

$$\begin{array}{lll} pn1 :: & \neg(\exists\ val :: \langle addr, val\rangle \in MEM) & \longrightarrow \quad MEM := MEM \cup \{\langle addr, - \rangle\} \\ pn2 :: & true & \longrightarrow \quad data := (val | \langle addr, val\rangle \in MEM) \end{array}$$

Program pn consists of two actions: the first action detects whether the given address exists in the memory. If the detection fails, then it adds an appropriate element $\langle addr, - \rangle$ to the memory. The second action is the same as the action of intolerant program p in Section 3.3, and it sets $data$ to the value in the memory.

Program pn is nonmasking 'page fault'-tolerant in the sense that in the absence of faults, it refines $SPEC_{mem}$, and in the presence of a page fault, it refines the nonmasking tolerance specification of $SPEC_{mem}$. More specifically, if no fault occurs, i.e., a tuple $\langle addr, val\rangle$ exists in the memory, it eventually sets $data$ to be equal to val , and it never sets $data$ to any other value. In the presence of a page fault, it may set the $data$ to an incorrect value, but eventually it will set $data$ to the correct value. We use the theory of correctors developed in the previous subsection to show that pn is nonmasking 'page fault'-tolerant.



$X1 \equiv (\exists val :: \langle addr, val\rangle \in MEM)$
$U1 \equiv (Z1 \Rightarrow X1)$

Figure 2: **Memory access**

Let $S := U1 \wedge X1$, $T := U1$, and $F :=$ page fault. Now, observe that p refines $SPEC_{mem}$ from S, pn refines p from S, and $pn[\!]F$ refines $(true)^*(pn \mid S)$ from T. Therefore, by Theorem 4.3, we have

pn is nonmasking 'page fault'-tolerant for $SPEC_{mem}$ from S, and.
pn is a nonmasking 'page fault'-tolerant corrector of an invariant of p .

The alert reader will notice this time that the correction and witness predicate of pn is $X1$ and the corrector is implemented by action $pn1$.

5 Detectors & Correctors and Their Role in Masking Tolerance

In this section, we show that both detectors and correctors exist in masking F-tolerant programs (cf. Theorem 5.5). Also, we show how masking tolerance relates to fail-safe tolerance and nonmasking tolerance (cf. Theorem 5.2).

Lemma 5.1

If

- αs maintains $SPEC$, and
- $s\beta \in SPEC$

then

- $\alpha s\beta \in SPEC$.

Proof.

αs maintains $SPEC \;\wedge\; s\beta \in SPEC$
$=$ { by definition of maintains }
$(\exists \gamma : \alpha s\gamma \in SPEC) \;\wedge\; s\beta \in SPEC$
$\Rightarrow$ { by fusion closure of $SPEC$ }
$\alpha s\beta \in SPEC$ □

Theorem 5.2

If

- p refines $SPEC$ from S,
- p refines $SSPEC$ from T, where $T \Leftarrow S$, and
- p refines $(true)^*(p \mid S)$ from T

then

- p refines the masking tolerance specification of $SPEC$ from T .

Proof. Consider a computation of p, say c, that starts in a state where T is true. Since p refines $(true)^*(p \mid S)$ from T, c contains a state, say s, where S is true. Let αs be the computation prefix of c upto s, and let $s\beta$ be the suffix of c starting from s.

Since p' refines $SSPEC$ from T, the projection of αs on $SPEC$ maintains $SPEC$. And, since p refines $SPEC$ from S, the projection of $s\beta$ on $SPEC$ is in $SPEC$. Therefore, by Lemma 5.1, it follows that the projection of c on $SPEC$ is in $SPEC$. Thus, for every computation of p that starts in a state where T is true, the projection of that computation on $SPEC$ is in $SPEC$, i.e., p refines the masking tolerance specification of $SPEC$ from T . □

Theorem 5.3 combines Theorem 3.4 and Theorem 4.1 to show that if p' is designed by transforming p to satisfy a specification, say $SPEC$, then it contains detectors and correctors.

Theorem 5.3

If

- p refines $SPEC$ from S,
- p' refines p from S
- p' encapsulates p,
- p' refines $(true)^*(p' \mid S)$ from T, where $T \Leftarrow S$, and
- p' refines $SSPEC$ from T

then

- $(\forall\ ac : ac$ is an action of $p : p'$ is detector of a detection predicate of $ac)$, and
- p' is a corrector of an invariant predicate of p .

Proof. The proof follows from Theorem 3.4 and Theorem 4.1. □

We generalize Theorem 5.3, as we did Theorem 4.1, to get Lemma 5.4 .

Lemma 5.4

If

- p refines $SPEC$ from S,
- p' refines p from R, where $R \Rightarrow S$,
- p' encapsulates p,
- p' refines $(true)^*(p' \mid R)$ from T, where $T \Leftarrow R$, and
- p' refines $SSPEC$ from T

then

- $(\forall\ ac : ac$ is an action of $p : p'$ is a masking tolerant detector of a detection predicate of $ac)$, and
- p' is a masking tolerant corrector of an invariant predicate of p .

Proof Part 1: detector. We use the definition of Z and X as in Theorem 3.4. From Theorem 3.5, we observe that p' refines the safety specification, namely Safeness and Stability, of Z detects X from T . We now show that p' also refines the liveness specification, namely Progress, of Z detects X from T .

Consider any computation, say c', of p' which starts in a state where T is true and X is true in each state in c' : Since p' refines $(true)^*(p' \mid R)$ from T, it follows that c' contains a state where R is true. Let $c1'$ be the suffix of c' starting from such a state. Since X is true at each state in c', it follows that X is true at each state in $c1'$ and, hence, g is true in each state in $c1'$. We leave it to the reader to verify that, similar to the proof of Progress in Theorem 3.4, there exists a state in $c1'$ where either Z is true or X is false. Thus, Progress is satisfied.

Part 2: corrector. In general, the predicate S in this theorem may depend on variables of p' that do not occur in p . Since p does not access these additional variables, we can strengthen 'p refines $SPEC$ from S' to 'p refines $SPEC$ from S_p', such that $S \Rightarrow S_p$ and S_p only depends on the variables of p . Specifically, we let

Let $S_p = \{s : s$ is a state of $p' : (\exists s' : s'$ is a state of $p' :$
S is true in state s', and
projection of s on p is the same as the projection of s' on $p)\ \}$

We now show that p refines $SPEC$ from S_p . For this, we first show that S_p is closed in p.

Then, we show that every computation of p that starts in a state where S_p is true is in $SPEC$.

To show that S_p is closed in p, we consider states $s0$ and $s1$ such that S_p is true in state $s0$, and $(s0, s1)$ is a transition of p . By definition of S_p, there exists a state $s0'$ such that S is true in $s0'$ and the projection of $s0'$ on p is the same as the projection of $s0$ on p . Therefore, there exists a transition $s1'$ such that $(s0', s1')$ is a transition of p and the projection of $s1'$ on p is the same as the projection of $s1$ on p . Since S is closed in p, S is true in $s1'$ and, hence, S_p is true in state $s1$. It follows that S_p is closed in p .

By definition of S_p, it follows that $S \Rightarrow S_p$. Thus, every computation of p that starts in a state where S_p is true is in $SPEC$. It follows that p refines $SPEC$ from T .

Now, we use the predicate S_p to define the corrector as follows:

Let $X = S_p$, and
$Z = R$.

We show that p' refines the masking tolerance specification of 'Z corrects X ' from T .

$R \Rightarrow S_p$ follows from $R \Rightarrow S$ and $S \Rightarrow S_p$. Thus, Safeness is satisfied.

Since p' refines $(true)^*(p' \mid R)$ from T, it follows that a computation of p' that starts in a state where T is true eventually reaches a state where R is true. Thus, Progress is satisfied.

Since p' refines p from R, R is closed in p' . Thus, Stability is satisfied.

Since S_p is closed in p, p' encapsulates and S_p only depends on variables of p, S_p is closed in p' . Moreover, a computation of p' starting in a state where T is true eventually reaches a state where R is true and, hence, it reaches a state where S_p is true. It follows that T converges to S_p in p' . Thus, Convergence is satisfied. □

Finally, we use Theorem 5.3 and Lemma 5.4 to show that masking F-tolerant programs contain masking tolerant detectors and correctors. We emphasize, however, that the masking tolerant correctors need not be masking F-tolerant; they may be merely nonmasking F-tolerant. More specifically, the Stability and Convergence property of the corrector may be violated by execution of a fault action in F but these properties are never violated by the execution of a program action.

Theorem 5.5 (Masking F-tolerant programs contain masking tolerant detectors and correctors.)

If
- p refines $SPEC$ from S,
- p' refines p from R, where $R \Rightarrow S$
- $p' [\!] F$ refines $(true)^*(p' \mid R)$ from T, where $T \Leftarrow R$,
- p' encapsulates p, and
- $p' [\!] F$ refines $SSPEC$ from T

then
- p' is masking F-tolerant for $SPEC$ from T,
- $(\forall\ ac : ac$ is an action of $p : p'$ is a masking F-tolerant detector of a detection predicate of $ac)$,
- p' is a masking tolerant corrector of an invariant predicate of p , and
- p' is a nonmasking F-tolerant corrector of an invariant predicate of p .

Proof. Part 1: *masking F-tolerance to $SPEC$.* Since $p' [\!] F$ refines $(true)^*(p' \mid R)$ from T, a computation of $p' [\!] F$, say c' that starts in a state in T, eventually reaches a state, say s,

where R is true. Since $p'[]F$ refines $SSPEC$ from T, the computation prefix upto s maintains $SPEC$. Also, since p refines $SPEC$ from S and $R \Rightarrow S$, the suffix of c starting from state S is in $SPEC$. Therefore, by Lemma 5.1, it follows that c' is in $SPEC$. Thus, a computation of $p'[]F$ that starts in a state in T is in $SPEC$, i.e., p is masking F-tolerant to $SPEC$ from T.

Part 2: *detector.* We use the definition of Z and X in Theorem 3.6 . Theorem 3.6 shows that p' is fail-safe F-tolerant for Z detects X from R and the fault-span of p' is T . To show that p' is masking F-tolerant we need to show that starting from any state in T, p' satisfies the liveness specification of Z detects X, namely Progress. Thus, we need to show that if X is continuously true then in a given computation of $p'[]F$ eventually Z is set to true. Since the number of faults is finite, there exists a suffix of the given computation where X is continuously true and only p executes in that computation. By the proof of Lemma 5.4 (Part 1), it follows that Progress is satisfied. Thus, p' is masking F-tolerant to Z detects X from R .

Part 3: *masking tolerant corrector.* This proof is identical to the proof of Lemma 5.4 (Part 2).

Part 4: *nonmasking F-tolerant corrector.* We use the same definitions of Z and X as in Lemma 5.4 (Part 2), and show that p' is nonmasking F-tolerant to Z corrects X from T and the F-span of p' is T. To this end, we first show that p' refines Z corrects X from T . Then, we show that $p'[]F$ refines the nonmasking tolerance specification of Z corrects X from T .

For the first part, from Lemma 5.4, we observe that p' refines Z corrects X from T .

For the second part, we observe that in the presence of F, stability of the corrector may be violated. However, since faults are finite, after the faults stop, the computation of p' alone is in Z corrects X . Thus, each computation of $p'[]F$ has a suffix that is in Z corrects X . In other words, $p'[]F$ refines the nonmasking tolerance specification of Z corrects X from T . □

5.1 Example : Memory Access (continued)

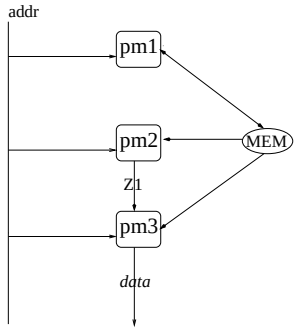
Continuing with the example in Section 4.3, consider the following program that is masking 'page fault'-tolerant:

$$
\begin{array}{lllll}
pm1 :: & \neg(\exists\ val :: \langle addr, val\rangle \in MEM) & & \longrightarrow & MEM := MEM \cup \{\langle addr, -\rangle\} \\
pm2 :: & (\exists\ val :: \langle addr, val\rangle \in MEM) \ \wedge & \neg Z1 & \longrightarrow & Z1 := true \\
pm3 :: & Z1 \wedge true & & \longrightarrow & data := (val | \langle addr, val\rangle \in MEM)
\end{array}
$$

Program pm consists of three actions: the first action adds a tuple $\langle addr, -\rangle$ if such a tuple does not exist in the memory. The second action detects if a tuple of the form $\langle addr, -\rangle$ exists in the memory. If this detection succeeds, it sets $Z1$ to true. Finally, the third action sets $data$ after $Z1$ is set to true.

Let $S := U1 \wedge X1$, $T := U1$, and $F :=$ page fault. Observe that pn refines $SPEC_{mem}$ from S, pm refines p from S, $pm[]F$ refines $(true)^*(pm|S)$ from T, pm encapsulates pn, and $pm[]F$ refines the safety specification of $SPEC_{mem}$ from T . Therefore, by Theorem 5.5, we have

pm is masking 'page fault'-tolerant to $SPEC_{mem}$ from S,
pm is a masking 'page fault'-tolerant detector of a detection predicate of $pn1$ (respectively $pn2$),
pm is a masking 'page fault'-tolerant corrector of an invariant of pn .



$X1 \equiv (\exists val :: \langle addr, val \rangle \in MEM)$
$U1 \equiv (Z1 \Rightarrow X1)$

Figure 3: **Memory access**

6 Role of Detectors and Correctors in Existing Methods

In this section, we show that detectors and correctors exist in programs designed using extant methods for fault-tolerance. More specifically, we show this for the canonical programs designed using two extant methods, namely replication and Schneider's state machine approach. Our approach is in fact constructive: we show that canonical fault-tolerant programs designed using these two methods can be designed by adding detectors and correctors to a fault-intolerant program.

Regarding replication, we focus our attention in Section 6.1 on the problem of triple modular redundant system design. And, regarding Schneider's state machine approach [14], we focus our attention in Section 6.2 on one of its requirements, namely *Agreement*, in the presence of Byzantine faults. For reasons of space, the other requirement, namely *Order*, is discussed in [11].

6.1 Triple modular redundancy

Consider a triple modular redundant system used for the input-output problem: the system consists of three inputs, say x, y and z, and one output, say out. In the absence of faults, all inputs are identical. Faults may corrupt any one of the three inputs. The specification of input-output problem, $SPEC_{io}$, requires that the output be assigned the value of an uncorrupted input.

Below, we show that the triple modular redundant system can be designed by first designing a fault-intolerant system, IR, and then adding to it a detector, DR, followed by a corrector, CR.

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

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

Detector DR. Observe that IR violates its safety specification from states where the value of x is corrupted. To preserve the safety specification, we will use a detector DR. Letting $uncor$ be the value of an uncorrupted input, the detection predicate of DR is $(x = uncor)$, and the

witness predicate of DR is $(x=y \vee x=z)$. Observe that $(x=y \vee x=z)$ detects $(x=uncor)$ in the program that merely evaluates the state predicate $(x=y \vee x=z)$ upon starting from the states where at most one one input value is corrupted. To add fail-safe tolerance, IR is restricted to execute only when the witness predicate of DR is satisfied.

Letting S be the state of states where no input is corrupted, T be the state of states where at most one input is corrupted, and F be the fault that corrupts at most one input, we observe that IR refines $SPEC_{io}$ from S, $DR;IR$ refines IR from S, $DR;IR$ encapsulates IR, and $(DR;IR)[]F$ refines the safety specification of $SPEC_{io}$ from T. Therefore, by Theorem 3.6, we have

$DR;IR$ is fail-safe 'one input corruption'-tolerant,
$DR;IR$ is a fail-safe 'one input corruption'-tolerant detector of a detection predicate of IR.

Corrector CR. Program $DR;IR$ deadlocks when the value of x gets corrupted. To achieve masking tolerance, we add corrector CR whose correction predicate and witness predicate are both $out=uncor$. CR consists of two actions: if the value of y is uncorrupted, y is copied into the output, and if the value of z is uncorrupted, z is copied into the output. These actions are as follows:

$$\begin{array}{lllll} CR1 :: & out=\bot \ \wedge \ (y=z \ \vee \ y=x) & \longrightarrow & out := y \\ CR2 :: & out=\bot \ \wedge \ (z=x \ \vee \ z=y) & \longrightarrow & out := z \end{array}$$

Thus, we have

$DR;IR \ [] \ CR$ is masking 'one input corruption'-tolerant.

Observe that $DR;IR \ [] \ CR$ is the triple modular redundancy program, and, by construction, it contains detectors and correctors.

6.2 Byzantine agreement

Consider the problem of Byzantine agreement: a unique general g outputs a binary value $d.g$, and every non-general process eventually outputs its decision subject to the following two conditions: (1) if g is not subject to a Byzantine fault, the decision output by all non-Byzantine processes are identical to $d.g$, and (2) even if g is subject to a Byzantine fault the decision output by all non-Byzantine processes are identical.

Byzantine faults corrupt processes permanently and undetectably such that the corrupted processes execute arbitrarily nondeterministic actions. It is well known that masking tolerant Byzantine agreement is possible iff there are at least $3f+1$ processes, where f is the number of Byzantine processes [12]. For ease of exposition, we will restrict our attention to the case where the total number of processes (including g) is 4 and, hence, f is 1. (We discuss the general case where f is greater than one elsewhere [11].)

We show that a Byzantine agreement program can be designed by first designing a fault-intolerant program, IB, and then adding to it a detector, DB, followed by a corrector, CB.

Fault-intolerant program IB. IB consists of two actions for each non-general process, j: the first action copies the decision $d.g$ of the general into the $d.j$. The value $\bot$ of $d.j$ is used to denote that j has not yet copied the decision of the general. (We assume that $d.g$ is not $\bot$.) The second action outputs the decision of j. Thus, the actions of j are as follows:

$$\begin{array}{lllll} IB1.j :: & d.j = \bot & \longrightarrow & d.j := d.g \\ IB2.j :: & d.j \neq \bot & \longrightarrow & \{ \text{ output } d.j \ \} \end{array}$$

To represent Byzantine faults, we introduce an auxiliary variable $b.j$ at each process j (including the general). $IB1.j$ and $IB2.j$ are executed when $b.j$ is false, i.e., j is non-Byzantine. If $b.j$ is true, i.e., process j is Byzantine, j is allowed to change its decision arbitrarily, or output an arbitrary decision. Thus, the Byzantine fault at process j is represented by the action that changes $b.j$ from false to true, thereby permitting the process to enter the Byzantine mode. In other words, to each process j, $BYZ.j$ is added that consists of the following two actions:

$$\begin{array}{lllll} BYZ1.j :: & b.j & \longrightarrow & d.g := 0|1 \\ BYZ2.j :: & b.j & \longrightarrow & \{ \text{ output } 0|1 \ \} \end{array}$$

Detector DB. Observe that a non-general process j violates the safety specification when it executes $IB2$ from states where g is Byzantine. To preserve the safety specification, we add a detector $DB.j$ to each non-general process j. $DB.j$ ensures that the decision being output by j satisfies the safety specification of Byzantine agreement, by checking that $d.j$ equals $corrdecn$, where

$$\begin{array}{llll} corrdecn & = & d.g & \text{if } \neg b.g \\ & & (majority\ j : j \neq g : d.j) & \text{otherwise} \end{array}$$

The detection predicate of $DB.j$ is $d.j = corrdecn$, and the witness predicate of $DB.j$ is $((\forall k : k \neq g : d.k \neq \bot) \ \wedge \ d.j = (majority\ k : k \neq g : d.k))$. $DB.j$ contains actions that let each non-general process copy the decision from the general, i.e., $DB.j$ consists of the action $IB1.k$ at the non-general processes. Thus, we have

$BYZ.g \ \ [\!] \ \ ([\!]j : j \neq g : IB1.j \ [\!] \ DB.j; IB2.j \ [\!] \ BYZ.j)$ is fail-safe Byzantine-tolerant.

Corrector CB. In program $BYZ.g \ \ [\!] \ \ ([\!]j : j \neq g : IB1.j \ [\!] \ DB.j; IB2.j \ [\!] \ BYZ.j)$, if g is Byzantine and sends different values to non-general processes, one non-general process will be blocked from being able to output its decision. To add masking tolerance, we add a corrector $CB.j$ to each non-general process j. The correction predicate of $CB.j$ is $d.j = corrdecn$ and its witness predicate is $((\forall k : k \neq g : d.k \neq \bot) \ \wedge \ d.j = (majority\ k : k \neq g : d.k))$. The actions of $CB.j$ consist of the actions $IB1.k$ at the non-general processes and the action $CB1.j$, where

$$CB1.j :: \ \ (\forall k : k \neq g : d.k \neq \bot) \ \wedge \ d.j \neq (majority\ k : k \neq g : d.k) \ \ \longrightarrow \ d.j \ := (majority\ k : k \neq g : d.k)$$

Thus, we have

$BYZ.g \;\; [] \;\; ([]j : j \neq g : IB1.j \;\; [] \;\; DB.j; IB2.j \;\; [] \;\; CB.j \;\; [] \;\; BYZ.j)$
is masking Byzantine-tolerant.

Observe that $BYZ.g \;\; [] \;\; ([]j : j \neq g : IB1.j \;\; [] \;\; DB.j; IB2.j \;\; [] \;\; CB.j \;\; [] \;\; BYZ.j)$ is the Byzantine agreement program where at most one process is Byzantine. And, by construction, it contains detectors and correctors.

7 Concluding Remarks

In this paper, we presented a theory of detectors and correctors to show that they are integral parts of a rich class of fault-tolerant programs that includes those designed using encapsulation and refinement. More specifically, we showed that (1) programs refining a safety specification contain detectors, and fail-safe F-tolerant programs contain fail-safe tolerant detectors, (2) programs that eventually refine a specification contain correctors, and nonmasking F-tolerant programs contain nonmasking tolerant correctors, and (3) masking F-tolerant programs contain masking tolerant detectors and correctors.

We showed that detectors and correctors also exist in fault-tolerant programs designed using extant methods such as replication and Schneider's state machine approach. Our approach was constructive in that we demonstrated how programs designed using these methods can be alternatively designed using detectors and correctors.

We note that our notion of detectors also applies to the recent work of Chandra and Toueg on failure detection [7], although that work makes some distinctions among failure detectors –such as strong and weak accuracy– that we do not need for our purposes. Our detectors are more general than failure detectors in the sense that the failure detectors are instantiations of detectors whose detection predicates are of the form 'the given process is *down*' or 'the given process is *up*'. Unlike detectors that focus on states reached in the execution of the program and the faults, failure detectors focus on the states reached immediately after the fault and, hence, for a given problem detectors are typically more abstract than failure detectors. Also, for a given problem it is possible to design the detectors required for designing a fault-tolerant program for that problem using failure detectors.

As mentioned in the introduction, in related work [4], we have addressed how detectors and correctors can be added to a fault-intolerant program to obtain a fault-tolerant program. Our design method has been used used to provide multitolerant and efficient solutions to barrier computations, repetitive Byzantine agreement, mutual exclusion, tree maintenance, leader election, termination detection and bounded-space network management [4, 10, 11, 5]. Based on this experience and the existence of a rich class of programs that contain detectors and correctors, we conclude that detectors and correctors provide a powerful basis for efficient, component-based design of fault-tolerance.

One observation of interest is that detectors and correctors required in one program as well as across different programs are often similar. Therefore, we are developing a framework of such components. This framework will speed up the development time for a new fault-tolerant program as instantiation of the framework may be used to design the components required for the problem at hand. It will also simplify proofs of interference freedom between components when we can discharge these proofs at the framework level.

We are also working on mechanized verification and synthesis of component based fault-tolerant programs. Towards mechanized verification, we are encoding the theory of detectors and correctors –including the theory that deals with adding these components to obtain fault-tolerant programs– into the theorem-prover PVS [15]. Using the partial theory we have currently encoded in PVS, we have mechanically proved the correctness of Dijkstra's token ring program [9] in a compositional manner. Towards mechanized synthesis, we are developing a method that will synthesize the fault-tolerance components required for achieving each of the tolerance requirements, and compose these components with the given fault-intolerant program to obtain a fault-tolerant program.

To facilitate the implementation of fault-tolerant programs, we are currently developing a tool called SIEFAST. SIEFAST provides an environment that enables stepwise design, implementation and validation of component-based fault-tolerant distributed programs. It also permits distributed and hybrid simulations; in a distributed simulation, the processes in a distributed program are run in parallel, typically on different machines. In a hybrid simulation, some components of the program are implemented while others are simulated. A hybrid simulation allows stepwise refinement of programs where we can implement the program one component at a time and still continue to verify the functionality, fault-tolerance and performance in the intermediate steps. Finally, apart from faults, SIEFAST allows modeling of intruders, and permits simulation and verification of security protocols.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

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

[4] A. Arora and S. S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[5] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, pages 435–450, June 1998. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995, pages 174–185.

[6] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), 1996.

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

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

[10] S. S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization*, 1998, to appear.

[11] S. S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive byzantine agreement. *Proceedings of the Seventeenth International Conference on Foundations of Software Technology and Theoretical Computer Science, Kharagpur, India*, pages 169–183, December 1997.

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

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

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

[15] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is released in 1998.

Appendix : Notation

Symbols	
$p, q, p', d, c, pf, pn, pm$	programs
F	faults
$s, s0, s1, s', s0', s1'$	program states
$ac, ac', ac1, ac1'$	program actions
$R, S, T, S_p, S', sf, X, Z, U, sf1, X1, Z1, U1, sf2$	state predicates
$c, c1, c', c1'$	program computations
$\alpha, \beta, \gamma, \delta$	state sequences
$SPEC$	problem specification
$SSPEC$	safety specification

Program compositions	
$[\!]$	parallel composition
$\wedge$	restriction
;	sequential composition

Propositional connectives (in decreasing order of precedence)	
$\neg$	negation
$\wedge, \vee$	conjunction, disjunction
$\Rightarrow, \Leftarrow$	implication, consequence
$\equiv, \not\equiv$	equivalence, inequivalence

First order quantifiers	
$\forall, \exists$	universal, existential
$\forall x : R(x) : S(x)$	forall x that satisfy $R(x)$, $S(x)$ is true
$\exists x : R(x) : S(x)$	there exists x such that $R(x)$ and $S(x)$ is true