Component Based Design of Fault-Tolerance

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Sandeep Kulkarni, B.Tech, M.S

* * * * *

The Ohio State University

1999

Dissertation Committee:

Professor Anish Arora, Adviser

Professor Mukesh Singhal

Professor Ten-Hwang Lai

Approved by

____

Adviser
Department of Computer
and Information Science

ABSTRACT

In this dissertation, we defend the thesis that a fault-tolerant program is a composition of a fault-intolerant program and a set of *fault-tolerance components*. To validate this thesis, we identify a basis set of fault-tolerance components, namely *detectors* and *correctors*. We show that depending upon the desired level of fault-tolerance, detectors, correctors, or both are necessary and sufficient for designing a rich class of fault-tolerant programs. Moreover, the fault-tolerant programs designed using existing methods such as replication and Schneider's state machine approach can be alternatively designed in terms of detectors and correctors.

Using these fault-tolerance components, we present a method for designing multi-tolerant programs, i.e., programs that tolerate multiple types of faults while providing potentially different levels of tolerance to each type of fault. Our method accommodates all types of faults, including process faults, communication faults, hardware and software faults, network failure and security intrusions, and it can be used in several application domains such as distributed systems, computer networks and parallel systems.

Using several examples, we illustrate how our fault-tolerance components can be used in the design of several fault-tolerant programs. Among these examples, we discuss in detail the problem of distributed reset. A distributed reset operation reinitializes a distributed system to a given global state. Our distributed reset program

is the first bounded state program that masks fail-stop and repair of processes and stabilizes in the presence of transient faults. In other words, it guarantees that if only fail-stop and repair faults occur then every reset operation is correct and if transient faults occur then eventually the program reaches a state from where subsequent reset operations are correct. Towards designing this distributed reset program, we have designed multitolerant components that are useful in adding multitolerance to several other applications.

We also argue that the decomposition of a fault-tolerant program into its fault-tolerance components is useful in verification of that program. Towards this end, we present a case study that illustrates our experience in verifying Dijkstra's token ring program with the theorem prover PVS.

Dedicated to my parents (Sadashiv and Savita) and my brother (Suhas),

for their love, faith and support

VITA

June 5, 1972	Born - Sangli, India
1993	B.Tech. Computer Science and Engineering Indian Institute of Technology, Bombay, India.
1994	M.S. Computer and Information Science, Ohio State University.
Winter98 to Winter99	Presidential Fellow, Ohio State University.
Summer98	Summer intern, Computer Science Laboratory SRI International.
Summer97 to Autumn97	Graduate Teaching Associate, Ohio State University.
Spring95 to Spring97	Graduate Research Associate, Ohio State University.
Autumn93 to Winter95	Graduate Teaching Associate, Ohio State University.
Summer91	Summer intern, Indian Institute of Technology, Bombay, India.

PUBLICATIONS

Research Publications

D. M. Dhamdhere and S. S. Kulkarni. "A token based k resilient mutual exclusion algorithm for distributed systems." *Information Processing Letters*, 50:151–157, 1994.

A. Arora and S. S. Kulkarni. "Designing masking fault-tolerance via nonmasking fault-tolerance." *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr*, 14:174–185, 1995.

S. S. Kulkarni and A. Arora. "Multitolerant barrier synchronization." *Information Processing Letters*, 64(1):29–36, October 1997.

S. S. Kulkarni and A. Arora. "Once-and-forall management protocol (OFMP)." *Proceedings of the Fifth International Conference on Network Protocols*, pages 87-94, October 1997.

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. A preliminary version of this paper appears in the Third Workshop on Self-Stabilizing Systems (WSS97), University of California, Santa Barbara, 1997, pages. 1-15.

A. Arora and S. S. Kulkarni. "Component based design of multitolerant systems." *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

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.

A. Arora and S. S. Kulkarni. "Designing masking fault-tolerance via nonmasking fault-tolerance." *IEEE Transactions on Software Engineering*, pages 435–450, June 1998.

S. S. Kulkarni and A. Arora. "Low-cost fault-tolerance in barrier synchronizations." *International Conference on Parallel Processing*, pages 132-139, August 1998.

S. Kulkarni and A. Arora. "Multitolerance in distributed reset." *Chicago Journal of Theoretical Computer Science*, 1998(4), December 1998.

S. S. Kulkarni, J. Rushby, and N. Shankar. "A case-study in component-based mechanical verification of fault-tolerant programs." *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Fault-tolerance: Prof. Anish Arora

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Fault-tolerance is the ability of a system to deliver desired level of functionality in the presence of faults that subject it to a less than ideal environment. This ability to function in an adverse environment is crucial in many systems, including telecommunication, electronic commerce, manufacturing and power systems. Moreover, as these systems are becoming more interdependent and their expected level of service is increasing, the need for fault-tolerance in these systems is also increasing.

Various methods have been proposed in the literature for designing fault-tolerance: These methods include replication, Schneider's state machine approach [59], checkpointing and recovery [37, 62] and recovery blocks [57]. These methods are limited in terms of the types of faults they can handle as well as the types of systems they can be employed in.

For example, in a replication based system, the fault-tolerant system consists of multiple copies of the fault-intolerant system and the output of the system is obtained by composing the outputs of these fault-intolerant systems. Such replication based methods can only deal with faults such as fail-stop of processes and Byzantine processes. Moreover, replication based techniques can be used only in systems where the fault-intolerant system is deterministic, i.e., for any given input there is only one correct output. Schneider's state machine approach which generalizes replication to the client-server model also suffers from the same limitations. In a checkpointing-and-recovery based system, after the detection of a fault, the system is restored to some previous state in its computation. For this reason, checkpointing-and-recovery method can only deal with systems that are subject to detectable faults such as fail-stop and repair of processes, channel failures and message loss.

These limitations of the existing methods are of a serious concern in the developing modern fault-tolerant systems that are subject to various types of faults such as process faults, communication faults, hardware faults, software faults, network failure, security intrusions, safety hazards, configuration changes and load variations. The development of these systems is further complicated since these faults may occur at any time: in particular, a fault of one type may occur while the system is recovering from another. Also, it is often necessary to modify existing fault-tolerant systems to deal with new types of faults that were not considered in the original design. Moreover, application-specific methods for fault-tolerance are not desirable in the design of these systems since they rarely allow the techniques used in one design to be reused in other designs.

In sum, there is a need for a systematic method for the design of fault-tolerant systems that

1. can deal with a rich class of faults,

2. can be used to make a rich class of systems fault-tolerant,

3. provides the potential to design efficient fault-tolerant programs,

4. can be used to incrementally add tolerance to a new type of fault, and

5. is not application-dependent.

In this dissertation, we present such a systematic method that is based on identifying the basic components in a fault-tolerant system.

1.1 Thesis

In this dissertation, we present a systematic method for designing fault-tolerance that is based on the following thesis:

> **A fault-tolerant program is a composition of a fault-intolerant program and a set of *fault-tolerance components***

This thesis suggests how the fault-tolerance components can be used in the design of new fault-tolerant programs as well as in the analysis of a given fault-tolerant program as follows:

1. The thesis suggests that given a fault-intolerant program (which provides the functionality in the absence of faults but cannot deal with faults), if suitable fault-tolerance components (which only provide the fault-tolerance capability) can be designed and added to that fault-intolerant program then the resulting program will be fault-tolerant.

2. The thesis also suggests that given a fault-tolerant program, it can be decomposed into a fault-intolerant program and a set of fault-tolerance components. It follows that such a decomposition can be used to analyze a given fault-tolerant program.

Towards defending this thesis, in this dissertation, we address the following questions:

1. *Is there a basis set of fault-tolerance components?*

   In this dissertation, we identify two fault-tolerance components, namely *detectors* and *correctors* that form one such basis. We show that for a rich class of fault-tolerant programs, including those that can be designed using existing methods, these components are both necessary and sufficient: the necessity is shown by the fact that any program in this class contains detectors and/or correctors, and the sufficiency is shown by the fact that any program in this class can be designed in terms of detectors and/or correctors (cf. Chapters 3 and 4).

2. *How can these components be used in the design of fault-tolerant programs, including* multitolerant *[11] programs, that tolerate multiple types of faults while providing different levels of tolerance to each type of fault?*

   We present a method that uses detectors and correctors and allows stepwise addition of fault-tolerance properties to a given fault-intolerant program. The method shows how to compute the specification of the desired fault-tolerance components and how to design these components in an hierarchical fashion in terms of smaller components. It also provides the ability to incrementally add fault-tolerance to an existing system (cf. Chapters 5 and 6).

3. *What are the benefits of designing a program in terms of detectors and correctors?*

   Using detectors and correctors, we have presented fault-tolerant solutions for several problems such as distributed reset [38], leader election [10], mutual exclusion [12], data transfer [12], network management [42], resource synchronization [41] and Byzantine agreement [39]. These solutions outperform previously known solutions in terms of efficiency or the tolerance level. In this dissertation, we present some of these solutions in Chapters 5, 6 and 8.

4. *What are the benefits in analyzing a program in terms of its detectors and correctors?*

   We have used the decomposition of a program into its components in the verification of several fault-tolerant programs. Our experience shows that the decomposition of a fault-tolerant program into detectors and correctors often provides a better understanding of that program, and such a decomposition is also useful in verification. We find that the advantages of the component-based verification also apply in mechanical verification of fault-tolerant programs. In Chapter 9, we discuss our experience in the mechanical verification of Dijkstra's token ring program [25].

5. *How does our method compare with existing methods for designing fault-tolerance?*

   The use of detectors and correctors is more general than existing methods in the sense that programs designed using the existing methods can be alternatively designed in terms of detectors and correctors. More specifically, we show how programs designed using replication and Schneider's state machine approach can be alternatively designed in terms of detectors and correctors (cf. Chapter 7).

1.2 Outline of the Dissertation

We proceed as follows: In Chapter 2, we give a formal definition of programs, problem specifications, faults and fault-tolerances. Using these definitions, in Chapters 3 and 4, we identify a basis set of fault-tolerance components. We identify the role of these components in Chapters 3, 4 and 5. In Chapter 6, we present our method for designing multitolerance. In Chapter 7, we show how the use of detectors and correctors relates to existing methods. Subsequently, in Chapter 8, we show how we used the detectors and correctors in the design of a multitolerant and bounded distributed reset program. In Chapter 9, we illustrate how the decomposition of a fault-tolerant program into its fault-tolerance components is useful in mechanical verification. Finally, we make concluding remarks in Chapter 10.

CHAPTER 2

PRELIMINARIES

Recall the thesis presented in the previous chapter: "A fault-tolerant program is a composition of a fault-intolerant program and a set of fault-tolerance component". Towards validating the thesis, we first need to formally define what a program is, what a fault is, what it means for a program to be fault-tolerant, and how fault-tolerance components are composed with a fault-intolerant program. In this chapter, we provide these definitions that enable us to define the fault-tolerance components in the next two chapters.

First, we formalize a program based on the work by Chandy and Misra [22], and then define several program compositions that are used in adding fault-tolerance components to a fault-intolerant program (cf. Section 2.1). Intuitively, a program is represented in terms of its state transitions. We choose this notation as it readily includes programs in any imperative language. Then, we define problem specifications based on the work by Alpern and Schneider [3] (cf. Section 2.2). A problem specification is represented as a set of state sequences. Subsequently, we adopt from the work by Arora and Gouda and represent faults also as state transitions (cf. Section 2.3). We choose this representation as it allows us to model a rich class of faults. Finally, we define what it means for a program to be fault-tolerant (cf. Section 2.4).

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:

$$\langle name \rangle \ :: \ \langle guard \rangle \longrightarrow \langle statement \rangle$$

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

Notation. To conveniently write an action as a restriction of another action, we use the notation

$$\langle name' \rangle \ :: \ \langle guard' \rangle \ \wedge \ \langle name \rangle \parallel \langle statement' \rangle$$

to define an action ⟨name′⟩ whose guard is obtained by restricting the guard of action ⟨name⟩ with ⟨guard′⟩, and whose statement is obtained by superposing the statement of action ⟨name⟩ with ⟨statement′⟩. Operationally speaking, ⟨name′⟩ is executed only if the guard of ⟨name⟩ and the guard ⟨guard′⟩ are both true. And, to execute ⟨name′⟩, both the statement of ⟨name⟩ and ⟨statement′⟩ are executed atomically. Likewise, to conveniently write a program as a restriction of another program, we use the notation

$$\langle guard \rangle \wedge \langle program \rangle \parallel \langle statement' \rangle$$

to define a program consisting of the set of actions ⟨guard⟩ $\wedge$ ac $\parallel$ ⟨statement′⟩ for each action ac of ⟨program⟩. *(End of Notation.)*

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. $\square$

Definition (*State Predicate*). A state predicate of p is a boolean expression over the variables of p . $\square$

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. $\square$

2.1.1 Program Compositions

Definition (*Parallel ($\|$) 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 (*Restriction ($\wedge$) 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 (*Sequential (;) 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. In a '; composition' when the predicate Z is clear from the context, we write $p; q$ to mean $p \;;_Z\; q$.

Remark. Note that in a sequential composition $p\ ;_Z\ q$, p may continue to execute after Z becomes true. However, q may execute only in states where Z is true. If Z is true then p and q may execute concurrently.

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 $g' \wedge ac \parallel st'$, where ac is an action of p *and* st' does not update variables of p .

Notation. When variables of a program are clear from the context, we omit them and simply present the actions of the program.

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. $\square$

Note that the state sequences in a problem specification may be finite or infinite. Following Alpern and Schneider [3], 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, where safety and liveness specification are defined as follows:

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). $\square$

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. $\square$

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}. $\square$

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 $k, k \geq j$. $\square$

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. $\square$

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

2.2.1 Program Correctness with respect to a Problem Specification

Let *SPEC* be a problem specification.

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

Definition (*Computation Projection*). The projection of a computation of p' on p (respectively $SPEC$) is the 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*. $\square$

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

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*. $\square$

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. $\square$

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. $\square$

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 and $SPEC$ is clear from the context, we will omit it; thus, "S is an invariant" abbreviates "S is an invariant of p for $SPEC$", "T is a fault-span" abbreviates "T is a F-span 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. Towards defining the nonmasking tolerance specification of $SPEC$, we use the following notation.

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 $\Gamma\Delta = \{ \alpha\beta : \alpha \in \Gamma \text{ and } \beta \in \Delta \}$. Thus, $(true)^* SPEC$ denotes the set of sequences which have a suffix in $SPEC$.

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 $(true)^* SPEC$, i.e., the nonmasking tolerance specification of $SPEC$ is the set of sequence which have a suffix in $SPEC$. □

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 for $SPEC$ 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 for $SPEC$ from S' as follows:

Definition (*F-tolerant for* $SPEC$ *from* S). p is masking F-tolerant for $SPEC$ from S (respectively nonmasking F-tolerant for $SPEC$ from S or fail-safe F-tolerant for $SPEC$ 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) . □

We define stabilizing tolerance as a special case of nonmasking tolerance where the program recovers from an arbitrary state.

Definition (*Stabilizing F-tolerant for $SPEC$ from S*). p is stabilizing F-tolerant for $SPEC$ from S iff the following two conditions hold:

- p refines $SPEC$ from S, and
- $p \parallel F$ refines the nonmasking tolerance specification of $SPEC$ from $true$ □

The type of tolerance characterizes the 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 $SPEC$. Stabilizing tolerance is a special case of nonmasking tolerance, where the F-span is $true$.

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. Also, we use the term "masking tolerant to F" to mean "masking F-tolerant".

2.5 A Note on Assumptions

For the reader's convenience, we reiterate and justify the assumptions made in this dissertation 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.

This assumption is justified by the observation that conventional specification languages typically yield specifications that are both suffix closed and fusion closed. Moreover, even if a given specification is not suffix closed and/or fusion closed, based on the following observation, the results in this dissertation can still be applied: Given a set of sequences L that is not suffix closed and/or fusion closed, it is possible (by adding a "history" variable) 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, where p' is a program obtained by modifying p such that the history variable is updated appropriately. Thus, if the given specification is not suffix closed or fusion closed it is still possible to determine the detection predicates, although they may depend on the added history variable.

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.

CHAPTER 3

DETECTORS : A BASIS OF FAIL-SAFE FAULT-TOLERANCE

In Chapter 2, we considered three types fault-tolerant programs: fail-safe, nonmasking and masking. In this chapter, we identify the first of the two fault-tolerance components, *detector*, that forms a basis of fail-safe fault-tolerance. Towards showing that detectors form a basis of fail-safe fault-tolerance, we show that (1) detectors are sufficient to design fail-safe fault-tolerant programs, i.e., a rich class of fail-safe fault-tolerant programs can be designed using detectors, and (2) detectors are necessary to design fail-safe fault-tolerance, i.e., a rich class of fail-safe fault-tolerant programs contain detectors. We also note that the commonly used techniques in designing fail-safe fault-tolerant programs such as comparators, error detection codes, consistency checkers, watchdog programs, snoopers, alarms, snapshot procedures, acceptance tests, and exception conditions are instances of detectors.

We proceed as follows: First, we give a formal definition of a detector (cf. Section 3.1). Then, we show how detectors can be constructed in an hierarchical and efficient manner (cf. Section 3.2). Subsequently, we show that detectors are sufficient and necessary in the design of fail-safe fault-tolerance (cf. Sections 3.3 and 3.4). Finally, we present sufficiency conditions that ensure that when a detector is added to a program, the detector and the program do not interfere with each other.

3.1 Definition

Definition (*detects*). Let X and Z be state predicates. Let 'Z detects X' be the problem specification that is the set of all sequences, $s_0, s_1, ...$ that satisfy 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, $(\{Z\}, \{Z \vee \neg X\})$.) □

Definition (*detector*). Let d be a program. 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 refines *Safeness* from U, it follows that d never lets Z witness X incorrectly. Since d refines *Progress* from U, it follows that if $U \wedge X$ is true continuously, d eventually detects this fact and truthifies Z. Moreover, since d refines *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 [22]. 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.)

3.1.1 Properties of the *detects* relation

The detects relation is reflexive, antisymmetric, and transitive in its first two arguments:

Lemma 3.1 Let X, Y and Z be state predicates of d and U be a state predicate that is closed in d. The following statements hold.

- X detects X in d from U .
- If Z detects X in d from U, and X detects Z in d from U
  then $U \Rightarrow (Z \equiv X)$.
- If Z detects Y in d from U, and Y detects X in d from U
  then Z detects X in d from U . □

Lemma 3.2 Let V be a state predicate such that $U \wedge V$ is closed in d. The following statements hold.

- If $\quad Z$ detects X in d from U

  then $\quad Z$ detects X in d from $U \wedge V$.

- If $\quad Z$ detects X in d from U, and $V \Rightarrow X$

  then $\quad Z \vee V$ detects X in d from U .

- If $\quad Z$ detects X in d from U, and $Z \Rightarrow V$

  then $\quad Z$ detects $X \wedge V$ in d from U . $\square$

3.2 Hierarchical and Distributed Design of Detectors

As discussed later in this chapter, to design a fail-safe detector we need to design a detector with a given detection predicate. It would be ideal if we could check whether this detection predicate holds atomically, i.e., by executing at most one action of the detector. In certain cases, evaluating the detection predicate may require access to a large portion of a shared state or access to a state that is distributed across different processes. Therefore, it is sometimes difficult or impossible to check the detection predicate atomically. In these cases, we may wish to compose "small" detectors to obtain "large" detectors in an hierarchical and distributed manner. More specifically, if we need to design a detector whose detection predicate is $X1 \wedge X2$ then we may choose to design a detector with detection predicate $X1$ and another with detection predicate $X2$. Then, we can compose these detectors in two ways: (i) in parallel and (ii) in sequence.

In the rest of the chapter, we will implicitly assume that the problem specification of a detector d (dn) is 'Z detects X in d from U' (respectively, 'Zn detects Xn in dn

from Un'). Also, we will implicitly assume that U (Un) is closed in d (respectively, dn).

Parallel composition of detectors. In the parallel composition of $d1$ and $d2$, denoted by $d1 \| d2$, both $d1$ and $d2$ execute concurrently. Formally, the parallel composition of $d1$ and $d2$ is the union of the (variables and actions of) programs $d1$ and $d2$.

Observe that '$\|$' is commutative ($d1 \| d2 = d2 \| d1$), associative ($(d1 \| d2) \| d3 = d1 \| (d2 \| d3)$), and that '$\wedge$' distributes over '$\|$' ($g \wedge (d1 \| d2) = (g \wedge d1) \| (g \wedge d2)$).

Theorem 3.3 Given $Z1$ detects $X1$ in $d1$ from U and $Z2$ detects $X2$ in $d2$ from U.

If the variables of $d1$ and $d2$ are mutually exclusive
then $Z1 \wedge Z2$ detects $X1 \wedge X2$ in $d1 \| d2$ from U.

Proof. *Safeness* of $d1 \| d2$ follows from *Safeness* of $d1$ and that of $d2$. Also, *Stability* of $d1 \| d2$ follows from *Stability* of $d1$ and that of $d2$. To prove *Progress* of $d1 \| d2$, we partition its computations into two classes: (1) where $X1 \wedge X2$ is falsified in some state, and (2) where $X1 \wedge X2$ is never falsified. In the first class, *Progress* is satisfied trivially. In the second class, from *Progress* of $d1$, $Z1$ is eventually truthified and, from *Stability* of $d1$, $Z1$ continues to be true in the computation of $d1$. Moreover, since the variables of $d1$ and $d2$ are disjoint, $Z1$ continues to be true in the computation of $d2$. Finally, from *Progress* of $d2$, $Z2$ is eventually truthified. Thus, *Progress* of $d1 \| d2$ is satisfied. $\square$

Sequential composition of detectors. In the sequential composition of $d1$ and $d2$, denoted by $d1; d2$, $d2$ executes only after $d1$ has completed its detection, i.e., after the witness predicate $Z1$ is true. Formally, the sequential composition of $d1$ and $d2$ is the program whose set of variables is the union of the variables of $d1$ and of $d2$ and whose set of actions is the union of the actions of $d1$ and of $Z1 \wedge d2$. We postulate the axiom that ';' is left-associative $(d1; d2; d3 = (d1; d2); d3)$.

Observe that ';' is not commutative, that ';' distributes over '$\|$' $(d1; (d2 \| d3) = (d1; d2) \| (d1; d3))$, and that '$\wedge$' distributes over ';' $(g \wedge (d1; d2) = (g \wedge d1); (g \wedge d2))$.

Theorem 3.4 Given $Z1$ detects $X1$ in $d1$ from U and $Z2$ detects $X2$ in $d2$ from $U \wedge X1$.

If the variables of $d1$ and $d2$ are mutually exclusive, and $U \Rightarrow (Z2 \Rightarrow X1)$ then $Z2$ detects $X1 \wedge X2$ in $d1; d2$ from U.

Proof. *Safeness* of $d1; d2$ follows from *Safeness* of $d2$ and from $U \Rightarrow (Z2 \Rightarrow X1)$. The proof of *Progress* of $d1; d2$ is identical to that of $d1 \| d2$. Since the variables of $d1$ and $d2$ are disjoint, execution of $d1$ does not falsify $Z2$. And, from *Stability* of $d2$, execution of $d2$ falsifies $Z2$ only if it falsifies $X2$. Thus, *Stability* of $d1; d2$ is satisfied.

$\square$

Large detectors can be designed by repetitive application of parallel and/or sequential composition. Conceptually speaking, in parallel composition, the composed detectors execute concurrently, whereas in sequential composition, the composed detectors effectively execute one after another. Therefore, the time required to complete the detection is expected to be higher in the sequential composition. This extra time may be warranted when the witness predicate of $d1; d2$, *viz* $Z2$, can be witnessed atomically although the witness predicate of $d1 \| d2$, *viz* $Z1 \wedge Z2$, cannot. Also, in sequential composition, the detector $d2$ can assume that $d1$ has completed its detection, i.e., $Z1$ and $X1$ are true. Hence, the design of $d2$ itself may be simplified.

3.3 Sufficiency of Detectors for Fail-Safe Fault-Tolerance

In this section, first, we show (1) detectors are sufficient for satisfying safety specifications, and (2) fail-safe tolerant detectors are sufficient for designing fail-safe tolerant programs. Towards proving (1) and (2), first, we show in Lemma 3.5 that if two prefixes of a computation maintain a safety specification then so does their concatenation. Then, we show in Lemma 3.6 that the violation of a safety specification can be detected from the current state, independent of how that state is reached. Subsequently, we show in Theorem 3.7 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 3.7, we show (1) and (2) in Theorems 3.8 and 3.9, respectively.

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 3.5

If $\quad \sigma s$ maintains $SPEC \quad$ and $\quad s\beta$ maintains $SPEC$

then $\quad \sigma s \beta$ 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 $SPEC$ $\square$

Lemma 3.6

If $\quad \sigma s$ maintains $SPEC$

then $\quad \sigma s s'$ maintains $SPEC$ iff ss' maintains $SPEC$.

Proof. *If part:*

$\sigma s s'$ maintains $SPEC$

$= \{$ by definition of maintains $\}$

$(\exists \beta : \sigma s s' \beta \in SPEC)$

$\Rightarrow \{$ by suffix closure of $SPEC$ $\}$

$(\exists \beta : s s' \beta \in SPEC)$

$= \{$ by definition of maintains $\}$

ss' maintains $SPEC$

Only if part:

ss' maintains $SPEC$ $\wedge$ σs maintains $SPEC$

$\Rightarrow \{$ by Lemma 3.5 $\}$

$\sigma s s'$ maintains $SPEC$ $\square$

Theorem 3.7 For each action ac of p there exists a predicate such that execution of ac maintains $SPEC$ iff it executes in a state where that state predicate is true.

Proof. Consider a prefix of a computation, say σs, that maintains $SPEC$. Execution of ac maintains $SPEC$ iff the extended prefix $\sigma s s'$, 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 3.6, 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 to be a witness for the theorem. $\square$

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 $SPEC$. $\square$

Note that the existence of detection predicates follows from Theorem 3.7, 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 detection predicate for each action.

Let q be a program. To ensure that q refines $SSPEC$, for each action ac of q, we need to add a detector whose detection predicate is a detection predicate of ac. Moreover, we must ensure that the added detectors and q do not interfere with each other. If the added detectors and q do not interfere with each other then the resulting program, say p, must itself refine the detector specification. We state this in Theorem 3.8: if p refines the detector specification for each action of p then it refines $SSPEC$.

Theorem 3.8 (*Detectors are sufficient for satisfying safety specifications.*)

If

- ($\forall$ actions $g \longrightarrow st$ of p :
  ($\exists Z, X : g \Rightarrow Z$,
  Z detects X in p from S, and
  X is a detection predicate of $g \longrightarrow st$ for $SPEC$))

then

- p refines $SSPEC$ from S

Proof. From the definition of detection predicates, it follows that whenever $g \longrightarrow st$ is executed, the extended prefix maintains $SPEC$. Therefore, p refines $SSPEC$ from S. $\square$

Once again, if we add detectors to a fault-intolerant program in order to achieve fail-safe F-tolerance, these detectors and the fault-intolerant program should not interfere with each other. In other words, the resulting program must itself refine the specification in the absence of faults and also refine the fail-safe F-tolerance specification. With this intuition, we write Theorem 3.9 as follows:

Theorem 3.9 (*Fail-safe tolerant detectors are sufficient for fail-safe tolerant programs.*)

If

- p refines $SPEC$ from S, and
- $(\exists\ T : T$ is an F-span of p from S, and
  $(\forall$ actions $g \longrightarrow st$ of p :
  $(\exists Z, X : g \Rightarrow Z$,
  X is a detection predicate of $g \longrightarrow st$ for $SPEC$, and
  p is fail-safe F-tolerant for 'Z detects X' from T)))

then

- p is fail-safe F-tolerant for $SPEC$ from S

Proof. Once again, from the definition of detection predicates, it follows that whenever $g \longrightarrow st$ is executed in states where T is true, the extended prefix maintains $SSPEC$. Therefore p refines $SSPEC$ from T. Also, p refines $SPEC$ from S. It follows that p is fail-safe F-tolerant for $SPEC$ from S. $\Box$

In closing, we note that detectors are also used in nonmasking tolerant programs to detect whether the program has been perturbed to an unintended state. They are also used in masking tolerant programs for ensuring the safety specification. These applications will be discussed in Chapters 4 and 5.

3.3.1 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, for any *addr*, at most one object of the form $\langle addr, value\rangle$

is in MEM, and if MEM does not contain an object of the form $\langle addr, -\rangle$ then $(val | \langle addr, val\rangle \in MEM)$ returns an arbitrary value):

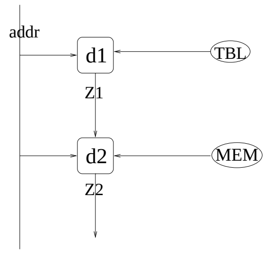
$$p :: \quad true \quad \longrightarrow \quad data := (val | \langle addr, val\rangle \in MEM)$$

Let $SPEC_{mem}$ be the specification of the memory transfer program; intuitively, $SPEC_{mem}$ requires that the *data* is eventually set to the correct value and it is never set to an incorrect value.

Now, consider two fault-classes: The first is a protection fault whereby *addr* is (initially) corrupted so that it falls outside the valid address space, and the second is a page fault whereby *addr* and its value are (initially) removed from the memory.

We use two detectors $d1$ and $d2$: $d1$ detects whether *addr* is valid, and $d2$ detects whether *addr* is in the memory. The detection predicates of these detectors is $X1$ and $X2$ respectively, and their witness predicates are $Z1$ and $Z2$ respectively (cf. Figure 3.1). Formally, the actions of $d1$ and $d2$ are as follows (where TBL is the set of valid addresses):

$$d1 :: \quad addr \in TBL \ \wedge \ \neg Z1 \quad \longrightarrow \quad Z1 := true$$
$$d2 :: \quad (\exists\ val :: \langle addr, val\rangle \in MEM) \ \wedge \ \neg Z2 \quad \longrightarrow \quad Z2 := true$$



$$X1 \equiv addr \in TBL$$
$$X2 \equiv (\exists val :: \langle addr, val\rangle \in MEM)$$
$$U1 \equiv (Z2 \Rightarrow X1) \ \wedge \ (Z1 \Rightarrow X1) \ \wedge \ (Z2 \Rightarrow X2)$$

Figure 3.1: Memory access

To add fail-safe tolerance to the protection fault (the page fault), it suffices to restrict p so that it executes only after $Z1$ (respectively $Z2$) is truthified. After such a restriction, program $d1; p$ is fail-safe 'protection fault'-tolerant in that in the absence of faults, p refines $SPEC_{mem}$ and in the presence of the protection fault, p refines the fail-safe tolerance specification of $SPEC_{mem}$. More specifically, if no fault occurs, i.e., an object $\langle addr, val\rangle$ exists in the memory, $d1; p$ eventually sets $data$ to be equal to val , and it never sets $data$ to any other value. In the presence of a protection fault, it never sets the value of $data$ incorrectly, although, it may not assign a value to $data$.

We can also compose $d1$ and $d2$ in parallel or in sequence. If $d1$ and $d2$ are composed in parallel (respectively in sequence), then to add fail-safe tolerance to both the protection fault and the page fault, it suffices to restrict p so that p executes only after $Z1 \wedge Z2$ (respectively $Z2$) is truthified.

In other words, we have

$d1; p$ is fail-safe 'protection fault'-tolerant from $U1 \wedge X1 \wedge X2$.

$d2; p$ is fail-safe 'page fault'-tolerant from $U1 \wedge X1 \wedge X2$.

$d1; d2; p$ is fail-safe 'page fault'-tolerant and fail-safe 'protection fault' tolerant from $U1 \wedge X1 \wedge X2$.

$(d1 \| d2); p$ is fail-safe 'page fault'-tolerant and fail-safe 'protection fault' tolerant from $U1 \wedge X1 \wedge X2$.

Remark. Observe that in sequential composition of $d1$ and $d2$, $d2$ can be implemented assuming that the given address is valid, although such an assumption cannot be made in the parallel composition. Also, in sequential composition, it is sufficient to restrict p with $Z2$ whereas, in parallel composition, it is necessary to restrict p with $Z1 \wedge Z2$.

3.4 Necessity of Detectors for Fail-Safe Fault-Tolerance

In order to show that detectors are necessary in the design of fail-safe tolerance, we show that every fault-tolerant program designed using encapsulation and refinement from a fault-intolerant program contains detectors. More specifically, we show that (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 uses Lemmas 3.5, 3.6 and Theorem 3.7. Recall that Theorem 3.7 shows that for each action of the program there exists a set of states from where its execution maintains the safety specification. Using the definition of detection predicates from Theorem 3.7, we show (1) and (2) in Theorems 3.10 and 3.12. 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 3.10, 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 3.10 we show that p' itself refines the specification of a detector.

Theorem 3.10 (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 detection 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. $\square$

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 3.10 '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' satisfies Safeness and Stability. Thus, we have

Lemma 3.11

If $\quad p'$ encapsulates p, and p' refines $SSPEC$ from S

then $\quad (\forall\ ac : ac$ is an action of p :

$\qquad\qquad p'$ 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 3.10, 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 3.10 can be used, verbatim, to prove that p' satisfies Safeness and Stability. □

We now use Theorem 3.10 and Lemma 3.11 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 3.12 (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 $\mathbf{F}$
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 detection 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 3.10, 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 3.10 . □

3.4.1 Example : Memory Access (continued)

Returning to the memory access example in Section 3.3.1, observe that program $d1; p$ is fail-safe 'protection-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 protection fault. We use the theory of detectors developed in the previous subsection to show that $d1; p$ is fail-safe 'protection-fault'-tolerant.

Let $S := U1 \wedge X1 \wedge X2$, $T := U1 \wedge X2$, and $F :=$ 'protection fault' (cf. Figure 3.1). Now, observe that p refines $SPEC_{mem}$ from S, $d1; p$ refines p from S, $d1; p$ encapsulates p, and $(d1; p) [\!] F$ refines the safety specification of $SPEC_{mem}$ from T. Therefore, by Theorem 3.12, we have

$d1; p$ is fail-safe 'protection fault'-tolerant for $SPEC_{mem}$ from S, and

$d1; p$ is a fail-safe 'protection fault'-tolerant detector of a detection predicate of p .

Likewise, we have

$d2; p$ is fail-safe 'page fault'-tolerant for $SPEC_{mem}$ from S, and

$d2; p$ is a fail-safe 'page fault'-tolerant detector of a detection predicate of p .

Note that the detection predicate of $d1; p$ is $X1$ and the witness predicate of $d1; p$ is $Z1$. Also, this detector is implemented by action $d1$ in program $d1; p$. Likewise, the detection predicate of $d2; p$ is $X2$ and the witness predicate of $d2; p$ is $Z2$, and the detector is implemented by action $d2$.

3.5 Composition of Detectors and Programs

In this section, we discuss how a detector component is correctly added to a program so that the resulting program satisfies the specification of the component. As far as possible, the proof of preservation should be simpler than explicitly proving all over again that the specification is satisfied in the resulting program. This is achieved by a compositional proof that shows that the program does not "interfere" with the component, i.e., the program and the component when executed concurrently do not violate the specification of the component.

Compositional proofs of interference-freedom have received substantial attention in the formal methods community [1, 18, 48, 52, 53] in the last two decades. Drawing from these efforts, we identify several simple sufficient conditions to ensure that when a program p is composed with a detector q, the safety specification of q, *viz* *Safeness* and *Stability*, and liveness specification, *viz* *Progress* and *Convergence*, are not violated.

Sufficient conditions for satisfying the safety specification of a detector. To demonstrate that p does not interfere with *Safeness* and *Stability*, a straightforward sufficient condition is that the actions of p be a subset of the actions of q; this occurs, for instance, when program itself acts as a detector. Another straightforward condition is that the variables of p and q be disjoint. A more general condition is that p only reads (but not writes) the variables of q; in this case, p is said to be "superposed" on q.

Sufficient conditions for satisfying the liveness specification of a detector. The three conditions given above also suffice to demonstrate that p does not interfere with *Progress* of q, provided that the actions of p and q are executed fairly. Yet another condition for satisfying *Progress* of q is to require that q be "atomic", i.e., that q achieves its *Progress* in at most one step. It follows that even if p and q execute concurrently, *Progress* of q is satisfied.

Alternatively, require that p executes only after *Progress* of q is achieved. It follows that p cannot interfere with *Progress* of q. Likewise, require that p terminates eventually. It follows that after p has terminated, execution of q in isolation satisfies its *Progress*.

More generally, require that there exists a variant function f (whose range is over a well-founded set) such that execution of any action in p or q reduces the value of f until *Progress* of q is achieved. It follows that even if q is executed concurrently with p, *Progress* of q is satisfied.

The sufficient conditions outlined above are formally stated in Table 3.1.

The discussion above has addressed how to prove that a program does not interfere with a component, but not how a component does not interfere with a program. Standard compositional techniques suffice for this purpose. In practice, detectors such as snapshot procedures, watchdog programs, and snooper programs typically read but not write the state of the program to which they are added. Thus, these detectors do not interfere with the program.

3.6 Chapter Summary

In this chapter, we identified a fault-tolerance component that is necessary and sufficient for the design of fail-safe fault-tolerance. Using our assumption that specifications are fusion-closed and suffix-closed, we showed that for each action ac in the fault-intolerant program, there exists a detection predicate of ac such that the execution of ac satisfies the safety specification only if it executes in a state where its detection predicate is true. Subsequently, we showed that if a fail-safe fault-tolerant program is designed by encapsulation and refinement from a fault-intolerant program then it contains fail-safe F-tolerant detectors. Moreover, such fail-safe fault-tolerant programs can be designed by adding detectors to a fault-intolerant program.

Intuitively, encapsulation and refinement restrict the program transformation in such a way that a fault-tolerant program is not designed from scratch, i.e., by ignoring the fault-intolerant program. In this sense, it is a very non-restrictive condition, and program transformations by existing methods satisfy this condition. Therefore, programs designed using these methods implicitly contain detectors.

Finally, given a fault-intolerant program p and a fault-class F, detectors are used to design a fail-safe F-tolerant program as follows: For each action, ac, of p, we identify a detection predicate, sf, for that action. Then, we design a detector whose detection predicate is detection predicate of sf. Depending upon the structure of sf, this detector is designed in an hierarchical manner using smaller detectors. Subsequently, action ac is restricted to execute only in states where the witness predicate is true.

In the following theorems, it is given that Z detects X in q from U and U is closed in p.

Theorem 3.13 (Superposition)

If q does not read or write any variable written by p, and
p only reads the variables written by q
then Z detects X in $q \| p$ from U.

Theorem 3.14 (Containment)

If actions of p are a subset of q
then Z detects X in $q \| p$ from U.

Theorem 3.15 (Atomicity)

If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$, and q is atomic
then Z detects X in $q \| p$ from U .

Theorem 3.16 (Order of execution)

If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$
then Z detects X in $q ; p$ from U.

Theorem 3.17 (Termination)

If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$, and U converges to V in $p \| q$
then Z detects X in $(\neg V \wedge p) \| q$ from U.

Theorem 3.18 (Variant function)

If $\{U \wedge (0 < f = K)\}\ q\ \{(0 < f <= K-1) \vee Z \vee \neg X\}$
$\{U \wedge (0 < f = K)\}\ p\ \{(0 < f <= K-1) \vee Z \vee \neg X\}$, and
$\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$
then Z detects X in $q \| p$ from U.

Table 3.1: Sufficient conditions for interference-freedom of a detector and a program

CHAPTER 4

CORRECTORS : A BASIS OF NONMASKING FAULT-TOLERANCE

In this chapter, we identify the second of the two fault-tolerance components, *corrector*, that forms a basis of nonmasking fault-tolerance. Towards showing that correctors form a basis of nonmasking fault-tolerance, we show that (1) correctors are sufficient to design nonmasking fault-tolerant programs, i.e., a rich class of nonmasking fault-tolerant programs can be designed using correctors, and (2) correctors are necessary to design nonmasking fault-tolerance, i.e., a rich class of nonmasking fault-tolerant programs contain correctors. We also note that the commonly used techniques in designing nonmasking fault-tolerant programs such as voters, error correction codes, reset procedures, rollback recovery, rollforward recovery, constraint (re)satisfaction, exception handlers, and alternate procedures in recovery blocks are instances of correctors.

We proceed as follows: First, we give a formal definition of a corrector (cf. Section 4.1). Then, we show how correctors can be constructed in an hierarchical and efficient manner (cf. Section 4.2). Subsequently, we show that correctors are sufficient and necessary in the design of nonmasking fault-tolerance (cf. Sections 4.3 and 4.4). Finally, we discuss the sufficiency conditions that ensure that when a corrector is added to a program the corrector and the program do not interfere with each other (cf. Section 4.5).

4.1 Definition

Definition (*corrects*). Let X and Z be state predicates. Let 'Z corrects X' be the problem specification that is the set of all state sequences, $s_0, s_1, ...$ that satisfy the following four 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, $Z \Rightarrow X$ 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, $(\{Z\}, \{Z \vee \neg X\})$.) □

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

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

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 [8]. 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.)*

4.1.1 Properties of the *corrects* relation.

If Z corrects X in c from U, then Z detects X in c from U. Also, the corrects relation is antisymmetric and transitive in its first two arguments:

Lemma 4.1 Let X, Y, and Z be state predicates of c and U be a state predicate that is closed in c. The following statements hold.

- If Z corrects X in c from U, and X corrects Z in c from U
  then $U \Rightarrow (Z \equiv X)$.

- If Z corrects Y in c from U, and Y corrects X in c from U
  then Z corrects X in c from U . $\square$

Lemma 4.2 Let V be a state predicate such that $U \wedge V$ is closed in c. The following statements hold.

- If $\quad Z$ corrects X in c from U

  then $\quad Z$ corrects X in c from $U \wedge V$.

- If $\quad Z$ corrects X in c from U and $V \Rightarrow X$

  then $\quad Z \vee V$ corrects X in c from U . $\square$

4.2 Hierarchical and Distributed Design of Correctors

Just as in case of detectors, it is possible to design correctors in an hierarchical and distributed manner. In this section, we discuss how a corrector with correction predicate $X1 \wedge X2$ can be designed by first designing a corrector with correction predicate $X1$, a corrector with correction predicate $X2$, and composing them in parallel or in sequence.

In the rest of the chapter, we will implicitly assume that the problem specification of a corrector c (cn) is 'Z corrects X in c from U' (respectively, 'Zn corrects Xn in cn from Un'). Also, we will implicitly assume that U (Un) is closed in c (respectively, cn).

Parallel composition of correctors. In the parallel composition of $c1$ and $c2$, denoted by $c1 \| c2$, both $c1$ and $c2$ execute concurrently. Formally, the parallel composition of $c1$ and $c2$ is the union of the (variables and actions of) programs $c1$ and $c2$.

Observe that, just as in case of detectors, '$\|$' is commutative ($c1 \| c2 = c2 \| c1$), associative ($(c1 \| c2) \| c3 = c1 \| (c2 \| c3)$), and that '$\wedge$' distributes over '$\|$' ($g \wedge (c1 \| c2) = (g \wedge c1) \| (g \wedge c2)$).

Theorem 4.3 Given $Z1$ corrects $X1$ in $c1$ from U and $Z2$ corrects $X2$ in $c2$ from U.

If the variables of $c1$ and $c2$ are mutually exclusive

then $Z1 \wedge Z2$ corrects $X1 \wedge X2$ in $c1 \| c2$ from U . $\square$

Sequential composition of correctors. In the sequential composition of $c1$ and $c2$, denoted by $c1; c2$, $c2$ executes only after $c1$ has completed its correction, i.e., after the witness predicate $Z1$ is truthified. Formally, the sequential composition of $c1$ and $c2$ is the program whose set of variables is the union of the variables of $c1$ and $c2$ and whose set of actions is the union of the actions of $c1$ and of $Z1 \wedge c2$. We postulate the axiom that ';' is left-associative ($c1; c2; c3 = (c1; c2); c3$)).

Observe that ';' is not commutative, that ';' distributes over '$\|$', and that '$\wedge$' distributes over ';'.

Theorem 4.4 Given $Z1$ corrects $X1$ in $c1$ from U and $Z2$ corrects $X2$ in $c2$ from $U \wedge X1$.

If the variables of $c1$ and $c2$ are mutually exclusive, and $U \Rightarrow (Z2 \Rightarrow X1)$

then $Z2$ corrects $X1 \wedge X2$ in $c1; c2$ from U . $\square$

Correctors are also designed by composing correctors with detectors. For example, one way to design a corrector with correction predicate X is by sequentially composing a detector and a corrector; first the detector detects that X is false and, then the corrector then truthifies X. Another way is by sequentially composing of a corrector and a detector: first the corrector truthifies X and, then the detector truthifies the desired witness predicate Z.

Theorem 4.5 Given Z detects $\neg X$ in d from U, $Z \Rightarrow Z'$, and X corrects X in c from $U \wedge Z'$.

If X is closed in d, and $\{U \wedge Z\}\ c\ \{Z \vee X\}$

then X corrects X in $(\neg Z \wedge d); c$ from U. □

Theorem 4.6 Given X corrects X in c from U and Z detects X in d from U.

If X is closed in d

then Z corrects X in $(\neg X \wedge c); d$ from U . □

4.3 Sufficiency of Correctors for Nonmasking Fault-Tolerance

In this section, we use the definition of nonmasking fault-tolerance to show in Theorem 4.7 that nonmasking correctors are sufficient for the design of nonmasking tolerant programs.

Theorem 4.7 (*Nonmasking tolerant correctors are sufficient for nonmasking tolerance.*)

If

- p refines $SPEC$ from S, and
- $(\exists\ T : T$ is an F-span of p for $SPEC$ from S, and
  p is nonmasking F-tolerant for 'S corrects S' from $T)$

then

- p is nonmasking tolerant for $SPEC$ from S.

Proof. Follows from the definition of nonmasking tolerance. □

4.3.1 Example : Memory Access (continued)

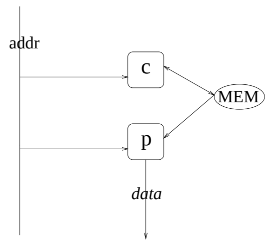
Continuing with the example in Section 3.3.1, consider the case where the given address is valid but is not in the memory. In this case, an object of the form $\langle addr, -\rangle$

has to be added to the memory (cf. Figure 4.1). To this end, we use a corrector c. (The corrector may, for instance, obtain the object from a disk, from a remote memory, or from a network; but we ignore these details.) Formally, the corrector c is specified as follows:

$$c :: \quad \neg(\exists\ val :: \langle addr, val\rangle \in MEM) \quad \longrightarrow \quad MEM := MEM \cup \{\langle addr, -\rangle\}$$

Thus, we may observe:

$c \| p$ is nonmasking 'page fault'-tolerant from $U1\ \wedge X1 \wedge X2$.



$X1 \equiv addr \in TBL$
$X2 \equiv (\exists val :: \langle addr, val\rangle \in MEM)$
$U1 \equiv (Z2 \Rightarrow X1)\ \wedge\ (Z1 \Rightarrow X1)\ \wedge\ (Z2 \Rightarrow X2)$

Figure 4.1: Memory access (continued)

4.4 Necessity of Correctors for Nonmasking Fault-Tolerance

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 4.8, 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 of p . As discussed in Section 3.4, we prove Theorem 4.8 by showing that p' itself refines the required corrector specification.

Theorem 4.8 (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 of p .

Proof.

Let $X = S$, and

$$Z = S \wedge \{s : s \text{ is a state of } p' : \quad s \text{ is reached in some computation of } p' \text{ starting from } T \} .$$

Since p refines $SPEC$ from S, it follows that X is an invariant 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. $\square$

The next lemma generalizes Theorem 4.8 . In general, given a program p that refines $SPEC$ from S, p' may not refine p from 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 4.9 shows that in such a case, p' contains a nonmasking corrector of an invariant of p . (The corrector is nonmasking in that the correction predicate is preserved only after p' reaches a state where R is true.)

Lemma 4.9

If $\quad p$ refines $SPEC$ from S,

$\quad\quad p'$ refines p from R, where $R \Rightarrow S$, and

$\quad\quad p'$ refines $(true)^*(p' \mid R)$ from T

then $\quad p'$ is a nonmasking corrector of an invariant 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 specification 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. $\square$

We now use Lemma 4.9 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 4.10 (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 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, R is an invariant of p'.

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 for SPEC from R .

Part 2: corrector. We use the definition of Z and X given in the proof of Lemma 4.9 and show that p' is nonmasking F-tolerant for 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 4.9, 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 4.9, 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, eventually faults stop (at least for long enough time for the system to recover)), p' eventually reaches a state where R is true. And, from this state, p' refines Z corrects X . Thus, p' is nonmasking F-tolerant for Z detects X from R. □

4.4.1 Example : Memory Access (continued)

Continuing with the memory access example in Section 4.3.1, observe that program $c [\!] p$ 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 $c\|p$ is nonmasking 'page fault'-tolerant.

Let $S := U1 \wedge X1 \wedge X2$, $T := U1 \wedge X1$, and $F :=$ page fault (cf. Figure 4.1). Now, observe that p refines $SPEC_{mem}$ from S, $c\|p$ refines p from S, and $c\|p\|F$ refines $(true)^*((c\|p) \mid S)$ from T. Therefore, by Theorem 4.10, we have

$c\|p$ is nonmasking 'page fault'-tolerant for $SPEC_{mem}$ from S, and

$c\|p$ is a nonmasking 'page fault'-tolerant corrector of an invariant of p .

The reader will notice this time that the correction and witness predicate of $c\|p$ is $X2$ and the corrector is implemented by action c .

4.5 Composing Correctors with Programs

When a fault-intolerant program and a corrector are composed together, they may access a shared memory or execute concurrently. Therefore, we need to ensure that they do not interfere with each other.

For instance, when a corrector c is added to program p, we need to ensure that they do not interfere with each other. The sufficient conditions to ensure that c is not interfered by p are similar to those in Table 3.1. To show that p is not interfered by c, standard compositional techniques suffice. An alternative technique to show that p is not interfered by c is to show that c executes only after occurrence of faults. For example, correctors such as reset, rollback recovery, and forward recovery procedures are typically restricted to execute only in states where the program is perturbed by faults. Thus, these correctors do not interfere with the program.

4.6 Chapter Summary

In this chapter, we identified a fault-tolerance component that is necessary and sufficient for the design of nonmasking fault-tolerance. We showed that if a nonmasking fault-tolerant program is designed using refinement from a fault-intolerant program then it contains correctors. Moreover, such nonmasking fault-tolerant programs can be designed by adding correctors to a fault-intolerant program. Once again, as discussed in Chapter 3, the refinement condition is non-restrictive.

Finally, given a fault-intolerant program p and a fault-class F, correctors are used to design a nonmasking F-tolerant program as follows: For program p, we identify an invariant of p, say S. Since S is an invariant of p, every computation of p that starts in a state where S is true satisfies the specification at hand. It follows that to add nonmasking fault-tolerance, we need to add a corrector whose correction predicate is S.

CHAPTER 5

DETECTORS AND CORRECTORS : A BASIS OF MASKING FAULT-TOLERANCE

In this chapter, we consider a basis set of components for designing masking fault-tolerance. It turns out that to design masking fault-tolerance, we do not need additional types of components; detectors and correctors discussed in last two chapters form a basis of designing masking fault-tolerance. This relation between masking fault-tolerance, fail-safe fault-tolerance and nonmasking fault-tolerance raises an interesting question: is it possible to design masking fault-tolerance *via* nonmasking (or fail-safe) fault-tolerance? In this chapter, we address this question as well.

This chapter is organized as follows: First, we show that detectors and correctors together are sufficient for designing masking fault-tolerance (cf. Section 5.1). Then, we show that these components are also necessary in the design of masking fault-tolerance (cf. Section 5.2). Finally, we present method for designing masking fault-tolerance *via* nonmasking fault-tolerance and illustrate this method with a simple data-transfer program (cf. Section 5.3).

5.1 Sufficiency of Detectors and Correctors for Masking Fault-Tolerance

In order to prove that masking tolerant detectors and correctors are sufficient for the design of masking tolerant programs, we show, in Lemma 5.1, that if a program computation has a prefix that maintains $SPEC$ and whose corresponding suffix is in $SPEC$, then that program computation is also in $SPEC$. Using this lemma, we show the sufficiency of detectors and correctors in Theorem 5.2.

Lemma 5.1

If $\quad \alpha s$ maintains $SPEC$ and $s\beta \in SPEC$

then $\quad \alpha s \beta \in SPEC$.

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 $SPEC$ }

$\alpha s \beta \in SPEC$ $\square$

Theorem 5.2 (*Masking tolerant detectors and correctors are sufficient for masking tolerant programs.*)

If

- p refines $SPEC$ from S, and
- $(\exists\ S, T$:
  T is an F-span of p for $SPEC$ from S,
  p is masking F-tolerant for 'S corrects S' from T, and
  ($\forall$ actions $g \longrightarrow st$ of p :
  $(\exists Z, X$:
  $g \Rightarrow Z$,
  X is a detection predicate of $g \longrightarrow st$ for $SPEC$, and
  p is masking F-tolerant for 'Z detects X' from T)))

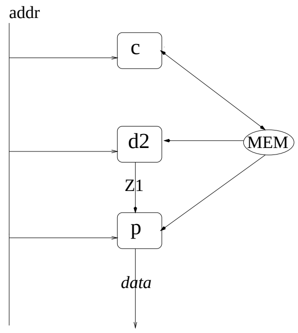
then

- $(\exists S$: p is masking F-tolerant for $SPEC$ from S) .

Proof. Let S_2 and T_2 be the state predicates that satisfy the antecedent. We show that $S := T_2$ satisfies the consequent. Observe, from the definition of F-span, that T_2 is closed in p and in F. Now, consider a computation of p, σ, that starts in a state where T_2 is true: σ is of the form $\alpha s \beta$, where s is a state where S_1 is true. It follows that $s\beta$ is in $SPEC$. Also, in any state where T is true, an action is executed only when its detection predicate holds. Therefore, αs maintains $SPEC$. Therefore, from Lemma 5.1, σ is in $SPEC$. In other words, T_2 is closed in p and in F, and every computation of p that starts from a state where T_2 is true is in $SPEC$, i.e., $S := T_2$ satisfies the consequent. □

5.1.1 Example : Memory Access (continued)

Continuing with the memory access example in Section 4.3.1, masking tolerance to page fault can be designed by adding the detector $d2$, corrector c and restricting program p so that it executes only after $Z2$ is truthified.



$$X1 \equiv addr \in TBL$$
$$X2 \equiv (\exists val :: \langle addr, val \rangle \in MEM)$$
$$U1 \equiv (Z2 \Rightarrow X1) \;\wedge\; (Z1 \Rightarrow X1) \;\wedge\; (Z2 \Rightarrow X2)$$

Figure 5.1: Memory access (continued)

Thus, we may observe:

$c [\!] (d2; p)$ is masking 'page fault'-tolerant from $U1 \wedge X1 \wedge X2$.

5.2 Necessity of Detectors and Correctors for Masking Fault-Tolerance

We show the necessity of detectors and correctors in the design of masking fault-tolerance in Theorem 5.5 by proving that both detectors and correctors exist in masking F-tolerant programs. As a first step towards proving 5.5, in Theorem 5.3, we combine Theorem 3.10 and Theorem 4.8 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 of p .

Proof. The proof follows from Theorem 3.10 and Theorem 4.8. □

We generalize Theorem 5.3, as we did Theorem 4.8, 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 of p .

Proof *Part 1: detector.* We use the definition of Z and X as in Theorem 3.10. From Theorem 3.11, 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 3.10, 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 p 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 of p , and
- p' is a nonmasking F-tolerant corrector of an invariant 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 for $SPEC$ from T.

Part 2: *detector.* We use the definition of Z and X in Theorem 3.12 . Theorem 3.12 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 for 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 for 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 the specification Z corrects X . In other words, $p' \| F$ refines the nonmasking tolerance specification of Z corrects X from T . $\square$

5.2.1 Example : Memory Access (continued)

Continuing with the example in Section 4.4.1, we use Theorem 5.5 to show that program $c \| (d2; p)$ is masking 'page fault'-tolerant.

Program $c \| (d2; p)$ consists of three actions: one action adds a tuple $\langle addr, - \rangle$ if such a tuple does not exist in the memory. Another action detects if a tuple of the form $\langle addr, - \rangle$ exists in the memory. If this detection succeeds, it sets $Z2$ to true. Finally, the data is set after $Z2$ is set to true.

Let $S := U1 \wedge X1 \wedge X2$, $T := U1 \wedge X1$, and $F :=$ page fault (cf. Figure 5.1). Observe that $c \| p$ refines $SPEC_{mem}$ from S, $c \| (d2; p)$ refines p from S, $(c \| (d2; p)) \| F$ refines $(true)^*((c \| p) | S)$ from T, $c \| (d2; p)$ encapsulates $c \| p$, and $c \| (d2; p) \| F$ refines the safety specification of $SPEC_{mem}$ from T . Therefore, by Theorem 5.5, we have:

$c \| (d2; p)$ is masking 'page fault'-tolerant for $SPEC_{mem}$ from S,

$c \| (d2; p)$ is a masking 'page fault'-tolerant detector of a detection predicate of p (respectively c),

$c \| (d2; p)$ is a masking 'page fault'-tolerant corrector of an invariant of $c \| p$.

5.3 Designing Masking Fault-Tolerance *via* Nonmasking Fault-Tolerance

Figure 5.2 summarizes the necessity and sufficiency of detectors and correctors in fault-tolerant programs. Note that in the design of masking fault-tolerance, we need to add both detectors and correctors. These detectors and correctors themselves can be added in a stepwise fashion. Specifically, it is possible to first add correctors to obtain nonmasking tolerance and then add detectors to obtain masking fault-tolerance. Likewise, it is possible to design masking fault-tolerance *via* fail-safe fault-tolerance where we first add detectors and then correctors. In this section, we explore how masking fault-tolerance can be designed via nonmasking fault-tolerance. Similar design is also possible via fail-safe fault-tolerance. An example of such a design is discussed in Section 7.1.

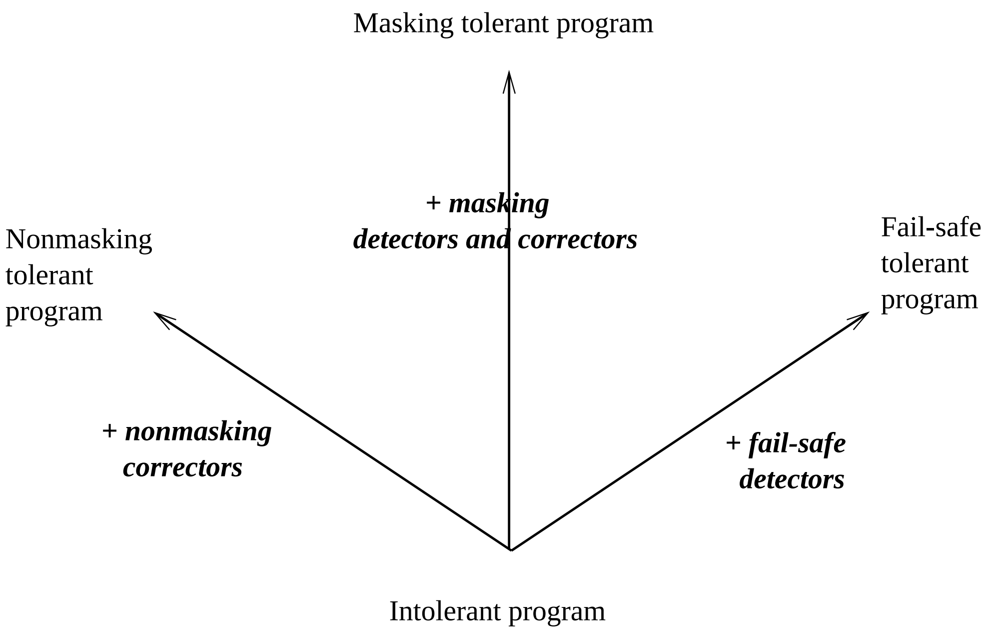


Figure 5.2: Necessity and Sufficiency of Detectors and Correctors

The design of masking fault-tolerance *via* nonmasking fault-tolerance is based on the following Theorem:

Theorem 5.6

If $\quad p$ refines $SPEC$ from S,

$\quad\quad p$ refines $SSPEC$ from T, where $T \Leftarrow S$, and

$\quad\quad p$ refines $(true)^*(p \mid S)$ from T

then $\quad p$ 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 . $\square$

Theorem 5.6 suggests that an intolerant program can be made masking fault-tolerant in two stages: In the first stage, the intolerant program is transformed into one that is nonmasking fault-tolerant for, say, the invariant S_{np} and the fault-span T_{np}. In the second stage, the tolerance of resulting program is enhanced from nonmasking to masking by transforming the nonmasking fault-tolerant program so that every computation upon starting from a state where T_{np} holds, in addition to eventually reaching a state where S_{np} holds, also satisfies the safety specification of the problem at hand. We address the details of both stages, next.

Stage 1. For a fault-intolerant program, say p, the problem specification is satisfied by computations of p that start at a state where the invariant of p, say S_p, is true but not necessarily by those that start at states where S_p is false. As discussed in Chapter 4, to add nonmasking tolerance to p, we need to add a corrector whose correction predicate is S_p.

Note that when a corrector is added to p, the resulting nonmasking fault-tolerant program, np, may have new variables and actions. Therefore, the invariant, S_{np}, and the fault-span, T_{np}, of the resulting nonmasking fault-tolerant program may be different.

Stage 2. For a nonmasking program, say np, the problem specification is satisfied *after* computations of np converge to states where S_{np} is true. However, the safety specification may not be satisfied in all computations of np that start at states where T_{np} is true. From Theorem 5.6, it follows that to obtain a masking fault-tolerant program, we need to transform np so that in addition to maintaining convergence to S_{np}, it also guarantees that its safety specification is satisfied.

As discussed in Chapter 3, to transform np into a masking fault-tolerant program, for each action ac of np, we need to add a detector whose detection predicate is a detection predicate ac. In our experience, we have observed that it is often necessary to add a detector for some limited set of actions as the detection predicate of other program actions is trivially *true*. This observation follows from the fact that the actions of masking tolerant programs can be conceptually characterized as either "critical" or "noncritical", with respect to the safety specification. Critical actions are those actions whose execution in the presence of faults can violate the safety specification; hence, only they require non-trivial detection predicates. In other words, the detection predicate of all non-critical actions is merely *true*.

For example, in terminating programs, e.g. feed-forward circuits or database transactions, only the actions that produce an output or commit a result are critical. In reactive programs, e.g. operating systems or plant controllers, only the actions that control progress while maintaining safety are critical. In the rich class of "total" programs [63] for distributed systems, e.g. distributed consensus [17,64], garbage collection [49], global function computation, distributed reset [9, 38], snapshot [21, 61], and termination detection [28,44], only the "decider" actions that declare the outcome of the computation are critical.

Verification obligations. The addition of corrector and detector components as described above may add variables and actions to an intolerant program and, hence, the invariant and the fault-span of the resulting program may be different from those of the original program. The addition of corrector and detector components thus creates some verification obligations for the designer.

Specifically, when a corrector is added to an intolerant program, the designer has to ensure that the corrector actions and the intolerant program actions do not interfere with each other. That is, even if the corrector and the fault-intolerant program execute concurrently, both accomplish their tasks: The corrector restores the intolerant program to a state from where the problem specification of the intolerant program is (re)satisfied. And starting from such a state, the intolerant program satisfies its problem specification.

Similar obligations are created when detectors are added to a nonmasking program. Even if the detectors and the nonmasking program are executed concurrently, the designer has to ensure that the detector components and the components of the nonmasking program all accomplish their respective tasks.

Another set of verification obligations is due to the fact that the corrector and detector components are themselves subject to the faults that the intolerant program is subject to. Hence, the designer is obliged to show that these components accomplish their task in spite of faults. More precisely, as discussed in Chapter 4, the corrector tolerates the faults by ensuring that when fault actions stop executing it eventually restores the program state as desired. In other words, the corrector is itself nonmasking tolerant to the faults. And, each detector tolerates the faults by never falsely witnessing its detection predicate, even in the presence of the faults. In other words, each detector is itself masking tolerant to the faults. As can be expected, our two-stage design method can itself be used to design masking tolerance in the detectors, if their original design did not yield masking tolerant detectors.

Adding detector components by superposition. One way of simplifying the verification obligations is to add components to a program by superposing them on the program: if a program p is designed by a superposition on the program q, then it is trivially true that p does not interfere with q (although the converse need not be true, i.e., q may interfere with p).

In particular, superposition is well-suited for the addition of detector components to a nonmasking tolerant program, np, in Stage 2, since detectors need only to read (but not update) the state of np. Thus, the detectors do not interfere with the tasks of the corrector components in np.

When superposition is used, the verification of the converse obligation, i.e. that np does not interfere with the detectors, may be handled as follows. Ensure that the corrector in np terminates after it restores np to an invariant state and that as long as it has not terminated it prevents the detectors from witnessing their detection predicate. Aborting the detectors during the execution of the corrector guarantees that the detectors never witness their detection predicate incorrectly, and the eventual termination of the corrector guarantees that eventually detectors are not prevented from witnessing their detection predicate.

More specifically, the simplified verification obligations resulting from superposition are explained from Theorems 5.7 and 5.8. Let program p be designed by superposition on q such that $T_p \Rightarrow T_q$, $S_p \Rightarrow S_q$, and T_q converges to S_q in q.

Theorem 5.7.

If $\quad T_q$ converges to S_q in q

then $\quad T_p$ converges to S_p in p. $\square$

Theorem 5.8.

If $\quad T_q$ converges to S_q in q, and

$\quad T_p \wedge S_q$ converges to S_p in p

then $\quad T_p$ converges to S_p in p

Proof: Since q is nonmasking fault-tolerant, T_q converges to S_q in q. Since p is designed by a superposition on q, it follows that ($T_p \wedge T_q$ converges to $T_p \wedge S_q$). Since the converges-to relation is transitive and ($T_p \wedge S_q$ converges to $S_p \wedge S_q$), it follows that ($T_p \wedge T_q$ converges to $S_p \wedge S_q$), i.e., T_p converges to S_p in p. $\square$

Theorems 5.7 and 5.8 imply that if p is designed by superposition on a nonmasking tolerant program q, then to reason about p, it suffices to assume that q always satisfies its invariant S_q, even in the presence of faults. Of course, other interference strategies discussed in Chapter 3 can be used to achieve interference-freedom between components.

5.3.1 Data transfer : An Example of Stepwise Design

In this subsection, we illustrate how a masking fault-tolerant solution to the data transfer problem can be designed by first designing a nonmasking fault-tolerant solution and then enhancing its tolerance to masking. Recall the data transfer problem: An infinite input array at a sender process is to be copied, one array item at a time, into an infinite output array at a receiver process. The sender and receiver communicate via a bidirectional channel that can hold at most one message in each direction at a time. It is required that each input array item be copied into the output array exactly once and in the same order as sent. Moreover, eventually the number of items copied by the receiver should grow unboundedly.

Data transfer is subject to the faults that lose channel messages.

We will design the masking fault-tolerance to data transfer in two stages. The resulting program is the well known *alternating-bit protocol*.

Intolerant program. Iteratively, a simple loop is followed: sender s sends a copy of one array item to receiver r. Upon receiving this item, r sends an acknowledgment to s, which enables the next array item to be sent by s and so on. To this end, the program maintains binary variables rs in s and rr in r; rs is 1 if s has received an acknowledgment for the last item it sent, and rr is 1 if the r has received an item but has not yet sent an acknowledgment.

The 0 or 1 items in transit from s to r are denoted by the sequence cs, and the 0 or 1 acknowledgments in transit from r to s are denoted by the sequence cr. Finally, the index in the input array corresponding to the item that s will send next is denoted by ns, and the index in the output array corresponding to the item that r last received is denoted by nr.

The intolerant program contains four actions, the first two in s and the last two in r. By $ID1$, s sends an item to r, and by $ID2$, s receives an acknowledgment from r. By $ID3$, r receives an item from s, and by $ID4$, r sends an acknowledgment to s. Formally, the actions of the intolerant program, ID, are as follows (where $c1 \circ c2$ denotes concatenation of sequences $c1$ and $c2$):

$$
\begin{array}{llll}
ID1 :: & rs=1 & \longrightarrow & rs, cs := 0,\ cs \circ \langle ns \rangle \\
ID2 :: & cr \neq \langle\rangle & \longrightarrow & rs, cr, ns\ :=\ 1, tail(cr), ns+1 \\
 & & & \\
ID3 :: & cs \neq \langle\rangle & \longrightarrow & cs, rr, nr\ :=\ tail(cs), 1, head(cs) \\
ID4 :: & rr=1 & \longrightarrow & rr, cs\ :=\ 0,\ cr \circ \langle nr \rangle
\end{array}
$$

Remark. For brevity, we have ignored the actual data transfered between the sender and the receiver: we only use the array index of that data.

Invariant. When r receives an item, $nr = ns$ holds, and this equation continues to hold until s receives an acknowledgment. When s receives an acknowledgment, ns is exactly one larger than nr and this equation continues to hold until r receives the next item. Also, if cs is nonempty, cs contains only one item, $\langle ns \rangle$. Finally, in any state, exactly one of the four actions is enabled. Hence, the invariant of program ID is, S_{ID}, where

$$
\begin{array}{lll}
S_{ID} & = & ((rr=1\ \vee\ cr \neq \langle\rangle)\ \Rightarrow\ nr=ns)\ \wedge \\
 & & ((rs=1\ \vee\ cs \neq \langle\rangle)\ \Rightarrow\ nr=ns-1)\ \wedge \\
 & & (cs=\langle\rangle\ \vee\ cs=\langle ns \rangle)\ \wedge\ (|cs|+|cr|+rs+rr\ =\ 1)
\end{array}
$$

Fault Actions. The faults in this example lose either an item sent from s to r or an acknowledgment sent from r to s. The corresponding fault actions are as follows:

$$
\begin{array}{lll}
cs \neq \langle\rangle & \longrightarrow & cs := tail(cs) \\
cr \neq \langle\rangle & \longrightarrow & cr := tail(cr)
\end{array}
$$

Nonmasking tolerant program. Program ID is intolerant as it deadlocks when a fault loses an item or an acknowledgment. Hence, we add nonmasking tolerance to this fault by adding a corrector whose correction predicate is S_{ID}. In order to perform this correction, whenever an item or an acknowledgment is lost, the corrector retransmits the last item.

Thus, the nonmasking program consists of five actions; four actions are identical to the actions of program ID, and the fifth action is the action of the corrector that retransmits the last item that was sent. This action is executed when both channels, cs and cr, are empty, and rs and rr are both zero. In practice, this action can be implemented by waiting for a some predetermined timeout so that the sender can be sure that either the item or the acknowledgment is lost, but we present only the abstract version of the action. Formally, the actions of the nonmasking program, ND, are as follows:

$$
\begin{array}{lll}
ND1 :: & ID1 & \\
ND2 :: & ID2 & \\
ND5 :: & cs = \langle\rangle \ \wedge \ cr = \langle\rangle \ \wedge \ rs = 0 \ \wedge \ rr = 0 \quad \longrightarrow & cs := cs \circ \langle ns \rangle \\
 & & \\
ND3 :: & ID3 & \\
ND4 :: & ID4 &
\end{array}
$$

Fault-span and invariant. If an item or an acknowledgment is lost, the program reaches a state where cs and cr are empty and rs and rr are both equal to zero. Also, even in the presence of faults, if cs is nonempty, it contains exactly the item whose index in the input array is $\langle ns \rangle$. Thus, the fault-span of the nonmasking program is

$$T_{ND} \;=\; (cs = \langle\rangle \;\vee\; cs = \langle ns \rangle) \quad \wedge \quad (|cs| + |cr| + rs + rr \;\leq\; 1)$$

and the invariant is the same as the invariant of ID, i.e.,

$$S_{ND} \;=\; S_{ID}$$

Enhancing the tolerance to masking. Program ND is not yet masking tolerant, since r may receive duplicate items if an acknowledgment from r to s is lost. Hence, to enhance the tolerance to masking, we need to restrict the action $ID3$ so that r copies an item into the output array iff it is not a duplicate.

Upon receiving an item, if r checks that nr is exactly one less than the index number received with the item, r will receive every item exactly once. Thus, we can enhance its tolerance to masking by adding a detector that checks whether the counter value in the message is correct. However, this check forces the size of the message sent from the s to r to grow unboundedly. However, we can exploit the fact that in ND, ns and nr differ by at most 1, in order to simulate this check by sending only a single bit with the item as follows.

Process s adds one bit, bs, to every item it sends such that the bit values added to two consecutive items are different and the bit values added to an item and its duplicates are the same. Thus, to detect that a message is duplicate, r maintains a bit, br, that denotes the sequence number of the last message it received. It follows that an item received by r is a duplicate iff br is the same as the sequence number in that message.

The masking program consists of five actions. These actions are as follows:

$$
\begin{array}{lll}
MD1 :: & rs=1 & \longrightarrow \; rs, cs \; := \; 0, \; cs \circ \langle ns, bs \rangle \\
MD2 :: & cr \neq \langle\rangle & \longrightarrow \; rs, cr, ns, bs \; := \; 1, tail(cr), ns+1, bs \oplus 1 \\
MD5 :: & cs=\langle\rangle \; \wedge \; cr=\langle\rangle \; \wedge & \\
 & rs=0 \; \wedge \; rr=0 & \longrightarrow \; cs \; := \; cs \circ \langle ns, bs \rangle \\
 & & \\
MD3 :: & cs \neq \langle\rangle & \longrightarrow \; \text{if } ((head(cs))_2 \neq br) \text{ then} \\
 & & \qquad nr, br \; := \; (head(cs))_1, (head(cs))_2; \\
 & & \quad\; cs, rr := tail(cs), 1 \\
MD4 :: & rr=1 & \longrightarrow \; rr, cs \; := \; 0, \; cr \circ \langle nr, br \rangle
\end{array}
$$

Remark. Observe that in the masking program, the array index ns and nr need not be sent on the channel as it suffices to send the bits bs and br. With this modification, the resulting program is the alternating bit protocol.

Invariant. In any state reached in the presence of program and fault actions, if cs is nonempty, cs has exactly one item, $\langle ns, bs \rangle$. Also, when r receives an item, $nr = ns$ holds, and this equation continues to hold until s receives an acknowledgment. Moreover, bs is the same as $ns \; mod \; 2$, br is the same as $nr \; mod \; 2$, and exactly one of the five actions is enabled. Finally, nr is the same as ns or nr is one less than ns. Thus, the invariant of the masking program is S_{MD}, where

$$
\begin{aligned}
S_{MD} \;=\; & (cs=\langle\rangle \; \vee \; cs=\langle ns, bs \rangle) \; \wedge \\
& ((rr=1 \; \vee \; cr \neq \langle\rangle) \Rightarrow nr=ns) \; \wedge \; (|cs| + |cr| + rs + rr \; \leq \; 1) \; \wedge \\
& bs=(ns \; mod \; 2) \quad \wedge \quad br=(nr \; mod \; 2) \quad \wedge \quad (nr=ns \; \vee \; nr = ns-1)
\end{aligned}
$$

Theorem 5.9. The alternating-bit program, MD, is masking tolerant from S_{MD}.

5.4 Chapter Summary

In this chapter, we showed how masking fault-tolerance is related to fail-safe fault-tolerance and nonmasking fault-tolerance. Using this relation, we showed that detectors and correctors together form a basis of masking fault-tolerance. We also used this relation to provide a stepwise method for designing masking fault-tolerance.

CHAPTER 6

MULTITOLERANCE AND ITS DESIGN

In the last three chapters, we identified the two fault-tolerance components that form a basis of fault-tolerance design. As mentioned in the introduction, one purpose of identifying these components is to develop a systematic method to design fault-tolerant programs, including multitolerant programs. In this chapter, we present this method.

We proceed as follows: First, we define what it means for a program to be multitolerant (cf. Section 6.1). Then, we present our method to transform a fault-intolerant program into a multitolerant program (cf. Section 6.2). Subsequently, we illustrate our method by designing a multitolerant token ring program (cf. Section 6.3). This example demonstrates the use of detectors and correctors in the design of a program which provides continuous tolerance in the sense that the time required to correct the program state is proportional to the severity of the faults.

6.1 Definition

Let p be a program with invariant S, $F1..Fn$ be n fault-classes, $SPEC$ be a specification, and $l1, l2, .., ln$ be types of tolerance (i.e., masking, nonmasking or fail-safe). We say that p is multitolerant to $F1..Fn$ for $SPEC$ from S iff for each fault-class $Fj, 1 \leq j \leq n$, p is lj Fj-tolerant for $SPEC$ from S . ☐

The definition may be understood as follows: In the absence of faults, p refines $SPEC$ from S. In the presence of faults in class Fj, p is perturbed only to states where some Fj-span predicate for S, Tj, is true. (Note that there exists a potentially different fault-span for each fault-class.) And, in states where Tj is true, p refines an appropriate tolerance specification of $SPEC$. For example, if $lj = fail-safe$, then p refines the fail-safe tolerance specification of $SPEC$, namely $SSPEC$, from Tj.

Example: Memory access (continued). Observe that the memory access program, $d1; (c; d2); p$, discussed in Section 3.3.1, is multitolerant to the classes of protection faults and page faults: it is fail-safe tolerant to the former and masking tolerant to latter. In particular, in the presence of a page fault, it always obtains the correct data from the memory. And in the presence of a protection fault, it obtains no data value.

6.2 Compositional and Stepwise Design Method

In this section, we describe our compositional method to design multitolerant system that adds tolerance to different classes of faults in a stepwise fashion. More specifically, our method starts with a fault-intolerant program and, in a stepwise manner, considers the fault-classes in some fixed total order, say $F1..Fn$. In the first step, the intolerant program is augmented with detector and/or corrector components so as to add suitable tolerance to the fault-class $F1$. The resulting program is then augmented with other detector/corrector components, in the second step, so as to add suitable tolerance the fault-class $F2$ while ensuring that the tolerance to $F1$ is preserved. And so on until, in the n-th step, the tolerance to Fn is added while preserving the tolerances to $F1..Fn{-}1$. The multitolerant program designed thus has the structure shown in Figure 6.1.

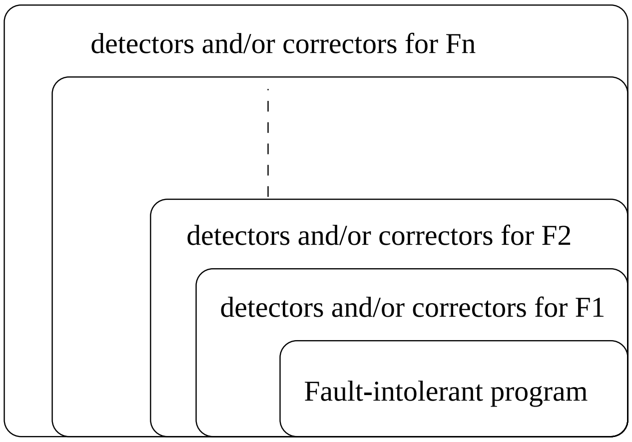


Figure 6.1: Structure of a multitolerant program designed using our method

First step. Let p be the intolerant program with invariant S. To add fault-tolerance to $F1$, we need to add suitable detector and corrector components to p. For example, if we need to add fail-safe tolerance to $F1$ then we compute a detection predicate, sf_{ac} for each action ac of p and add a detector whose detection predicate is sf_{ac}. As shown in Chapter 3, if each action in p is restricted to execute only in states where a detection predicate of that action is true then the resulting program is fail-safe tolerant. Likewise, if we need to design nonmasking tolerance to $F1$ then we need to add a corrector whose correction predicate is S. And, if we need to design masking fault-tolerance then we need to add both detectors and correctors.

As discussed in Chapter 5, the addition of the detectors and correctors in the design of masking tolerance may itself be simplified by using a stepwise approach: For instance, to design masking tolerance, we may first augment the program with detectors, and then augment the resulting fail-safe tolerant program with correctors. Alternatively, we may first augment the program with correctors, and then augment the resulting nonmasking tolerant program with detectors (cf. Figure 6.2).

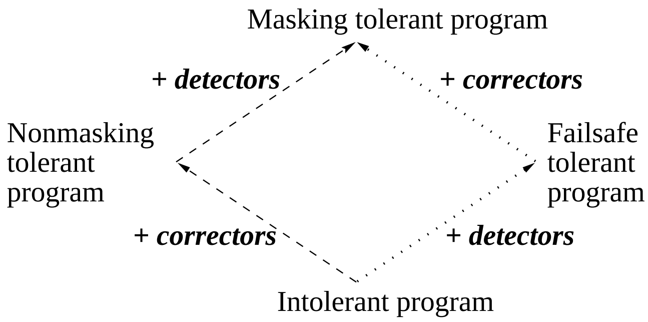


Figure 6.2: Two approaches for stepwise design of masking tolerance

Also, as discussed in Chapters 3 and 4, the added components themselves must be fault-tolerant. More specifically, the detectors added in the design of fail-safe tolerance must themselves be fail-safe tolerant, the correctors added in the design of nonmasking tolerance must themselves be nonmasking tolerant, and the detectors and correctors added in the design of masking tolerance must themselves be masking tolerant.

With the addition of detector and/or corrector components to p, it remains to show that, in the resulting program $p1$, the components do not interfere with p and that p does not interfere with the components. Note that $p1$ may contain variables and actions that were not in p and, hence, invariants and fault-spans of $p1$ may differ from those of p. Therefore, letting $S1$ be an invariant of $p1$ and $T1$ be an $F1$-span of $p1$ for $S1$, we show the following.

1. In the absence of $F1$, i.e., in states where $S1$ is true, the components do not interfere with p, i.e., each computation of p is in the problem specification even if it executes concurrently with the new components.

2. In the presence of $F1$, i.e., in states where $T1$ is true, p does not interfere with the components, i.e., each computation of the components is in the components' specification (in the sense prescribed by its type of tolerance) even if they execute concurrently with p.

Second step. This step adds $l2$-tolerance to $F2$ and preserves the $l1$-tolerance to $F1$. To add $l2$-tolerance to $F2$, just as in the first step, we add new detector and corrector components to $p1$. Then, we account for the possible interference between the executions of these added components and of $p1$. More specifically, letting $S2$ be an invariant of the resulting program $p2$, $T21$ be an $F1$-span of $p2$ for $S2$, and $T22$ denote an $F2$-span of $p2$ for $S2$, we show the following.

1. In the absence of $F1$ and $F2$, i.e., in states where $S2$ is true, the newly added components do not interfere with $p1$, i.e., each computation of $p1$ is in the problem specification even if it executes concurrently with the new components.

2. In the presence of $F2$, i.e., in states where $T22$ is true, $p1$ does not interfere with the new components, i.e., each computation of the new components is in the

new components' specification (in the sense prescribed by its type of tolerance) even if they execute concurrently with $p1$.

3. In the presence of $F1$, i.e., in states where $T21$ is true, the newly added components do not interfere with the $l1$-tolerance of $p1$ for $F1$, i.e., each computation of $p1$ is in the specification, $l1$-tolerant to $F1$, even if $p1$ executes concurrently with the new components.

Remaining steps. For the remaining steps of the design, where we add tolerance to $F3..Fn$, the procedure of the second step is generalized accordingly.

6.3 Case Study in Multitolerance Design : Token Ring

Recall the mutual exclusion problem: Multiple processes may each access their critical section provided that at any time at most one process is accessing its critical section. Moreover, no process should wait forever to access its critical section, assuming that each process leaves its critical section in finite time.

Mutual exclusion is readily achieved by circulating a token among processes and letting each process enter its critical section only if it has the token. In a token ring program, in particular, the processes are organized in a ring and the token is circulated along the ring in a fixed direction.

In this case study, we design a multitolerant token ring program. The program is masking tolerant to any number, K, of faults that each corrupt the state of some process detectably. Its tolerance is continuous in the sense that if K state corruptions occur, it corrects its state within $\Theta(K)$ time. Thus, a quantitatively unique measure of tolerance is provided to each FK, where FK is the fault-class that causes at most K state corruptions of processes.

By detectable corruption of the state of a process, we mean that the corrupted state is detected by that process before any action inadvertently accesses that state. The state immediately before the corruption may, however, be lost. (For our purposes, it is irrelevant as to what caused the corruption; i.e., whether it was due to the loss of a message, the duplication of a message, timing faults, the crash and subsequent restart of a process, etc.)

We proceed as follows: First, we describe a simple token ring program that is intolerant to detectable state corruptions. Then, we add detectors and correctors so as to achieve masking tolerance to the fault that corrupts the state of one process. Progressively, we add more detectors and correctors so as to achieve masking tolerance to the fault-class that corrupts process states at most K, $K > 1$, times.

6.3.1 Fault-Intolerant Binary Token Ring

Processes $0..N$ are organized in a ring. The token is circulated along the ring such that process j, $0 \leq j \leq N$, passes the token to its successor $j+1$. (In this section, $+$ and $-$ are in modulo $N+1$ arithmetic.) Each process j maintains a binary variable $x.j$. Process $j, j \neq N$, has the token iff $x.j$ differs from its successor $x.(j+1)$ and process N has the token iff $x.N$ is the same as its successor $x.0$.

The program, TR, consists of two actions for each process j. Formally, these actions are as follows (where $+_2$ denotes modulo 2 addition):

$$
\begin{array}{llll}
TR1 :: & j \neq 0 \ \wedge \ x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1) \\
TR2 :: & j = 0 \ \wedge \ x.j \neq (x.N +_2 1) & \longrightarrow & x.j := x.N +_2 1
\end{array}
$$

Invariant. Consider a state where process j has the token. In this state, since no other process has a token, the x value of all processes $0..j$ is identical and the x value of all processes $(j+1)..N$ is identical. Letting X denote the string of binary values $x.0, x.1, ..., x.N$, we have that X satisfies the regular expression $(0^l\, 1^{(N+1-l)} \cup 1^l\, 0^{(N+1-l)})$, which denotes a sequence of length $N+1$ consisting of zeros followed by ones or ones followed by zeros. Thus, an invariant of TR is

$$S_{TR} \;=\; X \in (\bigcup l : 0 \leq l \leq N+1 : (0^l\, 1^{(N+1-l)} \;\cup\; 1^l\, 0^{(N+1-l)}))$$

6.3.2 Adding Tolerance to 1 State Corruption

Based on our assumption that state corruption is detectable, we introduce a special value $\perp$, such that when any process j detects that its state (i.e., the value of $x.j$) is corrupted, it resets $x.j$ to $\perp$.

We can now readily design masking tolerance to a single corruption of state at any process j by ensuring that (i) the value of $x.j$ is eventually corrected so that it is no longer $\perp$ and (ii) in the interim, no process (in particular, $j+1$) inadvertently gets the token as a result of the corruption of $x.j$.

For (i), we add a corrector at each process j: it corrects $x.j$ from $\perp$ to a value that is either 0 or 1. The corrector at j, $j \neq 0$, copies $x.(j-1)$; the corrector at j, $j=0$, copies $x.N +_2 1$. Thus, the corrector action at j has the same statement as the action of TR at j, and we can merge the corrector and TR actions.

For (ii), we add a detector at each process j : Its detection predicate is $x.(j-1) \neq \perp$ and it has no actions. The witness predicate of this detector (which, in this case, is the detection predicate itself) is used to restrict the actions of program TR at j. Hence, the actions of TR at j execute only when $x.(j-1) \neq \perp$ is true. As a result, the execution of actions of TR is always safe (i.e., these actions cannot inadvertently generate a token).

The augmented program, PTR, is

$PTR1 ::$	$x.(j-1) \neq \perp$	$\wedge$	$TR1$
$PTR2 ::$	$x.N \neq \perp$	$\wedge$	$TR2$

Fault Actions. When the state of $x.j$ is corrupted, $x.j$ is set to $\perp$. Hence, the fault action is

$$x\!-\!corr :: \quad true \quad \longrightarrow \quad x.j := \perp$$

Proof of interference-freedom. Starting from a state where S_{TR} is true, in the presence of faults that set the x value of a process to $\perp$, string X always satisfies the regular expression $(0 \cup \perp)^l \ (1 \cup \perp)^{(N+1-l)}$or $(1 \cup \perp)^l \ (0 \cup \perp)^{(N+1-l)}$. Thus, an invariant of PTR is S_{PTR}, where

$$\begin{aligned} S_{PTR} = X \in (\textstyle\bigcup l : 0 \leq l \leq N+1 : \\ ((0 \cup \perp)^l \ (1 \cup \perp)^{(N+1-l)} \ \cup \ (1 \cup \perp)^l \ (0 \cup \perp)^{(N+1-l)})) \ \wedge \\ |\{j : x.j = \perp\}| \leq 1 \end{aligned}$$

Consider the detector at j: Both its detection and witness predicates are $x.(j–1) \neq \perp$. Since the detects relation is trivially reflexive in its first two arguments, it follows that $x.(j-1) \neq \perp$ detects $x.(j-1) \neq \perp$ in PTR. In other words, the detector is not interfered by any other actions.

Consider the corrector at j: Both its correction and witness predicates are $x.j \neq \bot$. Since the program actions are identical to the corrector actions, by Theorem 3.14, the corrector actions are not interfered by the actions of TR. Also, since the detectors have no actions, the detectors at processes other than j do not interfere with the corrector at j; moreover, since at most one x value is set to $\bot$, when $x.j = \bot$ and thus the corrector at j is enabled, the witness predicate of the detector at j is true and hence the corrector at j is not interfered by the detector at j.

Consider the program actions of TR: Their safety follows from the safety of the detectors, described above. And, their progress follows from the progress of the correctors, which ensure that starting from a state where S_{PTR} is true and a process state is corrupted every computation of PTR reaches a state where S_{TR} is true, and the progress of the detectors, which ensures that no action of TR is indefinitely blocked from executing.

Observe that our proof of mutual interference-freedom illustrates that we do not have to re-prove the correctness of TR for the new invariant. Observe, also, that if the state of process j is corrupted then within $\Theta(1)$ time the corrector at j corrects the state of j.

6.3.3 Adding Tolerance to $2..N$ State Corruptions

The proof of non-interference of program PTR can be generalized to show that PTR is also masking tolerant to the fault-class that twice corrupts process state.

The generalization is self-evident for the case where the state corruptions are separated in time so that the first one is corrected before the second one occurs. For the case where both state corruptions occur concurrently, say at processes j and k, we need to show that the correctors at j and k truthify $x.j \neq \perp$ and $x.k \neq \perp$, without interference by each other and the other actions of the program. Let us consider two subcases: (i) j and k are non-neighboring, and (ii) j and k are neighboring.

For the first subcase, j and k correct $x.j$ and $x.k$ from their predecessors $j-1$ and $k-1$, respectively. This execution is equivalent to the parallel composition of the correctors at j and k. By Theorem 4.4, PTR reaches a state where $x.j$ and $x.k$ are not $\perp$.

For the second subcase (letting j be the predecessor of k), j corrects $x.j$ from its predecessor $j-1$, truthifies $x.j \neq \perp$ and then terminates. Since the corrector at j does not read any variables written by the corrector at k. Thus, from the analogue of Theorem 3.13 for the case of correctors, the corrector at j is not interfered by the corrector at k. After $x.j \neq \perp$ is truthified, the corrector at k corrects $x.k$ from its predecessor j. By Theorem 4.5, the corrector at k is not interfered by the corrector at j. Since the correctors at j and k do not interfere with each other, it follows that the program reaches a state where $x.j$ and $x.k$ are not $\perp$.

In fact, as long as the number of faults is at most N, there exists at least one process j with $x.j \neq \perp$. PTR ensures that the state of such a j eventually causes $j+1$ to correct its state to $x.(j+1) \neq \perp$. Such corrections will continue until no process has its x value set to $\perp$. Hence, PTR tolerates up to N faults and the time required to converge to S_{TR} is $\Theta(K)$, where K is the number of faults.

6.3.4 Adding Tolerance to More Than N State Corruptions

Unfortunately, if more than N faults occur, program PTR deadlocks iff it reaches a state where the x value of all processes is $\perp$. To be masking tolerant to the fault-classes that corrupt the state of processes more than N times, a corrector is needed that detects whether the state of all processes is $\perp$ and, if so, corrects the program to a state where the x value of some process (say 0) to be equal to 0 or 1.

Since the x values of all processes cannot be accessed simultaneously, the corrector detects in a sequential manner whether the x values of all processes are $\perp$. Let the detector added for this purpose at process j be denoted as dj and the (sequentially composed) detector that detects whether the x values of all processes is corrupted be $dN; d(N-1); ...; d0$.

To design dj, we add a value $\top$ to the domain of $x.j$. When dN detects that $x.N$ is equal to $\perp$, it sets $x.N$ to $\top$. Likewise, when dj, $j < N$, detects that $x.j$ is equal to $\perp$, it sets $x.j$ to $\top$. Note that since dj is part of the sequential composition, it is restricted to execute only after $j{+}1$ has completed its detection, i.e., when $x.(j{+}1)$ is equal to $\top$. It follows that when j completes its detection, the x values of processes $j..N$ are corrupted. In particular, when $d0$ completes its detection, the x values of all processes are corrupted. Hence, when $x.0$ is set to $\top$, it suffices for the corrector to reset $x.0$ to 0.

To ensure that while the corrector is executing, no process inadvertently gets the token as a result of the corruption of $x.j$, we add detectors that restrict the actions of PTR at $j{+}1$ to execute only in states where $x.j \neq \top$ is true.

Actions. Program FTR consists of five actions at each process j. Like PTR, the first two actions, $FTR1$ and $FTR2$, pass the token from j to $j{+}1$ and are restricted by the trivial detectors to execute only when $x.(j-1)$ is neither $\perp$ nor $\top$. Action $FTR3$ is dN; it lets process N change $x.N$ from $\perp$ to $\top$. Action $FTR4$ is dj for $j<N$. Action $FTR5$ is the corrector action at process 0: it lets process 0 correct $x.0$ from $\top$ to 0. Formally, these actions are as follows:

$$
\begin{array}{llcll}
FTR1 :: & x.(j-1) \neq \top & \wedge & PTR1 & \\
FTR2 :: & x.N \neq \top & \wedge & PTR2 & \\
FTR3 :: & x.N = \perp & & \longrightarrow & x.N := \top \\
FTR4 :: & j \neq N \ \wedge \ x.j = \perp \ \wedge \ x.(j+1) = \top & & \longrightarrow & x.j := \top \\
FTR5 :: & x.0 = \top & & \longrightarrow & x.0 := 0
\end{array}
$$

Invariant. Starting from a state where S_{PTR} is true, the detector can change the trailing $\perp$ values in X to $\top$. Thus, FTR may reach a state where X satisfies the regular expression $(1 \cup \perp)^l \ (0 \cup \perp)^m \top^{(N+1-l-m)} \quad \cup \quad (0 \cup \perp)^l \ (1 \cup \perp)^m \top^{(N+1-l-m)}$. Subsequent state corruptions may perturb X to the form $(1 \cup \perp)^l \ (0 \cup \perp)^m (\perp \cup \top)^{(N+1-l-m)} \ \cup \ (0 \cup \perp)^l \ (1 \cup \perp)^m \ (\perp \cup \top)^{(N+1-l-m)}$. Since all actions preserve this last predicate, an invariant of FTR is

$$
\begin{array}{l}
S_{FTR} = X \in (\bigcup l, m, : 0 \leq l, m, l+m \leq N+1 : \\
\qquad\qquad (\ (1 \cup \perp)^l \ (0 \cup \perp)^m (\perp \cup \top)^{(N+1-l-m)} \ \cup \\
\qquad\qquad\quad (0 \cup \perp)^l \ (1 \cup \perp)^m \ (\perp \cup \top)^{(N+1-l-m)}))
\end{array}
$$

Proof of interference-freedom. To design FTR, we have added a corrector (actions $FTR3-5$) to program PTR to ensure that for some j, $x.j$ is not corrupted, i.e., the correction predicate of this corrector is V, where $V = (\exists j :: x.j = 0 \ \vee \ x.j = 1)$.

This corrector is of the form $dN; d(N–1); ...; d0; c0$, where each dj is an atomic detector at process j and $c0$ is an atomic corrector at process 0.

The detection predicate of $dN; d(N-1); ...; d0$ is $\neg V$ and its witness predicate is $x.0 = \top$. To show that this detector in isolation satisfies its specification, observe that

1. $x.N = \top$ detects $(\forall j : N \geq j \geq N : x.j \neq 0 \wedge x.j \neq 1)$ in dN for S_{FTR}.

2. $x.(N-1) = \top$ detects $(\forall j : N \geq j \geq N-1 : x.j \neq 0 \wedge x.j \neq 1)$ in $d(N-1)$ for $(S_{FTR} \wedge (\forall j : N \geq j \geq N : x.j \neq 0 \wedge x.j \neq 1))$.

From (1) and (2), by Theorem 3.4, $x.(N-1) = \top$ detects $(\forall j : N \geq j \geq (N-1) : x.j \neq 0 \wedge x.j \neq 1)$ in $dN; d(N–1)$ for S_{FTR}. Using the same argument, $x.0 = \top$ detects $(\forall j : N \geq j \geq 0 : x.j \neq 0 \wedge x.j \neq 1)$ in $dN; d(N-1); ...; d0$ for S_{FTR}, i.e., $x.0 = \top$ detects $(\neg V)$ in $dN; d(N-1); ...; d0$ for S_{FTR}.

Now, observe that S_{FTR} converges to V in $dN; d(N–1); ...; d0; c0$: if V is violated execution of $dN; d(N-1); ...; d0$ will eventually truthify $x.0 = \top$, and execution of $c0$ will truthify V. Thus, V corrects V in $dN; d(N-1); ...; d0; c0$ for S_{FTR}.

The corrector is not interfered by the actions $FTR1$ and $FTR2$. This follows from the fact that $FTR1$ and $FTR2$ do not interfere with each dj and $c0$ (by using Theorem 3.15).

In program FTR, we have also added a detector at process j that detects $x.(j–1) \neq \top$. As described above (for the 1 fault case), this detector does not interfere with other actions, and it is not interfered by other actions.

Finally, consider actions of program PTR: their safety follows from the safety of the detector described above. Also, starting from any state in S_{FTR}, the program reaches a state where x value of some process is not corrupted. Starting from such a state, as in program PTR, eventually the program reaches a state where S_{TR} is truthified, i.e., no action of PTR is permanently blocked. Thus, the progress of these actions follows.

Theorem 6.1 Program FTR is masking tolerant for invariant S_{FTR} to the fault-classes FK, $K \geq 1$, where FK detectably corrupts process states at most K times. Moreover, S_{FTR} converges to S_{TR} in FTR within $\Theta(K)$ time.

6.4 Chapter Summary

In this chapter, we presented a method that uses detectors and correctors to add fault-tolerance to multiple classes of faults in a stepwise fashion. To add fault-tolerance to each fault, we first designed appropriate detectors and correctors. Then, we ensured that these detectors and correctors do not affect the fault-tolerance to faults considered in previous steps.

Our method satisfies the goals discussed in the introduction. More specifically,

1. it can deal with a rich class of faults, including process faults, communication faults, hardware faults, software faults, network failure, security intrusions, safety hazards, configuration changes and load variations.

2. by designing efficient detectors and correctors, it provides the potential to design efficient fault-tolerant programs (examples of such programs are demonstrated in Chapter 8 as well as in [10–13, 38–42].),

3. can be used to make a rich class of systems fault-tolerant,

4. being stepwise in nature, it can be used to transform fault-tolerant programs to add tolerance to a new fault-class, and
5. it is not application-dependent.

CHAPTER 7

RELATION OF DETECTORS AND CORRECTORS TO EXISTING FAULT-TOLERANCE METHODS

In this chapter, we illustrate that programs designed using existing methods can be alternatively designed in terms of detectors and correctors. More specifically, we show that canonical fault-tolerant programs designed using two existing methods, namely replication and Schneider's state machine approach can be designed using detectors and correctors.

Regarding replication, we focus our attention on the central problem of triple modular redundant system design. Regarding, Schneider's state machine approach, recall that [59] this approach consists of two requirements, *Agreement* and *Order*. We, therefore, consider the problem of repetitive Byzantine agreement that focuses on these two requirements.

This chapter is organized as follows: In Section 7.1, we show how triple modular redundancy program can be designed in terms of detectors and correctors. In Section 7.2, we show how detectors and correctors are used in the design of repetitive Byzantine agreement. Since Schneider's state machine approach is intended towards designing masking fault-tolerance alone, a solution designed using this approach provides only masking Byzantine-tolerance. We show how such a masking fault-tolerant solution can be designed in Section 7.2.2. To further illustrate the use of detectors and correctors in the design of multitolerance, in Section 7.2.3, we add stabilizing tolerance to transient and Byzantine faults while preserving the masking fault-tolerance to Byzantine faults. Finally, in Section 7.3, we show how the program developed in Section 7.2 can be used to design a program that provides only nonmasking fault-tolerance to Byzantine faults.

7.1 Triple Modular Redundancy (Replication)

Consider a triple modular redundant system with 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. It is required 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 $\perp$ of *out* denotes that *out* has not been assigned. Thus, the action of IR is as follows:

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

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 $(x = y \lor x = z)$. Observe that $(x = y \lor x = z)$ detects $(x = uncor)$ in the program that merely evaluates the state predicate $(x = y \lor x = z)$ upon starting from the states S where all inputs are identical. To add fail-safe tolerance, IR is restricted to execute only when the witness predicate of DR is satisfied. Thus, we have

$DR; IR$ is fail-safe 'one input corruption'-tolerant from S.

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:

$$CR1 :: \quad out = \perp \ \land \ (y = z \ \lor \ y = x) \quad \longrightarrow \quad out := y$$

$$CR2 :: \quad out = \perp \ \land \ (z = x \ \lor \ z = y) \quad \longrightarrow \quad out := z$$

Thus, we have

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

Note that the program $DR; IR \ [\!] \ CR$ is the triple modular redundancy program.

7.2 Repetitive Agreement (State Machine Approach)

In the repetitive agreement problem, the system consists of a set of processes, including a "general" process, g. Each computation of the system consists of an infinite sequence of rounds; in each round, the general chooses a binary decision value

$d.g$ and, depending upon this value, all other processes output a binary decision value of their own.

The system is subject to two fault-classes: The first one permanently and undetectably corrupts some processes to be Byzantine, in the following sense: each Byzantine process follows the program skeleton of its non-Byzantine version, i.e., it sends messages and performs output of the appropriate type whenever required by its non-Byzantine version, but the data sent in the messages and the output may be arbitrary. The second one transiently and undetectably corrupts the state of the processes in an arbitrary manner and possibly also permanently corrupts some processes to be Byzantine.

(Note that, if need be, the model of a Byzantine process can be readily weakened to handle the case when the Byzantine process does not send its messages or perform its output, by detecting their absence and generating arbitrary messages or output in response.)

Repetitive agreement specification. Repetitive agreement specification is a set of computations such that each round in those computations satisfies Validity and Agreement, defined below.

- *Validity:* If g is non-Byzantine, the decision value output by every non-Byzantine process is identical to $d.g$.

- *Agreement:* Even if g is Byzantine, the decision values output by all non-Byzantine processes are identical.

The problem Design a program that provides the following fault-tolerance properties for repetitive agreement specification.

1. *Masking tolerance.* In the presence of the faults in the first fault-class, i.e., Byzantine faults, masking tolerance specification of repetitive agreement is satisfied, i.e., each round in the program computation satisfies Validity and Agreement.

2. *Stabilizing tolerance.* In the presence of the faults in the second fault-class, i.e., transient and Byzantine faults, stabilizing tolerance specification of repetitive agreement is satisfied, i.e., upon starting from an arbitrary state (which may be reached if transient and Byzantine failures occur), eventually a state must be reached in the program computation from where every future round satisfies Validity and Agreement. □

Before proceeding to compositionally design a masking as well as stabilizing tolerant repetitive agreement program, let us recall the wellknown fact that for repetitive agreement to be masking tolerant it is both necessary and sufficient for the system to have at least $3t+1$ processes, where t is the total number of Byzantine processes [47]. Therefore, for ease of exposition, we will initially restrict our attention, in Sections 7.2.1-7.2.3, to the special case where the total number of processes in the system (including g) is 4 and, hence, t is 1. In other words, the Byzantine failure fault-class may corrupt at most one of the four processes. Later, in Section 7.2.4, we will extend our multitolerant program for the case where t may exceed 1.

7.2.1 Designing a Fault-Intolerant Program

The following simple program suffices in the absence of faults: In each round, the general sends its new $d.g$ value to all non-general processes. When a process receives this $d.g$ value, it outputs that value and sends an acknowledgment to the general.

After the general receives acknowledgments from all the non-general processes, it starts the next round which repeats the same procedure.

We let each process j maintain a variable $d.j$, denoting the decision of j, that is set to $\perp$ when j has not yet copied the decision of the general. Also, we let j maintain a sequence number $sn.j$, $sn.j \in \{0..1\}$, to distinguish between successive rounds.

The general process. The general executes only one action, RG1: when the sequence numbers of all processes become identical, the general starts a new round by choosing a new value for $d.g$ and incrementing its sequence number, $sn.g$. Thus, letting $\oplus$ denote addition modulo 2, the action of the general is:

$$RG1 :: \quad (\forall k :: sn.k = sn.g) \quad \longrightarrow \quad d.g, sn.g := new\_decision(), sn.g \oplus 1$$

The non-general processes. Each non-general process j executes two actions: The first action, $RO1$, is executed after the general has started a new round, in which case j copies the decision of the general. It then executes its second action, $RO2$, which outputs its decision, increments its sequence number to denote that it is ready to participate in the next round, and resets its decision to $\perp$ to denote that it has not yet copied the decision of the general in that round. Thus, the two actions of j are:

$$RO1 :: \quad d.j = \perp \ \wedge \ (sn.j \oplus 1 \ = \ sn.g) \quad \longrightarrow \quad d.j := d.g$$

$$RO2 :: \quad d.j \neq \perp \quad \longrightarrow \quad \{ \text{ output } d.j \ \}; \, d.j, sn.j := \perp, sn.j \oplus 1$$

Lemma 7.1. Starting from states where the sequence numbers of all processes are identical and the decisions of all non-general processes are equal to $\perp$, each computation of R satisfies the repetitive agreement specification.

Proof. In any start state, only the general can execute, thus starting a new round by executing $RG1$. In the resulting state, each non-general process can only copy the decision of the general by executing $RO1$ and then output this decision by executing action $RO2$. Thus, Validity and Agreement are satisfied. Also, after each processes executes action $RO2$, the resulting state is again a starting state. Therefore, Validity and Agreement are satisfied in each successive round.

7.2.2 Adding Masking Fault-Tolerance to Byzantine Faults

Program R is neither masking tolerant nor stabilizing tolerant to Byzantine failure. In particular, R may violate Agreement if the general becomes Byzantine and sends different values to the non-general processes. Note, however, that since these values are binary, at least two of them are identical. Therefore, for R to mask the Byzantine failure of any one process, it suffices to add (1) a detector that restricts $RO2$ in such a way that each non-general process only outputs a decision that is the majority of the values received by the non-general processes, and (2) a corrector that guarantees that eventually the decision of a process would be equal to the majority of the non-general processes. More specifically, for each round, let $v.j$ denote the value obtained by j in that round when it executes $RO1$, and let *cordec* be defined as follows:

$$\begin{aligned} cordec &= d.g && \text{if } \neg b.g \\ &= (majority\ j :: v.j) && \text{otherwise} \end{aligned}$$

In order to ensure safety in the presence of Byzantine faults, we need to add a detector whose detection predicate is $d.j = cordec$ and restrict action $RO2$ so that it executes only after the detection is complete. Also, we need to add a corrector whose correction predicate is $d.j = cordec$ so that action $RO2$ can be eventually executed. To implement these detectors and correctors, for each process k (including j itself), we let j maintain a local copy of $d.k$ in $D.j.k$. Hence, the decision value of the majority can be computed over the set of $D.j.k$ values for all k. To determine whether a value $D.j.k$ is from the current round or from the previous round, j also maintains a local copy of the sequence number of k in $SN.j.k$, which is updated whenever $D.j.k$ is. Also, we associate with each process j an auxiliary variable $b.j$ that is true iff j is Byzantine.

The general process. To capture the effect of Byzantine failure, one action, $MRG2$, is added to the original action $RG1$ (which we rename as $MRG1$): $MRG2$ lets g change its decision value arbitrarily and is executed only if g is Byzantine. Thus, the actions for g are:

$MRG1 ::\quad RG1$

$MRG2 ::\quad b.g \quad\longrightarrow\quad d.g := 0|1$

The non-general processes. The masking Byzantine-tolerant program consists of five actions: $MRO1$–5. Action $MRO1$ is the same as action $R1$. Actions $MRO2$–3 are the detector actions and actions $MRO2-4$ are the corrector actions. (Note that the actions of the detector and the corrector overlap.) $MRO2$ is executed after j receives a decision value from g, to set $D.j.j$ to $d.j$, provided that all non-general processes had obtained a copy of $D.j.j$ in the previous round. $MRO3$ is executed after another process k has obtained a decision value for the new round, to set $D.j.k$ to $d.k$. $MRO4$ is executed if j needs to correct its decision value to the majority of the decision values of its neighbors in the current round. Action $MRO5$ is the restricted version of action $RO2$ where process j outputs its decision only after the detection is complete.

Finally, to model Byzantine execution of j, we introduce action $MRO6$ that is executed only if $b.j$ is true: $MRO6$ lets j change $D.j.j$ and, thereby, affect the value read by process k when k executes $MRO3$. $MRO6$ also lets j obtain arbitrary values for $D.j.k$ and, thereby, affect the value of $d.j$ when j executes $MRO4$. Thus, the six actions of MRO are as follows:

MRO1 :: *RO1*

MRO2 :: $d.j \neq \perp \ \wedge SN.j.j = sn.j \wedge \ compl.j \longrightarrow D.j.j, SN.j.j := d.j, SN.j.j \oplus 1$

MRO3 :: $SN.j.k \oplus 1 = SN.k.k \longrightarrow D.j.k, SN.j.k := D.k.k, SN.k.k$

MRO4 :: $d.j \neq \perp \ \wedge \ majdefined.j \ \wedge \ d.j \neq maj.j \longrightarrow d.j := maj.j$

MRO5 :: $d.j \neq \perp \ \wedge \ majdefined.j \ \wedge \ d.j = maj.j \longrightarrow output\_decision(d.j) \ ; d.j, sn.j := \perp, \ sn.j \oplus 1$

MRO6 :: $b.j \longrightarrow D.j.j := 0 \mid 1;$
$(\| \ k : SN.j.k \oplus 1 = SN.k.k : D.j.k, SN.j.k := 0 \mid 1, SN.k.k)$

---

where,
$$compl.j \equiv (\forall k :: SN.j.j = SN.k.j)$$
$$majdefined.j \equiv compl.j \ \wedge \ (\forall k :: SN.j.j = SN.j.k) \ \wedge \ (sn.j \neq SN.j.j)$$
$$maj.j = (majority \ k :: D.j.k)$$

Fault Actions. If the number of Byzantine processes is less than 1, the fault actions make some process Byzantine. Thus, letting l and m range over all processes, the fault actions are:

$$|\{l : b.l\}| < 1 \quad \longrightarrow \quad b.m := true$$

Theorem 7.2. Program MR is masking fault-tolerant to Byzantine faults.

Proof. In accordance with the design issues discussed in Chapter 6, this proof consists of two parts: (1) In the presence of Byzantine faults, the detectors and correctors ensure that the masking tolerance specification of repetitive agreement is satisfied, and (2) the detectors and correctors do not interfere with R in the absence of faults.

(1) Observe that in the start state of the round —where the sequence numbers of all processes are identical, i.e. $(\forall j, k :: sn.j = SN.j.k = sn.g)$, and no non-Byzantine process has read the decision of g, i.e. $(\forall j : \neg b.j : d.j = \bot)$— only action $RG1$ in g can be executed. Thereafter, the only action enabled at each non-Byzantine process j is $RO1$.

After j executes $RO1$, j can only execute the actions of the detector and corrector. Moreover, j cannot execute $RO2$ until the detector and corrector actions at j terminate in that round.

The detector action $MRO2$ executes first and increments $SN.j.j$. By the same token, action $MRO2$ in process k increments $SN.k.k$. Subsequently, action $MRO3$ at process j can execute to update $SN.k.j$ and $D.k.j$. Note that if k is non-Byzantine, $D.j.k$ is the same as $v.k$, which in turn is equal to $d.g$ if g is also non-Byzantine. It follows that eventually $majdefined.j \wedge maj.j = cordec$ holds, and the action $MRO4$ at j can subsequently ensure that $d.j = maj.j$ before it terminates in that round.

After the detector and corrector actions are disabled in that round, j can only execute action $RO2$. It follows that, in the presence of a Byzantine fault, each round of the system computation satisfies Validity and Agreement.

(2) Observe that, in the absence of a Byzantine fault, the detector and corrector actions eventually satisfy $majdefined.j \wedge d.j = maj.j$ in each round and then terminate. Therefore, the detectors and correctors do not interfere with R in the absence of a Byzantine fault. $\square$

7.2.3 Adding Stabilizing Fault-Tolerance to Transient and Byzantine Faults

Despite the addition of the masking component to the program R, the resulting program MR is not yet stabilizing tolerant to transient and Byzantine failures. For example, MR deadlocks if its state is transiently corrupted into one where some non-general process j incorrectly believes that it has completed its last round, i.e., $d.j = \perp \ \wedge \ SN.j.j \neq sn.j$. It therefore suffices to add a corrector to MR that ensures stabilizing tolerance to transient and Byzantine failures while preserving the masking tolerance to Byzantine failure.

Towards designing the corrector, we observe that in the absence of transient faults the following state predicates are invariantly true of MR: (i) whenever $d.j$ is set to $\perp$, by executing action $MRO5$, j increments $sn.j$, thus satisfying $SN.j.j = sn.j$; and (ii) whenever j sets $sn.j$ to be equal to $sn.g$, by executing action $MRO5$, $d.j$ is the same as $\perp$. In the presence of transient faults, however, these two state predicates may be violated. Therefore, to add stabilizing tolerance, we need to guarantee that these two state predicates are corrected.

To this end, we add two corresponding correction actions, namely $MRO7$ and $MRO8$, to the non-general processes. Action $MRO7$ is executed when $d.j$ is $\perp$ and $SN.j.j$ is different from $sn.j$, and it sets $SN.j.j$ to be equal to $sn.j$. Action $MRO8$ is executed when $sn.j$ is the same as $sn.g$ but $d.j$ is different from $\perp$, and it sets $d.j$ to be equal to $\perp$. With the addition of this corrector to MR, we get a multitolerant program SMR.

$$MRO7 :: \quad d.j = \perp \ \wedge \ SN.j.j \neq sn.j \quad \longrightarrow \quad SN.j.j := sn.j$$

$$MRO8 :: \quad d.j \neq \perp \ \wedge \ sn.j = sn.g \quad \longrightarrow \quad d.j := \perp$$

Fault Actions. In addition to the Byzantine fault actions, we now consider the transient state corruption fault actions (let j and k range over non-general processes):

$$true \quad \longrightarrow \quad d.g, sn.g := 0|1, 0|1$$

$$true \quad \longrightarrow \quad d.j, sn.j := 0|1, 0|1$$

$$true \quad \longrightarrow \quad SN.j.k, D.j.k := 0|1, 0|1$$

Theorem 7.3. Program SMR is masking tolerant to Byzantine faults and stabilizing tolerant to transient and Byzantine faults.

Proof. This proof consists of three parts: (1) The added corrector offers stabilizing tolerance to MR in the presence of transient and Byzantine faults, (2) it does not interfere with the execution of MR in the absence of faults, and (3) it does not interfere with the masking tolerance of MR in the presence of Byzantine faults only.

(1) Observe that execution of the added corrector in isolation ensures that eventually the program reaches a state where the state predicate S holds, where

$$S \;=\; (d.j\!=\!\perp \;\Rightarrow\; SN.j.j\!=\!sn.j) \;\wedge\; (sn.j\!=\!sn.g \;\;\Rightarrow\;\; d.j\!=\!\perp)\ .$$

Since both disjuncts in S are preserved by the execution of all actions in MR, program MR does not interfere with the correction of S by corrector. We now show that after SMR reaches a state from where S holds at most one round is executed incorrectly.

I) Starting from any state where $S \wedge sn.j \neq sn.g \wedge d.j \neq \perp$ holds, eventually $SN.j.j \neq sn.j$ holds. Consider two cases on the starting state: (a) it has a process k such that $SN.k.j \neq SN.j.j$ holds or (b) it has no such k. In case (a), $MRO3$ is continuously enabled at k and its execution satisfies $SN.k.j = SN.j.j$. After all such k execute $MRO3$, the resulting state satisfies case (b). In case (b), $MRO2$ continuously enabled at j and its execution satisfies $SN.j.j \neq sn.j$. Moreover, if $SN.j.j \neq sn.j$ holds, it continues to hold until j executes $MRO5$.

II) Starting from a state where $SN.j.j \neq sn.j$ holds, if $SN.j.k \neq SN.j.j$ also holds then eventually $SN.j.k = SN.j.j$ is satisfied. Again, consider two cases: (a) $sn.j \neq sn.k$ and (b) $sn.j = sn.k$. In case (a), $sn.k = sn.g$. Therefore, from S, $SN.k.k = sn.k$ holds. If $SN.j.k \neq SN.j.j$, $SN.j.j \neq sn.j$, $sn.j \neq sn.k$, and $sn.k = SN.k.k$ all hold, it follows that $SN.j.k \neq SN.k.k$ holds. Therefore, j can execute $MRO3$ to satisfy $SN.j.k = SN.j.j$. In case (b), $sn.k \neq sn.g$ and, hence, either $d.k \neq \perp$ holds or k can execute $MRO1$ to satisfy it. From the previous paragraph, eventually $SN.k.k \neq sn.k$ is satisfied. Again, in this state, $SN.j.k \neq SN.k.k$ holds. Therefore, k can execute $MRO3$ to satisfy $SN.j.k = SN.j.j$.

III) Finally, starting from a state where $SN.j.j \neq sn.j$ holds, k can execute $MRO3$ to satisfy $SN.k.j = SN.j.j$. And, if $SN.j.j \neq sn.j$, $SN.j.k = SN.j.j$, and $SN.k.j =$

$SN.j.j$ are satisfied, $majdefined.j$ is also satisfied. Thus, j will eventually execute $MRO5$.

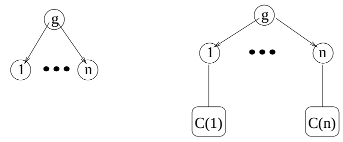
Now, observe that between any two executions of $MRG1$, each non-general process j executes $MRO5$, after which $SN.j.k$ is the same as $SN.j.j$ which in turn is equal to $sn.g$. Thus, all sequence numbers are equal and the decisions of all non-general processes are $\perp$ before the second execution of $MRG1$. As shown in the proof of masking tolerance, starting from such a state, Validity and Agreement are satisfied.

(2) Observe that, in the absence of faults, S continues to be preserved, and hence the added corrector is never executed. Therefore, the added corrector does not interfere with MR in the absence of faults.

(3) As in part (2), observe that, in the presence of Byzantine faults only, S continues to be preserved and, hence, the stabilizing component is never executed. Therefore, the added corrector does not interfere with MR in the presence of Byzantine faults. $\square$

7.2.4 Extension to Tolerate Multiple Byzantine Faults

To motivate the generalization of SMR to handle t Byzantine failures given n non-general processes, where $n \geq 3t$, let us take a closer look at how program SMR is derived from R. To design SMR, we added to each process j a set of components $C(j)$, which consists of a detector and a corrector (see Figure 7.1).



(a) : Program R

(b) : Program SMR

g : Actions $MRG1-2$
j : Actions $RO1-2$ and $MRO6$
$C(j)$: Actions $MRO2-4$, $MRO7-8$, and the restriction of $MRO5$

Figure 7.1: Structure of R and SMR

Note that action $MRO2$ is of the form of $RG1$ and that action $MRO3$ is of the form $RO1$ followed by $RO2$. ($D.j.j$ and $SN.j.j$ play the role of $d.g$ and $sn.g$ and $D.j.k$ and $SN.j.k$ play the role of the d values at the non-general processes.) In other words, $C(j)$ itself contains a repetitive agreement program !

With this insight, we are now ready to generalize program SMR to handle the multiple Byzantine faults, based on an idea that is essentially due to Lamport, Shostak, and Pease [47]. (Our generalization, of course, is distinguished by being multitolerant.) Let g denote the general process, X denote the set of non-general processes, t denote the maximum number of Byzantine processes. We define $SMR(g, X, t) = BYZ(g, X, t, \langle\rangle)$, where

$$\begin{array}{lcl} BYZ(g,X,t,s) & = & inp(g,X,t,s) \wedge MRG1(g,X,t,s) \quad [\!] \quad MRG2(g,X,t,s) \\ & & [\!] \\ & & ([\!] \ j : j \in X : \quad RO1(j,X,t,s) \quad [\!] \quad w(j,X,t,s) \wedge RO2(j,X,t,s) \quad [\!] \\ & & \qquad\qquad\qquad MRO6(j,X,t,s) \ ) \\ & & [\!] \\ & & ([\!] \ j : j \in X : \quad C(j,X,t,s \circ g) \ ) \end{array}$$

and

$$\begin{array}{lcll} inp(g,X,t,s) & = & d.(last(s), X \cup \{g\}, t{+}1, trlast(s)) \neq \bot \quad \wedge & \\ & & sn.(g,X,t,s) \ = \ sn.(last(s), X \cup \{g\}, t{+}1, trlast(s)) & \text{if } s \neq \langle\rangle \\ & = & new\_decision() & \text{otherwise} \\ \\ w(j,X,t,s) & = & majdefined.(j,X,t,s) \quad \wedge \quad d.(j,X,t,s) = maj.(j,X,t,s) & \text{if } t > 0 \\ & = & true & \text{otherwise} \\ \\ C(j,X,t,s) & = & MRO4(j,X,t,trlast(s)) \quad [\!] \quad MRO7(j,X,t,trlast(s)) \quad [\!] & \\ & & MRO8(j,X,t,trlast(s)) \quad [\!] \quad BYZ(j, X - \{j\}, t{-}1, s) & \text{if } t > 0 \\ & = & \text{the empty program} & \text{otherwise} \end{array}$$

Here s is a sequence ; $last(s)$ denotes the last element of s ; $trlast(s)$ denotes the sequence obtained by omitting the last element of s ; $s \circ j$ denotes the sequence obtained by appending j to s ; and action ac in program j is modified as follows:

- j is replaced with the quadruple (j, X, t, s)

- The quantification over k in *compl* is over the set

$$\{ \ (k, X - \{j\}, t-1, s \circ j) \ : \ k \in (X - \{j\}) \ \} \ \cup \ \{ \ (j, X, t, s) \ \}$$

- The quantification over k in *majdefined* and *maj* is over the set

$$\{ \ (j, X-\{k\}, t-1, s \circ k) \ : \ k \in (X - \{j\}) \ \} \ \cup \ \{ \ (j, X, t, s) \ \}$$

- If s is nonempty, the *output_decision* is assigned to the variable

$$D.(j, X, t, s).(j, X \cup last(s), t+1, trlast(s))$$

Observe that if the definition of $SMR(g, X, t)$ is instantiated with $t = 0$, the resulting program is R. And, if the definition is instantiated with $t=1$, the resulting program is SMR (with the previously noted exception that action $MRO3$ in j of SMR is implemented by $RO1$ and $RO2$ in the bottommost instantiation of BYZ, namely $BYZ(j, X - \{j\}, 0, \langle g \rangle)$).

Program $SMR(g, X, t)$ is multitolerant, i.e., it is masking tolerant to Byzantine faults and stabilizing tolerant to transient and Byzantine faults. We note that the structure of the proof of stabilization is the same as the proof for SMR: upon starting from any state, the program reaches a state where S holds; subsequently, g is guaranteed to start a new round infinitely often; and when g starts the $(t+1)-th$ round, the resulting computation satisfies Validity and Agreement. The proof of masking tolerance is similar to the one in [47].

7.3 Alternative Tolerances to Byzantine Failures

In the presence of Byzantine failures alone, SMR satisfies the specification of repetitive agreement in *each* round.

The goal of this section is to point out that every repetitive agreement program may not satisfy such a strong property. Zhao and Bastani [66] have presented a program that is nonmasking tolerant to Byzantine failures, i.e., that could violate the specification in some finite number of rounds only. In this section, we further show that even if a program is stabilizing tolerant to transient and Byzantine faults, it may be merely nonmasking tolerant to Byzantine faults. Our program is composed from SMR and a nonmasking tolerant program outlined below. Again, for simplicity, we consider the special case where there are four processes and at most one is Byzantine.

In our nonmasking tolerant program, each non-general process j chooses a "parent" of j that is initially g. In each round, j receives the decision value of its parent and outputs that value as the decision of j. In parallel, j obtains the decision value of g and forwards it to other non-general processes. If the values that j receives from g and the other two processes are not all identical, j is allowed to change its parent, so that it will output a correct decision in the following rounds, as follows.

Let j, k, and l be the three non-general processes. We consider two cases: (1) g is Byzantine and (2) g is non-Byzantine. Case (1): If g sends the value B to l and $B \oplus 1$ to j and the remaining process k, j and k will suspect that l or g is Byzantine, and l will know that g is Byzantine and that j and k are non-Byzantine. Without loss of generality, let the id of j be greater than that of k. We let both j and l change their parent to k (to avoid forming a cycle in the parent relation, k retains g as its parent). In all future rounds, j and l output the value received from k and, hence, the decision output by j, k, and l is identical. Case (2): Since the values sent by both j and k are the same, both j and k are non-Byzantine. Again, assuming the id of k is greater than that of j, it is safe to let j change its parent to k. In all future rounds, the decision output by j and k is the same as that output by g.

It follows that the nonmasking tolerant program executes only a finite number of rounds incorrectly in the presence of at most one Byzantine failure. This program is made stabilizing by adding SMR to it, as follows: Each process j is in one of two modes: *nonmasking* or *stabilizing*. It executes the nonmasking tolerant program when it is in the nonmasking mode, and it executes SMR when it is in the stabilizing mode. Further, it is allowed to change from the nonmasking mode to the stabilizing mode, but not vice versa. Observe that the nonmasking tolerant program satisfies the state predicate "if the parent of j is k for some $k \neq g$, then k is non-Byzantine, the parent of k is g, and the parent of l is k provided l is non-Byzantine". Hence, if j suspects that this predicate is violated, i.e., in some round j detects that either g or k is Byzantine, or the parent of k is not g, or the parent of l is not k, then j changes to the stabilizing mode and starts executing SMR. Moreover, whenever j detects that some other process is in the stabilizing mode, it changes its mode to stabilizing. Thus, if the composite program is perturbed to a state that is not reached by the nonmasking tolerant program, eventually all processes execute actions of SMR. It follows that the composite program is stabilizing tolerant but not masking tolerant to Byzantine failures.

7.4 Chapter Summary

In this chapter, we showed that the canonical fault-tolerant programs designed using replication and Schneider's state machine approach can be alternatively designed in terms of detectors and correctors. This result can be extended to other programs designed using these methods. The results in this chapter show that by designing a program in terms of detectors and correctors is at least as general as designing a fault-tolerant program using these methods.

Designing a program in terms of detectors and correctors also provides an insight that can be used to design alternative fault-tolerance properties. In case of repetitive Byzantine agreement, we used this insight to show how to design a program that is merely nonmasking fault-tolerant to Byzantine faults.

CHAPTER 8

DISTRIBUTED RESET : AN APPLICATION OF DETECTORS AND CORRECTORS

In this chapter, we show how detectors and correctors are used in the design of the first multitolerant distributed reset solution that uses bounded memory at each process. Intuitively, the problem of distributed reset [9] requires that a distributed system be reset to some given global state. We focus our attention on this as it is widely applible in the designing fault-tolerance in distributed programs.

Bounding the sequence numbers in the presence of fail-stop, repair, and transient faults is well-known to be difficult, even if we consider distributed resets that are only stabilizing fault-tolerant. For the case of stabilizing fault-tolerance, we recall a 1990 comment by Lamport and Lynch [46] that a solution for the distributed reset problem "using a finite number of identifiers would be quite useful, but we know of no such algorithm". A few bounded-space stabilizing solutions have been discovered since then [5, 9, 14–16, 29, 65]. However, these solutions are only stabilizing tolerant and allow distributed reset operations to complete incorrectly if faults occur during a reset operation. As discussed later this chapter, stabilizing tolerance is not ideal for fail-stop and repair of processes; the ideal fault-tolerance for these faults is masking.

Our solution provides masking fault-tolerance to fail-stop and repair of processes while preserving the stabilizing tolerance to transient faults.

This chapter is organized as follows: In Section 8.1, we define the problem of distributed reset and describe the challenges in the design of multitolerant distributed reset. In Section 8.2, we describe previous solutions to the distributed reset problem. In Section 8.3, we give an outline of our solution. In Section 8.4, we develop the fault-intolerant program for distributed reset. In Section 8.5, we transform the fault-intolerant program to add masking tolerance to fail-stops and repairs. In Section 8.6, we transform the masking reset program to add stabilizing tolerance to transient faults. In Section 8.7, we describe how this reset program can be used to design multitolerant application programs.

8.1 Problem Statement

Distributed reset. A distributed reset program consists of two modules at each process: an application module and a reset module. The application module can initiate a reset operation to reset the program to any given global state. Upon initiation of the reset operation, the reset module resets the state of the application to a state that is reachable from the given global state, and then informs the application module that the reset operation is complete. In other words, each reset operation satisfies the following two properties.

- Every reset operation is *non-premature*, i.e., if the reset operation completes, the program state is reachable from the given global state.

- Every reset operation *completes eventually*, i.e., if an application module at a process initiates a reset operation, eventually the reset module at that process informs the application module that the reset operation is complete.

The definition captures the intuition that resetting the distributed program to exactly the given global state is not necessarily practical nor desirable, since that would require freezing the distributed program while the processes are individually reset. The definition therefore allows the program computation to proceed concurrently with the reset, to any extent that does not interfere with the correctness of the reset.

Observe that to reset the program state to the given global state, the application module at every process needs to be reset. Furthermore, if two application modules communicate only if both have been reset in the same reset operation and all processes have been reset in that reset operation, the current program state is reachable from the corresponding global state.

Note that we have intentionally omitted how the application module chooses the global reset state parameter for each reset operation, but it is worthwhile to point out that these global states may be determined dynamically, say by a checkpointing program.

Masking tolerance. The ideal fault-tolerance for distributed reset is masking tolerance. A reset program is masking tolerant to a fault-class if every reset operation is correct in the presence of the faults in that class. In other words, the safety of distributed reset (i.e., that every reset operation is non-premature) is satisfied before, during, and after the occurrences of faults. And, the liveness of distributed reset (i.e., that every reset operation completes) is satisfied after fault occurrences stop.

Consider a fault that fail-stops a process and repairs it instantaneously. It is possible to design masking tolerance to this fault. In fact, even if the fault only fail-stops processes or only repairs processes, it is still possible to design masking tolerance to the fault, with respect to the processes that are up throughout the reset operation. We therefore redefine a *premature* reset operation as follows: a reset operation is premature if its initiator completes it without resetting the state of all processes that are up throughout the reset operation. In the rest of the chapter, we use this refined definition of premature reset.

Stabilizing tolerance. An alternative fault-tolerance for distributed reset is stabilizing tolerance. A reset program is stabilizing tolerant to a fault-class if starting from any arbitrary state, eventually the program reaches a state from where every reset operation is correct, i.e., the safety and liveness of distributed reset are satisfied.

Stabilizing tolerance is ideal when an arbitrary state may be reached in the presence of faults. Arbitrary states may be reached, for example, in the presence of fail-stops, repairs, message loss, as demonstrated by Jayaram and Varghese [35]. In such cases, masking tolerance cannot be designed as the fault itself may perturb the program to a state where the reset operation has completed prematurely.

Since arbitrary states can be reached in the presence of arbitrary transient faults, the ideal tolerance to transient faults is stabilizing tolerance. From the definition of stabilizing tolerance, if the program is stabilizing tolerant to transient faults, it is also stabilizing tolerant to fail-stops and repairs. However, this is not the ideal tolerance to fail-stops and repairs.

Multitolerant Reset. As motivated above, the best suited tolerance to fail-stop and repair faults is masking and the best suited tolerance to transient faults is stabilizing. We therefore design a multitolerant program that offers for each of these two classes their best suited tolerance. (Note that by being stabilizing tolerant to transient faults, our program is also stabilizing tolerant to fail-stops and repairs. Therefore, it is both masking and stabilizing tolerant to fail-stops and repairs.)

It is important to emphasize that our reset program is not just a stabilizing program. A reset program that is only stabilizing tolerant to fail-stops and repairs permits a reset operation to complete incorrectly in the presence of fail-stops and repairs. (All existing stabilizing reset programs in fact do so.) By way of contrast, our program ensures that in the presence of fail-stops and repairs, every reset operation is correct. In fact, as discussed below, designing a multitolerant reset program is significantly more complex compared to designing just a stabilizing reset program.

In principle, to design a multitolerant reset program, the initiator of the reset operation needs to "detect" whether all processes have been reset in the current operation. For the program to be masking tolerant to fail-stops and repairs, this "detection" must itself be masking tolerant to fail-stops and repairs. Also, for the program to be stabilizing, this detection must itself stabilize if perturbed to an arbitrary state. Thus, the design of the multitolerant reset program involves the design of a multitolerant detector. (Note that in the design of masking fault-tolerance, we also need to design a multitolerant corrector. It turns out, however, that the design of this corrector is easier than that of the detector.)

Also, adding such a multitolerant detector to a stabilizing reset program is not sufficient for the design of a multitolerant reset program. Since the detector and the actions of the stabilizing program execute concurrently, the detector may interfere with the stabilizing program, making it non-stabilizing, and the stabilizing actions may interfere with the detector, causing incorrect detection. Thus, to design a multitolerant program, we also need to ensure interference-freedom between the detector and the actions of the stabilizing program.

Both these problems are further complicated if the program uses bounded memory. In the masking reset program, the added detector needs to check that all processes are reset in the current reset operation. To implement this detector, each process detects whether all its neighbors have reset their states in the current reset operation. Using bounded sequence numbers to distinguish between different reset operations is tricky since it is still possible that multiple processes have the same sequence number even if they were last reset in different reset operations.

(Notice that sequence numbers for old reset operations may exist in the system, since some communication channels may be slower than others; communication channels may allow messages to be reordered; processes may repair with incorrect sequence numbers; or transient faults may arbitrarily corrupt the sequence numbers. In Section 8.5.2, we illustrate this with an example where multiple processes end up with the same sequence number even though they have been reset in different reset operations.)

System assumptions. A distributed system consists of processes, each with a unique integer identifier, and bidirectional channels, each connecting a unique pair of processes. At any time, a process is either *up* or *down*. Only up processes execute their actions.

Remark. Henceforth, we use the term "process" and "up process" interchangibly. Also, when the context is clear, we use "process" to mean the "reset module of the process".

8.2 Related Work

To our knowledge, this is the first bounded-memory multitolerant distributed reset program. In fact, we are not aware of bounded-memory distributed reset program that is masking tolerant to fail-stops and repairs. We note that Afek and Gafni [2] have shown a masking tolerant solution under the severe assumption that processes do not lose their memory if they fail. They do, however, allow channels to fail and the messages sent on those channels to be lost. Their program is not stabilizing tolerant.

While little work has been done on bounded-memory masking tolerant resets, bounded-memory stabilizing tolerant resets have received more attention [5, 9, 14–16, 29, 65]. All of these programs are stabilizing tolerant to fail-stops and repairs, but they are not masking tolerant to them. Specifically, in the presence of fail-stops and repairs, they allow premature completion of distributed resets.

Masuzawa has presented a reset program [50] that tolerates two fault-classes: transient faults and undetectable crash faults. His solution assumes that at most M processes fail undetectably for some fixed M such that the process graph is $(M+1)$-connected. While his solution ensures that in a reset operation eventually all processes are reset, it permits premature completion of a reset operation. Also, his solution uses unbounded memory.

8.3 Outline of the Solution

Recall that in accordance with our approach to designing multitolerance in Chapter 6, we first design a fault-intolerant reset program; then transform this program to add masking tolerance to fail-stops and repairs; and finally add stabilizing tolerance to this program, while ensuring that the masking tolerance to fail-stops and repairs is preserved. In this section, we outline the structure of each of these three programs and their proofs of correctness.

Fault-intolerant program. We use a variation of the diffusing computation program. In particular, we use a tree that spans all up processes in the system, and perform the reset diffusing computation over the tree edges only. The root of the tree initiates the diffusing computation of each reset. When a process receives a diffusing computation from its parent in the tree, it resets its local state, and propagates the diffusing computation to its children in the tree. When all descendents of process j have reset their local states, j completes its diffusing computation. Thus, when the root completes the diffusing computation, all processes have reset their local states.

Adding masking tolerance. In the presence of fail-stops and repairs, the tree may become partitioned. Hence, the diffusing computation initiated by a root process may not reach all processes in the system.

To add masking tolerance, it suffices that we ensure the following two conditions for every distributed reset: (1) eventually, the local states of all processes are reset and (2) in the interim, no diffusing computation completes incorrectly.

To achieve (1), we add a corrector that ensures that eventually the tree spanning all up processes is restored, so that a diffusing computation can reach all up processes. To this end, we reuse a nonmasking tolerant tree program due to Arora [7] that, in the presence of fail-stops and repairs, maintains the graph of the parent relation of all up processes to always be a forest and, when faults stop occurring, restores the graph to be a spanning tree. The details of this program are given in Section 8.5.1.

To achieve (2), we add a detector which restricts the completion of the reset computation to occur only when the root can detect that all processes participated in the current diffusing computation. Suppose that some processes have not participated when the root completes: since the up processes remain connected in the presence of fail-stops and repairs, it follows that there exists at least one pair of neighboring processes j and k such that j participated in the diffusing computation but k did not. To detect such pairs, a "result" is associated with each diffusing computation: process j completes a diffusing computation with the result true only if all its neighbors have propagated the diffusing computation and, hence, reset their local states. Otherwise, j completes the diffusing computation with the result false. The result is propagated in the completion wave of the diffusing computation. In particular, if j has completed a diffusing computation with the result false, then the parent of j completes that diffusing computation with the result false, and so on. Also, if j fails or moves to a different tree, then the (old) parent of j cannot always determine the result of j. Hence, when j fails or moves to a different tree, the parent of j completes with the result false. It follows that when the root completes the diffusing computation with the result true, all processes have participated in the diffusing computation.

Finally, if a root completes a diffusing computation with the result false, it starts a new diffusing computation. Since the nonmasking tolerant tree program eventually spans a tree over the up processes, in any diffusing computation initiated after the tree is constructed and during which no failures occur, the distributed reset completes correctly.

Adding stabilizing tolerance. To design stabilizing tolerance to the program, we add a corrector that ensures that eventually the program reaches a state from where subsequent resets will complete correctly. In particular, in the presence of transient faults, the graph of parent relation may form cycles and, so, we add actions to detect and eliminate cycles in the graph of the parent relation. While adding stabilizing tolerance to transient faults, we preserve the masking tolerance to fail-stops and repairs, by ensuring that the newly added corrector and the actions of the masking program do not interfere. The details of this program are given in Section 8.6.

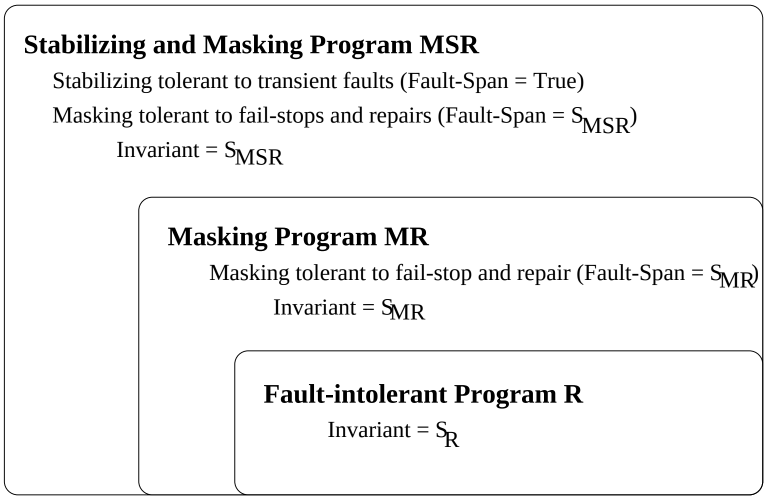


Figure 8.1: Structure of Our Multitolerant Distributed Reset Program

8.4 Fault-Intolerant Distributed Reset

In this section, we describe a straightforward distributed reset program in terms of a diffusing computation over a rooted spanning tree. For simplicity, we assume that only the root process of the tree initiates distributed resets.

When the root process initiates a distributed reset, it marks its state as *reset*, resets its local state, and propagates a reset wave to its children in the tree. Likewise, when a process j receives a reset wave from its parent, j marks its state as *reset*, resets its local state, and propagates the reset wave to its children. We refer to these propagations as the propagation of the reset wave.

When a leaf process j propagates a reset wave, j completes the reset wave, marks its state as *normal*, and responds to its parent. When all children of process j have responded, j completes the reset wave, and responds to its parent. We denote these completions as the completion of the reset wave.

It follows that when a process completes a reset wave, all its descendents have completed that reset wave. In particular, when the root process completes the reset wave, all processes have completed the reset wave, and the root process can declare that the distributed reset has been successfully completed.

Variables. As described above, every process j maintains the following variables:

- $st.j$: state of j; the state is *reset* if j is propagating a reset wave, otherwise, it is *normal*
- $sn.j$: sequence number of j; the sequence number of j is either 0 or 1
- $par.j$: parent of j; the parent of the root process is set to itself

Actions. As described above, program R consists of three actions at each process j. Action $R1$ lets j initiate a new reset wave, if j is a root process. Action $R2$ lets j propagate a reset wave, if $par.j$ is propagating a reset wave and the sequence number of j and $par.j$ are different. Action $R3$ lets j complete a reset wave, if j is in the reset state and all children of j have completed in that reset wave; if j is a root process, j also declares that the distributed reset is complete.

Formally, the actions of the program at process j are as follows: (Let $ch.j$ denote the set of children of j).

$R1 ::$ $\quad par.j\!=\!j \;\;\wedge\;\; st.j\!=\!normal \;\;\wedge\;\;$ {j needs to initiate a new reset wave}

$\longrightarrow\;\; st.j,\; sn.j := reset,\; sn.j\!\oplus\!1;$ { reset local state of j }

$R2 ::$ $\quad st(par.j)\!=\!reset \;\;\wedge\;\; sn.j\!\neq\! sn.(par.j)$

$\longrightarrow\;\; st.j,\; sn.j := reset,\; sn.(par.j);$ { reset local state of j }

$R3 ::$ $\quad st.j\!=\!reset \;\;\wedge\;\; (\forall k : k\!\in\! ch.j : sn.j\!=\!sn.k \;\wedge\; st.k\!\neq\! reset)$

$\longrightarrow\;\; st.j := normal;$ { if $par.j\!=\!j$ then declare reset complete }

Invariant. Observe that if both j and $par.j$ are propagating a reset wave then they have the same sequence number. Also, if $par.j$ is in the normal state then j is also in normal state and has the same sequence number as $par.j$. Hence, the predicate $GD = (\forall j :: Gd.j)$ is in the invariant of the program, where

$$\begin{aligned} Gd.j = {} & ((st.(par.j)\!=\!reset \;\wedge\; st.j\!=\!reset) \;\;\Rightarrow\;\; sn.j\!=\!sn.(par.j)) \quad \wedge \\ & ((st.(par.j)\!\neq\! reset) \;\;\Rightarrow\;\; (st.j\!\neq\! reset \;\wedge\; sn.j\!=\!sn.(par.j))) \end{aligned}$$

Moreover, the invariant of the program is

$$S_R \;=\; GD \;\wedge\; \text{graph of the parent relation forms a tree}$$

Remark. Although, for simplicity, we assumed that the reset request initiated by the root process, our program can be generalized to let any process initiate a reset wave. Towards this end, any request made by a process is propagated towards the root, which then performs the reset. This extension is identical to the one given in [9] and, hence, is omitted.

8.5 Masking Fault-Tolerant Distributed Reset

In this section, we transform the fault-intolerant program of Section 8.4 to add masking tolerance to fail-stop and repair faults. As described in Section 8.3, to add masking tolerance to fail-stop and repair faults, it suffices that

- After faults stop occurring, the program eventually reaches a state from where no distributed reset ever completes incorrectly.
- When a root process declares that a distributed reset has completed, all up processes have participated in that reset wave.

As discussed in Chapter 5, we design the masking fault-tolerant program in two stages, *via* nonmasking fault-tolerance. More specifically, in the first stage, we add correctors to obtain a nonmasking fault-tolerant program. The nonmasking fault-tolerant program ensures that in the presence of fail-stop and repair of processes, the program eventually reaches a state from where no distributed reset will complete incorrectly. In the second stage, we transform the nonmasking fault-tolerant program into one that is masking fault-tolerant, by adding detectors for the actions of the nonmasking fault-tolerant program so that a process declares completion of a distributed reset only if all up processes have participated in the last reset wave.

Below, in Section 8.5.1, we design the nonmasking program to tolerate fail-stop and repair faults. Then, in Section 8.5.2, we restrict the actions of the nonmasking program, so that the resulting program is masking fault-tolerant.

8.5.1 Nonmasking Fault-tolerant Distributed Reset

Observe that if the program reaches a state where the invariant S_R holds, then starting from such a state no distributed reset will complete incorrectly. To design a nonmasking reset program for fail-stop and repair faults, we add a corrector that restores the program to a state where S_R holds. This corrector consists of actions that (re)construct the rooted spanning tree and restore $Gd.j$ for every process.

To construct a spanning tree, we use Arora's program for tree maintenance [7], which allows fail-stops and repairs to yield states where there are multiple, possibly unrooted, trees. We briefly describe, next, how the program deals with multiple trees and unrooted trees, and thereby eventually converges to a state where there is exactly one tree spanning all processes. We refer the reader to [7] for the proof of the nonmasking program.

The program merges multiple trees such that no cycles are formed: Each process maintains a variable $root.j$ to denote the process that j believes to be the root. When process j observes a neighbor k such that $root.k$ is greater than $root.j$, j merges in the tree of k, by setting $root.j$ to $root.k$ and $par.j$ to k. By merging thus, cycles are not formed and the root value of each process remains at most the root value of its parent. This process continues until no merge action is enabled, at which point, all processes have the same *root* value.

The program has actions to let each process detect if it is in an unrooted tree. To detect whether a process is in an unrooted tree, each process j maintains a variable $col.j$ to denote the color of process j (which is either red or green). Whenever j detects that parent of j has failed, j sets $col.j$ to red, denoting that j is in an unrooted tree. This color is propagated from the tree root to the leaves so that all descendents of j

detect that they are in an unrooted tree, i.e., when a process l observes that parent of l has set its color to red, denoting that parent of l is in an unrooted tree, l sets $col.l$ to red. Finally, when a leaf process sets its color to red, it separates from the tree, forms a tree consisting only of itself, and sets its color to green denoting that it is no longer in an unrooted tree. Thus, Arora's tree maintenance program ensures that after faults stop occurring, the parent tree is (re)constructed.

To restore $Gd.j$ at every process, we proceed as follows: We ensure that if j and $par.j$ are in the same tree (i.e., if $root.j$ is the same as $root.(par.j)$ and their color is green), then $Gd.j$ is satisfied. Also, when j merges into the tree of k, j satisfies $Gd.j$ by copying the state and sequence number from k. It follows that in all stable states where the root values of all processes are equal, $Gd.j$ holds for all processes.

Variables. Every process j maintains the following variables:

- $col.j$: color of the process j ; the color of j is either green or red
- $root.j$: root of the process j ; the identifier of the process that j believes to be the root.

Actions. The nonmasking fault-tolerant program NR consists of six actions. The first three actions the actions of the fault-intolerant program. These actions are restricted to execute only in states where the tree being formed is consistent. This restriction ensures that the fault-intolerant program does not interfere with the corrector. The last three actions are the actions of the corrector.

The actions of the fault-intolerant program are as follows: Action $NR1$ is the initiation action; it is the same as action $R1$. Action $NR2$ is the propagation action; it is a restricted version of action $R2$, where j executes the action $R2$ only if the tree being formed is consistent with respect to j and $par.j$, i.e., if $col.(par.j)$ is green and their $root$ value is the same. Action $NR3$ is the completion action; it is a restricted version of $R3$, where j executes action $R3$ only if the tree being formed is consistent with respect to j and $ch.j$ is consistent, i.e., if $col.j$ is green and the $root$ value of the children of j is the same as $root.j$.

The actions of the corrector are as follows: Action $NR4$ deals with unrooted trees: If j detects that $par.j$ has failed or $col.(par.j)$ is red, j sets $col.j$ to red. Action $NR5$ lets a leaf process change its color from red to green: If j is a red leaf, then j separates from its tree and resets its color to green, thus forming a tree consisting only of itself. Action $NR6$ merges two trees: A process j merges into the tree of a neighboring process k when $root.k > root.j$. Upon merging, j sets $root.j$ to be equal to $root.k$, $par.j$ to be equal to k, and it copies the state and sequence number from k.

Formally, the actions of program NR at process j are as follows (Let $Adj.j$ denote the neighbors of j):

$NR1 ::$ $R1$

$NR2 ::$ $root.j = root.(par.j) \ \wedge\ col.(par.j) = green\ \wedge$

$st(par.j) = reset\ \wedge\ \ sn.j \neq sn.(par.j)$

$\longrightarrow$ $st.j,\ sn.j := reset,\ sn.(par.j);$ { reset local state of j }

$NR3 ::$ $st.j = reset\ \wedge\ col.j = green$

$(\forall k : k \in ch.j : root.j = root.k\ \wedge\ sn.j = sn.k\ \wedge\ st.k \neq reset)$

$\longrightarrow$ $st.j := normal;$ if $par.j = j$ then { declare reset complete }

$NR4 ::$ $col.j = green\ \wedge\ (par.j \notin Adj.j \cup \{j\}\ \vee\ col.(par.j) = red)$

$\longrightarrow$ $col.j := red$

$NR5 ::$ $col.j = red\ \wedge\ (\forall k : k \in Adj.j : par.k \neq j)$

$\longrightarrow$ $col.j,\ par.j,\ root.j := green,\ j,\ j$

$NR6 ::$ $k \in Adj.j\ \ \wedge\ \ root.j < root.k\ \ \wedge\ \ col.j = green\ \ \wedge\ \ col.k = green$

$\longrightarrow$ $par.j, root.j, := k, root.k;\ \ st.j, sn.j := st.k, sn.k$

Fault Actions. The fail-stop and repair actions are as follows:

$$
\begin{array}{llll}
Fail\!-\!stop :: & up.j & \longrightarrow & up.j := false \\
Repair :: & \neg up.j & \longrightarrow & up.j,\ par.j,\ root.j,\ col.j := true,\ j,\ j,\ red
\end{array}
$$

Fault-Span and Invariant. From Arora's tree maintenance program, we know that in the presence of fail-stops and repairs, the program actions preserve the acyclicity of the graph of the parent relation as well as the fact that the root value of each process is at most the root value of its parent. They also preserve the fact that if a process is colored red then its parent has failed or its parent is colored red. Thus, the predicate T_{TREE} is in the fault-span, where

T_{TREE} = the graph of the parent relation is a forest $\wedge$ $(\forall j : up.j : T1.j)$, where

$$
\begin{aligned}
T1.j = (&(col.j = red \;\Rightarrow\; (par.j \notin Adj.j \cup \{j\} \;\vee\; col(par.j) = red)) \;\wedge \\
&(par.j = j \;\Rightarrow\; root.j = j) \;\wedge\; (par.j \neq j \;\Rightarrow\; root.j > j) \;\wedge \\
&(par.j \in Adj.j \;\Rightarrow\; (root.j \leq root.(par.j) \;\vee\; col(par.j) = red)))
\end{aligned}
$$

Observe that $Gd.j$ is preserved when $root.j$ is same as $root.(par.j)$ and $col.(par.j)$ is green. Hence, the predicate $NGD = (\forall j :: NGd.j)$ is in the fault-span, where

$$NGd.j \;=\; (root.j = root.(par.j) \;\wedge\; col.(par.j) = green \;\;\Rightarrow\;\; Gd.j)$$

Thus, the fault-span of the program is

$$T_{NR} \;=\; T_{TREE} \;\wedge\; NGD$$

In a stable state, the color of all processes is green and the root value of all processes are identical. Thus, the invariant of the program S_{NR} is

$$S_{NR} \;=\; T_{NR} \;\wedge\; S_R \;\wedge\; (\forall j,k :: par.j \in Adj.j \;\wedge\; col.j = green \;\wedge\; root.j = root.k)$$

Theorem 8.1. Program NR is nonmasking tolerant to fail-stop and repair faults from S_{NR}.

8.5.2 Enhancing Tolerance to Masking

Program NR, being nonmasking tolerant, allows a reset operation to complete prematurely. To enhance the tolerance of NR to masking, we now add a detector that checks whether all processes participated in the given reset operation. In other words, we need to add a detector whose detection predicate is '$(\forall j$: j has been *up* throughout the diffusing computation : j has reset its state in the current diffusing computation$)$'. Recall from Section 8.3 that this detection is made possible by letting each process maintain for each reset wave a "result" that is true only if its neighbors have propagated that wave. The result of each process is propagated towards the initiator in the completion of the reset wave. In particular, if j has completed a reset wave with the result false, then the parent of j completes that reset wave with the result false, and so on. Also, if j fails or changes its tree, the (old) parent of j cannot always determine the result of j. Hence, when j fails or changes its tree, the parent of j completes the reset wave with the result false. It follows that when the root completes the reset wave with the result true, all processes have participated in the reset wave.

It remains to specify how a process detects whether its neighbors have propagated the current wave. One possibility is for j to detect whether the sequence numbers of all its neighbors are the same as that of j. Unfortunately, because the sequence numbers are bounded, such a detection is insufficient. We illustrate this by an example. In particular, we exhibit a program computation whereby: (1) even though j and l have the same sequence number, they are in different reset waves, and (2) even though j completes one reset wave after it changes its tree and j and l have the same sequence number, they are in different reset waves.

Let the initial state be as shown in Figure 8.2 (a). Process k is the root and the root value of all processes is k. Also, k has initiated a reset wave with sequence number 0, which all processes except l have propagated. The computation proceeds as follows:

- Process n fails.

- Process k completes its reset wave and initiates a new reset wave with sequence number 1 (Figure 8.2 (b)).

- Process j separates from the tree, changes its parent to k, and propagates the reset wave with sequence number 1 (Figure 8.2 (c)).

- Process j completes its reset wave (Figure 8.2 (d)). Observe that when j completes this reset wave, although the sequence number of l is the same as that of j, l has not propagated the current reset wave of k. Thus, (1) is satisfied.

- Process l propagates the reset wave with sequence number 0. Also, k completes its reset wave and initiates a new reset wave with sequence number 0 (Figure 8.2 (e)).

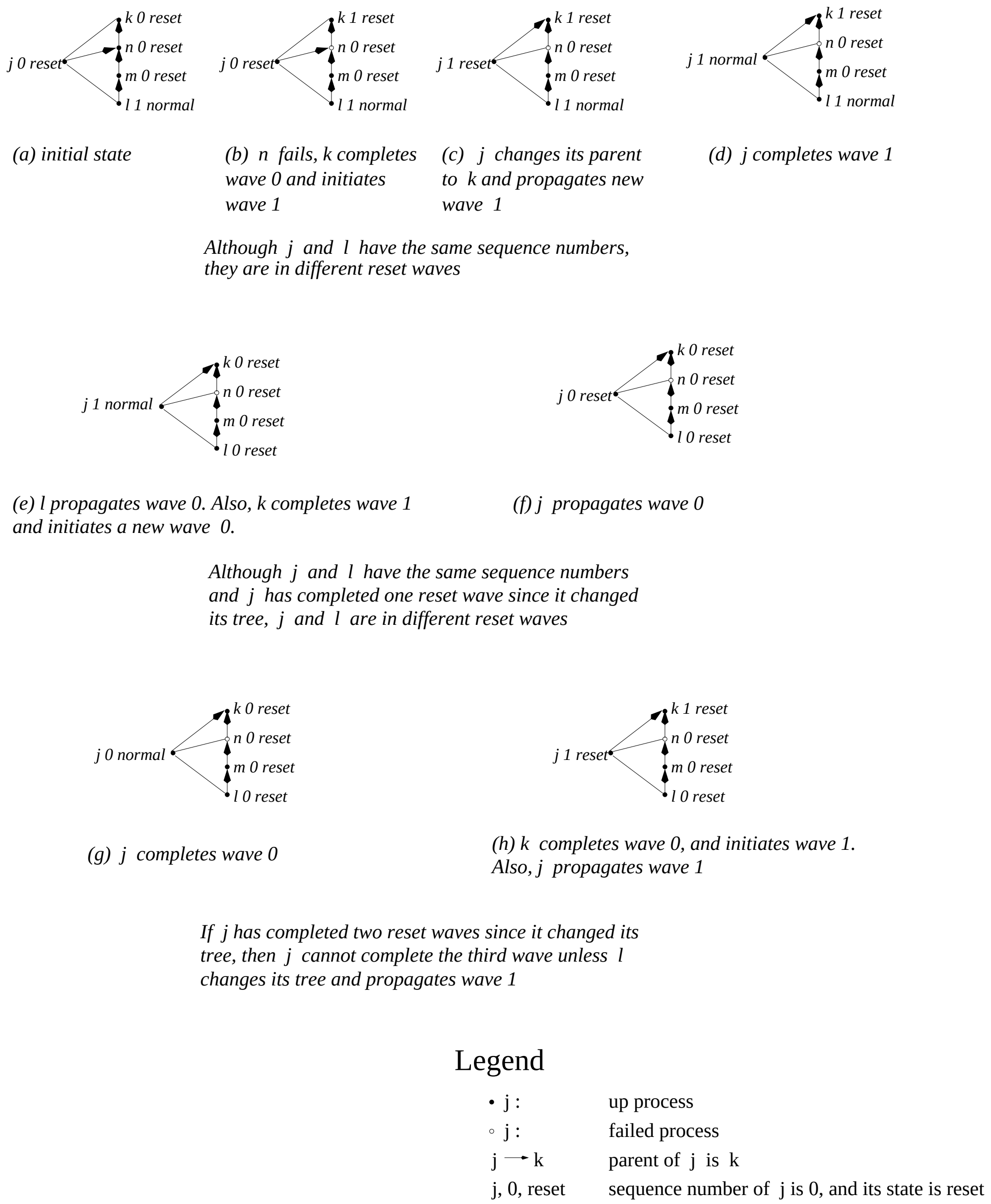


Figure 8.2: Detecting whether a neighbor is reset in the current reset operation

- Process j propagates the new reset wave of k with sequence number 0 (Figure 8.2 (f)). Observe that although j has completed one reset wave since it changed its tree and the sequence numbers of j and l are the same, they are in different reset waves. Thus, (2) is satisfied.

From the above computation, we observe that after j changes its tree, it cannot safely detect whether l is in the same wave as j during the subsequent two waves.

Fortunately, in the third wave after j changes its tree, j can safely detect whether l is in the same wave provided j and l do not change their tree in the interim. Returning to our example,

- Process j completes the reset wave with sequence number 0. Again, observe that when j completes this second reset wave, l is still propagating an old reset wave (Figure 8.2 (g)).

- Process k completes its reset wave and initiates a new reset wave with sequence number 1. Also, process j propagates this reset wave (Figure 8.2 (h)).

Observe that starting from the state in Figure 8.2 (h), j can complete its reset wave only when the sequence number of l is 1. However, since all ancestors of l have sequence number 0 and none of them is a root, l cannot change its sequence number to 1 unless l changes its tree. Also, in any tree that l joins and completes two reset waves, l cannot propagate the next reset wave unless k is an ancestor of l, in which case the reset wave propagated by l is the current reset wave of k. Thus, if j and l both complete at least two reset waves since they changed their respective trees, j can safely detect whether the reset wave propagated by l is the current current reset wave of k. Thus, the following lemma holds.

Lemma 8.2. (**Sufficiency condition for bounded-memory safety detection**) Let j, k, l be processes such that j and l are neighbors. Consider a state in S_{MR} where $root.j = root.l = root.k = k$ and k is an ancestor of j. In every computation of MR starting from this state, if j and l do not change their tree and complete two reset waves then the next reset wave propagated by j (respectively, l) is the current reset wave being propagated by k. $\square$

We use Lemma 8.2 to specify how j detects whether its neighbors have propagated the same reset wave as that of j, as follows. Process j maintains a ternary variable $new.j$, whose value is either 0, 1, or 2. When j changes its tree, $new.j$ is set to 2. When j completes a reset wave, if $new.j$ is nonzero, j decrements $new.j$ by 1. Thus, j detects that it has completed at least two reset waves since it last changed its tree by checking that $new.j$ is zero. And, j detects that all its neighbors have propagated at least two reset waves since they changed their tree by checking that their new values are zero.

Thus, new implements the "result" associated with the reset wave. The new value being zero is equivalent to the result being true, and the new value being nonzero is equivalent to the result being false. Therefore, if the new value of j or any neighbor of j is nonzero, j sets $new.(par.j)$ to a nonzero value to ensure that when $par.j$ completes the reset wave, $par.j$ sets $new.(par.(par.j))$ to a nonzero value, and so on. Thus, when the initiator completes the reset wave, its new value is nonzero, and the initiator concludes that the reset wave has completed incorrectly.

As mentioned in Section 8.3, if the initiator detects that the reset wave has completed incorrectly, it initiates a new reset wave. Also, if a process j fails, the parent of j cannot obtain the value of $new.j$. In that case, the parent of j aborts that reset wave by setting its new value to 2.

Variables. As described above, each process j additionally maintains the variable $new.j$ which is either 0 or 1 or 2.

Actions. The masking fault-tolerant program MR consists of the actions of the fault-intolerant program, the corrector from Section 8.5.1 and the detector. In this example, the detector does not have any new actions. However, it restricts the completion action in the fault-intolerant program so that a process checks the state of its neighbors before completion and updates the variable new accordingly. Since the added detector does not affect the tree reconstruction protocol of the corrector, the detector does not interfere with the corrector. To ensure that the corrector does not interfere with the detector, we require that whenever the corrector actions execute they abort the detector. This guarantees that as long as the corrector executes the root will not be able to successfully complete its detection. Since the corrector eventually terminates, the root will eventually be able to complete its detection.

Program MR consists of six actions for each process j. The first three actions implement the reset wave. Action $MR1$ is the initiation action; it is the same as action $NR1$. Action $MR2$ is the propagation action; it is the same as action $NR2$. Action $MR3$ is the completion action; it is a restricted version of $NR3$, where j executes action $NR3$ only if the predicate $(\forall l : l \in Adj.j : root.j = root.l \ \wedge \ sn.j = sn.l)$ holds. Also, j updates the variable new as described above. If the initiator completes a reset wave incorrectly, it initiates a new reset wave. Actions $MR4-6$ are the corrector actions. Whenever j executes these actions, it aborts the detection by setting $new.j$

and $new.(par.j)$ to 2. (Note that when j changes its tree, by executing action $NR5$ and $NR6$, the new value of the old parent is set to 2.)

Formally, the actions for the process j are as follows: (For simplicity, we let actions $MR3, MR5$ and $MR6$ at j update the value of $new.(par.j)$. As discussed in Section 8.8, this program can be refined so that j writes only the variables at process j.)

$MR1 ::\quad NR1$

$MR2 ::\quad NR2$

$$
\begin{array}{lll}
MR3 :: & & st.j = reset \ \wedge \ col.j = green \ \wedge \\
& & (\forall l : l \in Adj.j : root.l = root.j \ \wedge \ sn.j = sn.l) \ \wedge \\
& & (\forall l : l \in ch.j : st.l \neq reset) \\
& \longrightarrow & st.j := normal; \\
& & \text{if } (\exists l : l \in Adj.j \cup \{j\} : new.l > 0) \text{ then} \\
& & \quad \text{if } (par.j = j) \text{ then} \\
& & \qquad st.j,\ sn.j := reset, sn.j \oplus 1; \ \{ \text{ reset local state of } j \ \} \\
& & \quad \text{else } new.(par.j) := max(new.(par.j), 1) \\
& & \text{else if } (par.j = j) \text{ then } \{ \text{ declare reset complete } \}; \\
& & new.j := max(0, new.j - 1)
\end{array}
$$

$MR4 ::\quad NR4 \ \ \| \ \ new.j, new.(par.j) := 2, 2$

$MR5 ::\quad NR5 \ \ \| \ \ new.j, new.(par.j) := 2, 2$

$MR6 ::\quad NR6 \ \ \| \ \ new.j, new.(par.j) := 2, 2$

Fault Actions. As described above, when a process j fail-stops, $par.j$ needs to set $new.(par.j)$ to 2. Since a process does not know whether j is its child, we implement this by letting all neighbors of j to set their new value to 2. When a process j is repaired, j sets $new.j$ to 2. Formally, the actions are as follows:

$$
\begin{array}{llll}
Fail\!-\!stop :: & up.j & \longrightarrow & up.j := false;\ (\forall l : l \in Adj.j : new.l := 2) \\
Repair :: & \neg up.j & \longrightarrow & up.j, par.j, root.j, col.j, d.j, new.j := true, j, j, red, 0, 2
\end{array}
$$

Remark. In the fail-stop action, we have overloaded the operator $\forall$ to denote that the statement "$new.l := 2$" is executed at all processes in $Adj.j$. In Section 8.8, we observe that this parallel execution can be refined so that these statements are executed asynchronously.

Invariant. To characterize the predicates of the invariant, S_{MR}, we define:

$$
\begin{array}{lll}
X.j & = \{j\} & \text{if } par.j = j \ \vee\ par.j \notin Adj.j \ \vee\ col.j = red \ \vee\ root.j \neq root.(par.j) \\
 & \{j\} \cup X.(par.j) & \text{otherwise}
\end{array}
$$

$pc.j.k$ = set of processes to whom j propagated the current reset wave of k; thus if j is propagating the current reset wave initiated by k, and a child of j, say l, propagates this reset wave by executing action $MR2$, l is added to $pc.j.k$. If j has not propagated the current reset wave initiated by k, or if k is not a root process, $pc.j.k$ is the empty set

$$des.j.k = \{j\} \ \cup\ (\textstyle\bigcup l : l \in pc.j.k : des.l.k)$$

$$nbrs(des.j.k) = des.j.k \cup \{l : \exists j :: (l \in Adj.j \ \wedge\ j \in des.j.k)\}$$

$failed.k$ = set of processes that repaired after k started its reset wave

Intuitively, $X.j$ is the set of ancestors of j that have the same root value as j and their color is green, and $des.j.k$ is the set of processes that have propagated the current reset wave initiated by k *via* j. It follows that $des.k.k$ denotes the set of processes that have propagated the current reset wave of k.

When j completes a reset wave initiated by k, j detects whether all of its neighbors have propagated the current reset wave of k. It also detects that all processes that have propagated this reset wave *via* j (i.e., $des.j.k$) have completed their detections successfully. Thus, all neighbors of $des.j.k$ have propagated the current reset wave of k (i.e., they are in $des.k.k$) or they are newly repaired processes (they are in $failed.k$). Thus, $nbrs(des.j.k)$ is a subset of $(des.k.k \cup failed.k)$.

Observe that when a process j, such that $k \in X.j$, completes a reset wave, j detects the predicate $nbrs(des.j.k) \subseteq (des.k.k \cup failed.k)$ by checking $new.j$ and the new values of its neighbors. If j cannot conclude that $nbrs(des.j.k) \subseteq (des.k.k \cup failed.k)$ holds, j sets $new.(par.j)$ to a nonzero value. And, $new.(par.j)$ remains nonzero until $par.j$ completes this reset wave. Thus, the predicate $I1$ is in the invariant, where

$$
\begin{aligned}
I1 = (&k \in X.j \ \wedge \ new.j = 0 \ \wedge \ st.j \neq reset \ \wedge \\
&st.k = reset \ \wedge \ sn.j = sn.k) \qquad \Rightarrow nbrs(des.j.k) \subseteq (des.k.k \cup failed.k) \ \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad new.(par.j) > 0 \ \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (par.j \neq j \ \wedge \ st.(par.j) \neq reset)
\end{aligned}
$$

For the following discussion (predicates $I2-8$), we assume that j and l are two processes such that $k \in X.j$ (i.e., there exists a path from j to k in the graph of the parent relation, and all processes on this path are green and have the root value k) and $l \in Adj.j$ (i.e, l is a process adjacent to j).

When j completes a reset wave, $sn.j$ is the same as $sn.l$. The predicate $sn.j = sn.l$ is violated only if either $sn.j$ or $sn.l$ is updated. The variable $sn.j$ is updated only if j changes tree, or propagates a reset wave. If j (l) changes its tree, violating $sn.j = sn.l$, then $new.j$ ($new.l$) is set to 2. When l propagates a reset wave, from NGD, all processes in $X.l$ have the same sequence number. When j propagates a reset wave, j is in the reset state and it cannot complete the reset wave until $sn.j = sn.l$ is (re)satisfied. Thus, the predicate $I2$ is in the invariant, where

$$
\begin{aligned}
I2 = \quad (new.j \neq 2 \;\wedge\; new.l \neq 2) \quad \Rightarrow \quad & (st.j \neq reset \;\equiv\; sn.j = sn.l) \\
& \vee\; (\forall m : m \in X.l : sn.m = sn.l)
\end{aligned}
$$

When j completes a reset wave and decrements $new.j$ from 1 to 0, j is in the reset state and $sn.j$ is the same as $sn.l$. Thus, from the predicate $I2$, the second disjunct, $(\forall m : m \in X.l : sn.m = sn.l)$, must be true, i.e., all processes in $X.l$ have the same sequence number. Also, when j completes a reset wave, $sn.l$ is the same as $sn.j$, which in turn is equal to $sn.k$ (from NGD). Thus, the predicate $I3$ is in the invariant, where

$$
\begin{aligned}
I3 = \quad & (st.j \neq reset \;\wedge\; sn.j = sn.k \;\wedge \\
& new.j = 0 \;\wedge\; new.l \neq 2) \quad \Rightarrow \quad (\forall m : m \in X.l : sn.m = sn.k)
\end{aligned}
$$

Consider the case where k starts a new reset wave by changing $sn.k$: From $I3$, observe that all processes in $X.l$ have the same sequence number (in this case different from $sn.k$) unless $k \in X.l$. If $k \in X.l$, then trivially there exists a process in $X.l$ (namely k itself) which has propagated the current reset wave of k. Thus, the predicate $I4$ is in the invariant, where

$$
\begin{aligned}
I4 = \quad & (sn.j \neq sn.k \;\wedge\; new.j = 0 \;\wedge \\
& new.l \neq 2) \quad \Rightarrow \quad (\forall m : m \in X.l : sn.m \neq sn.k) \\
& \qquad \vee\ (sn.l \neq sn.k \;\wedge \\
& \qquad\qquad (\exists m : m \in X.l \cap des.k.k : sn.m = sn.k)) \\
& \qquad \vee\ (sn.l = sn.k \;\wedge\; l \in des.k.k)
\end{aligned}
$$

Starting from a state where $I4$ holds, if j propagates a reset wave initiated by k, then the consequent of $I4$ continues to hold. When l executes the completion action and $new.l$ is decremented from 2 to 1, from NGD, all ancestors of l have the same sequence number. Thus, the predicate $I5$ is in the invariant, where

$$
\begin{aligned}
I5 = \quad & (sn.j = sn.k \;\wedge\; st.j = reset \;\wedge\; new.j = 0 \;\wedge\; new.l \neq 2) \\
\Rightarrow \\
& (\forall m : m \in X.l : sn.m \neq sn.k) \\
& \vee\ (sn.l \neq sn.k \;\wedge\; (\exists m : m \in X.l \cap des.k.k : sn.m = sn.k)) \\
& \vee\ (sn.l = sn.k \;\wedge\; l \in des.k.k) \\
& \vee\ (new.l = 1 \;\wedge\; (\forall m : m \in X.l : sn.m = sn.k) \;\wedge\; st.l \neq reset)
\end{aligned}
$$

As claimed earlier, when a process j completes a reset wave, if $new.l$ and $new.j$ are zero, then l has propagated the current reset wave initiated by k. Thus, the predicate $I6$ is in the invariant, where

$$
\begin{aligned}
I6 = \quad & (sn.j = sn.k = sn.l \;\wedge\; st.j = reset \;\wedge \\
& new.j = 0 \;\wedge\; new.l = 0) \quad \Rightarrow \quad (l \in des.k.k)
\end{aligned}
$$

When k starts a new reset wave, if $sn.j$ is different from $sn.k$, j has not propagated this reset wave, i.e., $pc.j.k$ is the empty set. When j propagates a reset wave, all processes in $pc.j.k$ are children of j, unless some of these children have moved to a different tree or failed. If one of the process in $pc.j.k$ moves to a different tree or fails, $new.j$ is set to 2. Thus, the predicate $I7$ is in the invariant, where

$$I7 = \quad (new.j = 0) \quad \Rightarrow \quad (sn.j \neq sn.k \ \Rightarrow\ pc.j.k = \phi) \\ \wedge\ (sn.j = sn.k \ \Rightarrow\ pc.j.k \in ch.j)$$

Finally, if j and $par.j$ are propagating a reset wave initiated by k and $par.j$ is propagating the current wave of k, j is also propagating the current wave of k. Thus, the predicate $I8$ is in the invariant, where

$$I8 = \quad (par.j \in des.k.k \ \wedge\ sn.j = sn.(par.j) \ \wedge\ st.j = reset) \quad \Rightarrow \quad (j \in des.k.k)$$

Finally, the predicates T_{TREE} and NGD are in the invariant. Thus, the invariant of the masking distributed reset program, MR, is

$$S_{MR} = (NGD \ \wedge\ T_{TREE} \ \wedge\ (\forall j, l : k \in X.j \ \wedge\ l \in Adj.j \ \wedge\ root.j = root.l = k : \\ I1 \ \wedge I2 \ \wedge I3 \ \wedge I4 \ \wedge I5 \ \wedge I6 \ \wedge I7 \ \wedge I8)$$

Proof of Lemma 8.2. When j concludes that j and l are propagating the same reset wave, $new.j$ and $new.l$ is zero. From $I6$, it follows that l has propagated the current reset wave propagated by k. □

Lemma 8.3. At any state where S_{MR} holds, if a root process declares that a distributed reset has completed correctly, then $nbrs(des.k.k) \subseteq (des.k.k \cup failed.k)$.

Proof. When k declares that reset is complete by executing action $MR3$, we have

$(\forall l : l \in (Adj.k \cup \{k\}) : new.l = 0 \ \wedge \ root.l = root.k \ \wedge \ sn.l = sn.k) \quad \wedge$

$(\forall l : l \in ch.k : st.j \neq reset) \quad \wedge \quad (st.k = reset)$

$\Rightarrow$ { from I6, I1, I7 }

$(\forall l : l \in (Adj.k \cup \{k\}) : l \in des.k.k) \quad \wedge$

$(\forall l : l \in ch.k : nbrs(des.l.k) \subseteq (des.k.k \cup failed.k)) \quad \wedge \quad (pc.k.k \subseteq ch.k)$

$\Rightarrow$ { by predicate calculus }

$(\forall l : l \in Adj.k : l \in des.k.k) \quad \wedge$

$(\forall l : l \in pc.k.k : nbrs(des.l.k) \subseteq (des.k.k \cup failed.k))$

$\Rightarrow$ { by definition of $des.k.k$ }

$nbrs(des.k.k) \subseteq (des.k.k \cup failed.k)$ $\square$

Theorem 8.4. Program MR is masking tolerant to fail-stop and repair faults from S_{MR}.

8.6 Stabilizing and Masking Fault-Tolerant Distributed Reset

In this section, we transform program MR to add stabilizing tolerance to transient faults. To this end, we add a corrector to program MR that restores it from an arbitrary state to a state where S_{MR} holds.

We proceed as follows: Since each state where S_{MR} holds satisfies T_{TREE} and NGD, we add convergence actions to program MR that restore it from an arbitrary state to a state where $T_{TREE} \ \wedge \ NGD$ holds. Further, we show that starting from a state where T_{TREE} and NGD holds, the program converges to a state where S_{MR} holds.

By the definition of T_{TREE} and NGD, $T_{TREE} \wedge NGD$ may be violated if any of the following three conditions hold.

1. The graph of the parent relation contains cycles

2. There exists a process j such that $T1.j$ is violated

3. There exists a process j such that $NGd.j$ is violated

To handle (1), each cycle in the graph of the parent relation is detected and removed. To detect cycles, j maintains a variable $d.j$ to be the distance between j and $root.j$ in the graph of the parent relation. Thus, if j is a root process $d.j$ is zero; otherwise $d.j$ is $d.(par.j) + 1$. Observe that if the graph of the parent relation is acyclic and there are atmost K *up* processes, then for all processes their distance is less than K. Hence, whenever $par.j \in Adj.j$ and $d.(par.j) < K$, j maintains $d.j$ to be $d.(par.j) + 1$. If j belongs to a cycle then $d.j$ increases repeatedly. Whenever $d.j$ exceeds $K-1$, a cycle is detected. To remove the detected cycle, j sets the parent of j to j.

To handle (2), whenever $T1.j$ is violated, j corrects $T1.j$ by separating from the tree and setting $par.j$ and $root.j$ to j.

To handle (3), whenever $NGd.j$ is violated, j corrects $NGd.j$ by copying the state and sequence number values from its parent.

Variables. As described above, each process maintains a variable $d.j$ to denote the distance of j from $root.j$ in the graph of the parent relation.

Actions. Program MSR consists of nine actions for each process j. Actions $MSR1$–6 construct the spanning tree and implement the reset wave; they are identical to the actions $MR1-6$. Action $MSR7$ corrects $d.j$ whenever $par.j \in Adj.j$ and

$d.(par.j) < K$, by setting $d.j$ to $d.(par.j)+1$. Action $MSR8$ is executed when $d.j$ exceeds K or $T1.j$ is violated. As described earlier, when j executes $MSR8$, j sets $par.j$ and $root.j$ to j and $d.j$ to zero. Since j changes its tree in action $MSR8$, j sets both $new.j$ and the new of the old parent of j to 2. Action $MSR9$ is executed when j violates $NGd.j$. j corrects $NGd.j$ by copying the state and sequence number from its parent.

Formally, the actions are as follows:

$$
\begin{array}{lll}
MSR1 :: & MR1 & \\
 & \quad\vdots & \\
MSR6 :: & MR6 & \\
MSR7 :: & & par.j \in Adj.j \ \wedge \ d.j \neq d.(par.j)+1 \ \wedge \ d.(par.j) < K \\
 & \longrightarrow & d.j := d.(par.j)+1 \\
 & & \\
MSR8 :: & & d.j \geq K \ \vee \ \neg T1.j \ \vee \ (par.j = j \ \wedge \ d.j \neq 0) \\
 & \longrightarrow & par.j,\ col.j,\ root.j,\ d.j,\ new.j,\ new.(par.j) := j,\ red,\ j,\ 0,\ 2,\ 2 \\
 & & \\
MSR9 : & & \neg NGd.j \\
 & \longrightarrow & st.j, sn.j := st.(par.j), sn.(par.j)
\end{array}
$$

Invariant. The invariant of the stabilizing and masking program, MSR, is

$$S_{MSR} = S_{MR}$$

To show that masking fault-tolerance to fail-stop and repair faults is preserved, we prove the following lemma.

Lemma 8.5. Actions $MSR7-9$ preserve S_{MR} and do not execute indefinitely after program MSR reaches a state in S_{MR}.

Theorem 8.6. Program MSR is masking tolerant to fail-stop and repair faults and stabilizing tolerant to transient faults from S_{MSR}.

Proof. Here, we show that MSR is stabilizing tolerant to transient faults. The proof of masking tolerance follows from Lemma 8.5 below. Proof of stabilization is based upon the "convergence stair" method of Gouda and Multari [33]. In particular, we exhibit predicates $S.1, S.2, S.3$, and $S.4$, where

$S.1 = True$

$S.2 = NGD$

$S.3 = S.2 \wedge$ graph of the parent relation forms a tree $\wedge$

$$(\forall j,k :: root.j = root.k \ \wedge\ col.j = green \ \wedge\ par.j \in Adj.j \ \wedge$$
$$(par.j = j \ \Rightarrow\ d.j = 0) \ \wedge\ (par.j \neq j \Rightarrow d.j = d.(par.j) + 1))$$

$S.4 = S.3 \ \wedge\ S_{MSR}$

and show that for each i, $1 \leq i < 4$, $S.i$ converges to $S.(i+1)$ in program MSR. It follows that starting from an arbitrary state, program MSR reaches a state where $S.4$ and, hence, S_{MSR} holds.

To prove $S.1$ converges to $S.2$, consider the set of processes for which $NGd.j$ is violated. When a process executes action $MSR9$, the cardinality of this set decreases by 1. The cardinality of the set never increases. Thus, eventually, the cardinality of the set reduces to zero and, hence $S.2$ holds. Also, it is easy to observe that $S.2$ is closed under the execution of program MSR.

To prove that $S.2$ converges to $S.3$, we use a proof identical to the convergence proof in Arora and Gouda's reset [9]. For brevity, we omit this proof here. We merely note that in a state that satisfies $S.3$, since the graph of the parent relation forms a tree and the predicate $NGd.j$ holds for all processes, actions $MSR4-9$ are disabled. Actions $MSR1-3$ trivially preserve $S.3$. Thus, $S.3$ is closed under execution of program MSR. Also, if the set of processes form a rooted tree and all processes are all green, the predicate T_{TREE} holds trivially.

To prove $S.3$ converges to $S.4$, we need to prove that the program converges to a state where the predicates $I1-8$ hold. We first prove that the program reaches a state where k, the root of the graph of the parent tree, is in the normal state. We then prove that in such a state, the predicates $I1-8$ hold.

As mentioned above, in a state satisfying $S.3$, only actions $MSR1-3$ may be enabled. If action $MSR1$ is executed, then k is in the normal state. Hence, to prove that eventually the program reaches a state where k is in the normal state, we can assume that action $MSR1$ is not executed.

Consider the variant function $|\{j : st.j = reset\}| + 2|\{j : sn.j \neq sn.k\}|$. When a process executes either the propagation or the completion action, the value of this function decreases. Thus, eventually the system will reach a state where the value of this function is zero, in which case k is in the normal state.

We now prove that when k is in the normal state, the predicates $I1-I8$ hold. From NGD, we have that all processes are in the normal state with the sequence number $sn.k$. Hence, the following predicate is true for each process j:

$$(\forall j :: sn.j = sn.k \ \wedge \ st.j \neq reset)$$

Since k is not in the reset state, predicates $I1, I5, I6$ are satisfied. Since all processes have the same sequence number, the predicate $(\forall m : m \in X.l : sn.m = sn.k)$ holds for all processes l, i.e., predicates $I2, I3$ are satisfied. Since, $sn.j$ is equal to $sn.k$ and j is in normal state, predicates $I4, I7, I8$ are also satisfied. Thus, when k is in the normal state, the predicate S_{MSR} holds.

It now follows that starting from an arbitrary state, the program execution converges to a state where the predicate S_{MSR} holds. $\square$

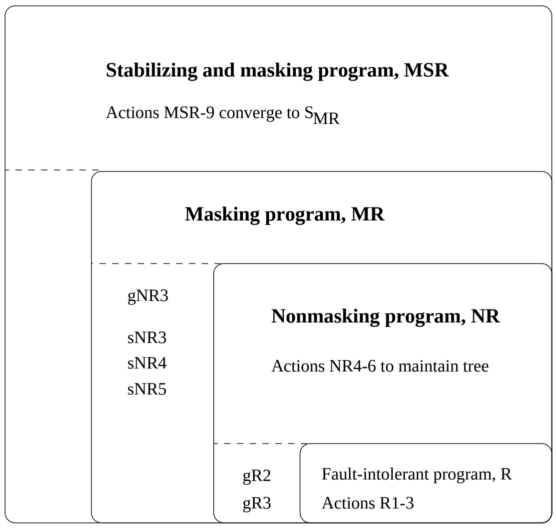
Proof of Lemma 8.5. From the above proof eventually the program reaches a state where $S.3$ is satisfied. In a state where $S.3$ is satisfied, $MSR7-9$ are disabled. Since $S.3$ is closed in MSR, these actions remain disabled. Thus, actions $MSR7-9$ do not execute indefinitely.

Since action $MSR7$ does not update any variable in S_{MR} and $MSR9$ is disabled in S_{MR}, these two actions trivially preserve S_{MR}. Observe that when j executes action $MSR8$, the effect is equivalent to a fail-stop of j followed by an instantaneous repair of j. And, we know that S_{MR} is preserved under fail-stop and repair faults. Thus, $MSR8$ also preserves S_{MR}.

Remark. The fault-span of MSR in the presence of fail-stop and repair faults, $T1$, is identical to the invariant S_{MSR}. Since the MSR is stabilizing tolerant to transient faults, the fault-span in the presence of transient faults, $T2$, is *true*.

8.7 Bounding the Sequence Numbers of Multitolerant Applications

In this section, we discuss how applications use the bounded sequence numbers associated with the multitolerant distributed reset in order to become multitolerant. For ease of exposition, we focus our attention on the case when the application has



Legend :

Action Yi of program Y is obtained by action Xi of program X with gXi added to the guard of Xi and sXi added to the statement of Xi. Unless explicitly mentioned gXi is true and sXi is skip

gR2 = (root.j = root.(par.j)) and col.(par.j) = green
gR3 = (forall k : k in ch.j : root.j = root.k and col.j = green)

gNR3 = (forall l : l in Adj.j : root.j = root.l and sn.j = sn.l)

sNR3 = update new.j, new.(par.j)
sNR4 = new.j, new.(par.j) := 2, 2
sNR5 = new.j, new.(par.j) := 2, 2

Figure 8.3: Composition of the Masking and Stabilizing Reset Program (MSR)

to be made masking fault-tolerant to some fault-class. In this case, when a fault is detected, the application is reset while ensuring that both the old and the new instance of the application execute correctly. In particular, we ensure that two processes communicate only when they have been reset in the same reset wave.

Of course, if the sequence numbers associated with the distributed resets are unbounded, ensuring this property is simple: it suffices to restrict the communication between any two processes to occur only when their sequence numbers are identical. When the sequence numbers are bounded, however, it is possible for two processes to have the same sequence number even though they have reset their states in different reset waves.

We therefore introduce an *incarnation number* for each process. Two processes have the same incarnation number iff they have reset in the same reset wave. Hence, it suffices to restrict the communication between two processes to occur only when their incarnation numbers are identical.

It turns out that a bound of two on the incarnation number is insufficient. To see this, consider the following computation: The initial state is as shown in Figure 8.4 (a): Processes $1, 2$, and 3 form a tree rooted at 3. Let 3 initiate a reset wave to change the incarnation numbers to 1. Processes 1 and 3 propagate this reset wave, and change their incarnation number to 1 (Figure 8.4 (b)). Before 2 propagates this reset wave, 3 fails, and the resulting tree is rooted at 2 (Figure 8.4 (c)). Unfortunately, the state in Figure 8.4 (c) can alternatively be reached by starting from a state in Figure 8.4 (d) and letting process 2 initiate a new reset wave. In the first computation, all processes should change their incarnation numbers to 1 in the future. In the second computation, all processes should change their incarnation numbers to 0 in the future. Thus, keeping only two incarnation numbers is insufficient, as processes cannot determine which incarnation number is current.

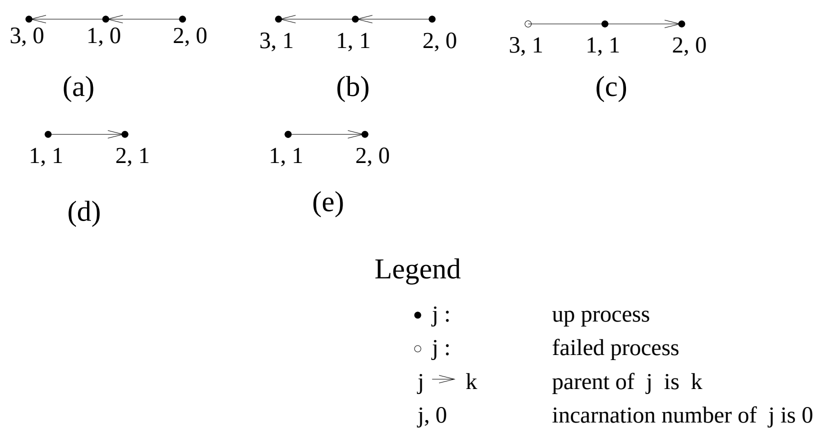


Figure 8.4: Insufficiency of two incarnation numbers

Fortunately, determining the current incarnation number is possible if we use ternary ($0, 1$ or 2) incarnation numbers and follow the rule that a new incarnation number $B + 1$ can be created only if the incarnation number $B - 1$ does not exist in the system (in this section, we let $+$ and $-$ denote modulo 3 addition and subtraction respectively.) It follows that at any invariant state at most two incarnation numbers may coexist. Also, for any pair of incarnation numbers, it is easy to determine which of the two is associated with the current reset wave.

When to reset the local state of a process. When process j propagates a reset wave, it resets its state iff the incarnation number of the parent of j is one greater (in mod 3 arithmetic) than the incarnation number of j. When j resets its state, it also copies the incarnation number of its parent.

When to create a new incarnation number. As motivated above, all three incarnation numbers should not exist simultaneously. Hence, we ensure that a process changes its incarnation number from B to $B+1$ only if no process with incarnation number $B-1$ exists in the system. From the invariant of the reset program, we know that when the root completes a reset wave successfully, all processes have propagated that reset wave. Thus, if a process with incarnation number $B-1$ existed in the system when the root initiated the reset wave, this process would have set its incarnation number to B upon propagation of the reset wave. Also, no process can change its incarnation number from B to $B-1$. Hence, when the root process completes a reset wave successfully, it can safely increment its incarnation number.

When to declare that reset is complete. When a root completes a reset wave successfully with incarnation number B, no process in the system has incarnation number $B-1$. It is, however, possible that some processes have incarnation number $B+1$. (Such a state may arise if the tree program converges to a state where the root does not have the current incarnation number). Hence, we let the root declare completion of a reset only if the reset completes correctly *and* the root can detect that the incarnation numbers of all processes are the same. To this end, we let each process check whether all its children have the same incarnation numbers when it completes in the reset wave. The completion wave propagates this information towards the root so that the root can determine whether the incarnation numbers of all processes are the same.

Variables. Every process j maintains the following variables.

- $inc.j$: incarnation number; one of 0, 1, or 2
- $ares.j$: application result; a boolean

Actions. Program APP consists of nine actions for each process j. The first three actions implement the reset wave. These actions are obtained by modifying actions MSR1–3 to update inc and $ares$ appropriately. Action $APP1$ is the initiation action; whenever j initiates a reset wave, j increments its incarnation number iff $new.j$ is zero. Action $APP2$ is the propagation action; whenever j propagates a reset wave, j increments its incarnation number iff the incarnation number of the parent of j is one higher than that of j. Action $APP3$ is the completion action; whenever j completes a reset wave, j sets $ares.j$ to true iff all children of j have the same incarnation number and their application result is true. Finally, the remaining actions $APP4-9$ break cycles in the graph of parent relation and maintain the underlying tree; these actions are identical to the actions $MSR4-9$ of program MSR.

$APP1 ::$ $par.j = j \ \wedge \ st.j = normal \ \wedge$

$\{j$ needs to initiate a new reset wave$\}$

$\longrightarrow$ $st.j, \ sn.j := reset, \ sn.j \oplus 1;$

if $(new.j = 0)$ then

$inc.j := inc.j + 1;$ { reset application state }

$APP2 ::$ $root.j = root.(par.j) \ \wedge \ col.(par.j) = green \ \wedge$

$st(par.j) = reset \ \wedge \ sn.j \neq sn.(par.j)$

$\longrightarrow$ $st.j, \ sn.j := reset, \ sn.(par.j);$

if $((inc.j + 1) = inc.(par.j)))$ then

$inc.j := inc.(par.j);$ { reset application state }

$APP3 ::$ $st.j = reset \ \wedge \ col.j = green \ \wedge \ (\forall l : l \in ch.j : st.l \neq reset) \ \wedge$

$(\forall l : l \in Adj.j : root.l = root.j \ \wedge \ sn.j = sn.l)$

$\longrightarrow$ $st.j, ares.j := normal, (\forall l : l \in ch.j : ares.l \ \wedge \ (inc.j = inc.l));$

if $(\exists l : l \in Adj.j \cup \{j\} : new.l > 0)$ then

if $(par.j = j)$ then $st.j, \ sn.j := reset, sn.j \oplus 1;$

else $new.(par.j) := max(new.(par.j), 1)$

else if $(par.j = j \ \wedge \ ares.j)$ then { declare reset complete }

else if $(par.j = j \ \wedge \ \neg ares.j)$ then

$st.j, sn.j, inc.j := reset, sn.j \oplus 1, inc.j + 1;$

{ reset application state };

$new.j := max(0, new.j - 1)$

$APP4 ::\ MSR4$

$$\vdots$$

$APP9 ::\ MSR9$

Outline of the proof. We need to show that: (1) When a root process with incarnation number B declares that a reset is complete, all processes have incarnation number B. (2) At most two incarnation numbers co-exist at any invariant state. (3) After a reset operation is initiated at an invariant state the root process eventually declares that the reset is complete.

When j completes a reset wave, it checks that the incarnation number of j is the same as its children. Thus, when j completes its reset wave, it can detect whether all processes that received the reset wave via j have the same incarnation number. Recall that when the root declares that reset is complete, we know from MSR that all processes have propagated that reset wave. Hence, when the root declares that a reset is complete, all processes have the same incarnation number. Thus, (1) is satisfied.

To show (2), we show that when a process creates a new incarnation number $B+1$, after completing a reset wave with incarnation number B, no process has incarnation number $B-1$. Suppose not: consider the first process, say j, that has incarnation number $B-1$ after propagating this reset wave. We consider three cases depending upon the value of the incarnation number of $par.j$ when j propagated this reset wave. (i) The incarnation number of $par.j$ cannot be $B-1$ since $par.j$ has propagated the reset wave and j is the first process to change its incarnation number to $B-1$. (ii) If incarnation number of $par.j$ is B, then after propagating the reset wave, incarnation number of j is either B or $B+1$. (iii) If incarnation number of $par.j$ is $B+1$, then before j propagates the reset wave, incarnation number of j cannot be $B-1$, as only two incarnation numbers coexist. Thus, (2) is satisfied.

From MSR, we know that eventually the root with incarnation number B will either declare that reset is complete or initiate a new reset wave with incarnation number $B+1$. Upon propagation of this reset wave, all processes with incarnation number B change their incarnation number to $B+1$. It follows that the root will eventually declare that the reset is complete. Thus, (3) is satisfied.

8.8 Refinement to Low Atomicity

In program MR, the actions $MR3, MR4$ and $MR6$ of process j update the $new.(par.j)$ variable of $par.j$ simultaneously with the variables of j. We now refine program MR so that in each action a process updates only its own variables. The refinement is made possible by the fact that the parent of j cannot complete its reset wave until j completes its reset wave. Hence, $new.(par.j)$ can be updated after the variables of j have been updated.

In the refined version of MR, for each of its neighbors k, j maintains a variable $new.j.k$ as a local copy of the nonlocal variable $new.k$. Whenever j needs to update the new value of k, j updates the value of $new.j.k$. Process k asynchronously reads $new.j.k$ and updates $new.k$. More specifically, MR is modified as follows (the modification to MSR is analogous):

1. Add a variable $new.j.k$ in j for every neighbor k.

2. Replace the statements $new.k := m$ by $new.j.k := m$.

3. Add the following two actions, the first to k and the second to j.

$$new.j.k > 0 \ \wedge \ new.j.k > new.k \quad \longrightarrow new.k := new.j.k$$

$$new.j.k > 0 \ \wedge \ new.j.k \leq new.k \quad \longrightarrow new.j.k := 0$$

4. Replace action $MR3$ of j as follows:

$$(\forall j : j \in Adj.j : new.j.k = 0) \ \wedge \ \ guard \quad \longrightarrow \quad statement,$$

where *guard* and *statement* denote the guard and statement of action MR3 respectively.

With this refinement, the predicates $I1$ and $I7$ of the invariant S_{MR} change as follows:

$$
\begin{aligned}
I1' &= \quad I1 \ \vee \ new.j.(par.j) > 0 \\
I7' &= \quad (new.j = 0) \quad \Rightarrow \quad (sn.j \neq sn.k \ \Rightarrow \ pc.j.k = \phi) \\
&\qquad\qquad\qquad\qquad \wedge \ (sn.j = sn.k \ \Rightarrow \quad pc.j.k \in ch.j \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee (\exists l : l \in Adj.j : new.l.j > 0))
\end{aligned}
$$

And, the invariant S_{MR} changes as follows:

$$
\begin{aligned}
S'_{MR} = (NGD \ \wedge \ T_{TREE} \ \wedge \ (\forall j, l : j \in KERN.k \ \wedge \ l \in Adj.j \ \wedge \ root.j = root.l = k : \\
I1' \ \wedge I2 \ \wedge I3 \ \wedge I4 \ \wedge I5 \ \wedge I6 \ \wedge I7' \ \wedge I8))
\end{aligned}
$$

Finally, in the fail-stop action of program MR, the state of all its the neighbors is updated when process j fail-stops. This action can be easily refined so that each neighbor asynchronously updates its own state when it detects that j has fail-stopped. Thus, in the refined program, each process updates only its own state.

8.9 Chapter Summary

In this chapter, we presented the first bounded memory reset protocol that masks failure and repair of processes and stabilizes in the presence of transient faults. Thus, it provides the ideal type of tolerance to each of these two fault-classes. Following our method in Chapter 6, we first designed a fault-intolerant program. Then, we added detectors and correctors to achieve masking fault-tolerance to failure and repair of processes; as suggested in Chapter 5, this itself was done in two stages. Finally, we added a corrector to achieve stabilizing tolerance to transient faults. In designing this reset program, we developed a multitolerant detector to check whether all processes have participated in a diffusing computation. This detector is applicable in designing bounded state multitolerant programs for several problems. We have illustrated the use of this detector in [42]

Distributed reset programs belong to the class of total programs [63]. Total programs characteristically contain one or more "decider" actions, whose execution depends on the state of all processes. In the case of distributed reset programs, the action which declares the completion of a reset operation is a decider action. The safety of the execution of this decider action was achieved in this chapter by using a detector that ensures that (1) if fail-stops and repairs occur, the decider action is executed only after all processes are contacted, and (2) after transient faults occur, eventually the program reaches a state from where the decider action executes only after all processes are contacted. This strategy provides a basis for making other total programs likewise multitolerant: Before a process executes a decider action, it waits for the detector to collect the state of all processes (i.e. whenever a process is contacted, its state is collected). From (1) and (2), it follows that the resulting total program is multitolerant.

CHAPTER 9

APPLICATION IN MECHANICAL VERIFICATION : A CASE STUDY

9.1 Introduction

In this chapter, we argue that the decomposition of a fault-tolerant program into its components is beneficial in its mechanical verification, and that such a decomposition admits reuse of the proofs for other fault-tolerant programs as well as the variations of the given fault-tolerant program.

As described in Chapters 3 and 4, a fault-tolerant program can be decomposed into a fault-intolerant program and a set of detectors and/or correctors. In particular, a fail-safe program, which satisfies only its safety specification in the presence of faults, can be decomposed into a fault-intolerant program and detector(s). Likewise, a self-stabilizing program, which guarantees recovery to a state from where its specification is satisfied, can be decomposed into a fault-intolerant program and corrector(s).

Decomposition of a fault-tolerant program permits the verification of a given property by focusing on the component that is responsible for satisfying it. For example, if we need to show that a program eventually recovers to a state from where it satisfies its specification, we should focus on its corrector components. Likewise, if we are interested in showing that the program satisfies its specification in the absence of faults, we should focus on the corresponding fault-intolerant program. Of course, we will have to show that other components of the program do not interfere with the component of interest. But this proof is typically simpler than the proof required to show that the overall program satisfies the given property. Moreover, if we change some components used in that program, the proofs of other components are not affected. Thus, it is possible for a small change in the program to lead to a small change in the proof.

With the motivation of developing a systematic approach for mechanical verification using program decomposition, we are implementing the theory of detectors and correctors into the theorem prover PVS [54][1]. In this chapter, we present a proof of one of Dijkstra's token ring program [25] that has been proved using this theory. Previously, Qadeer and Shankar [56] have verified this token ring program using PVS. While their proof is impressive, it is very specific to one program and, hence, much of their proof-technique cannot be reused to prove other fault-tolerant programs. Moreover, since they focus on the entire program, instead of its components, their proof is more complex than it needs to be. We use this case-study to illustrate how the decomposition of the program into its components can help in making the proofs simple and reusable.

Being self-stabilizing, Dijkstra's program can be decomposed into a fault-intolerant program and a corrector. The fault-intolerant program circulates the token along an initialized ring in the absence of faults. On the other hand, if faults perturb the program from its ideal states, the corrector restores the fault-intolerant program back

[1]The URL `http://www.cis.ohio-state.edu/~kulkarni/pvs/` contains the PVS specification and proofs.

to some ideal state, from where it continues to circulate the token. This program is self-stabilizing in that even if the faults perturb the program to an arbitrary state, the corrector restores it to an ideal state.

In Dijkstra's token ring program, processes $0..N$, $N \geq 1$, are organized in a ring. Each process j maintains a counter $x.j$, $0 \leq x.j < M$ for some $M > 1$. A non-zero process j has a token iff $x.j$ differs from $x.(j-1)$, and process 0 has a token iff $x.0$ is the same as $x.N$. If process j has a token then it passes it to process $j{+}1$ *mod* $N{+}1$ by setting $x.j$ to $x.(j-1)$, and if process 0 has a token then it passes it to process 1 by incrementing $x.0$. For any M, $M > 1$, the program guarantees that in the absence of faults there will be exactly one token that is being circulated in the ring. If $M \geq N+1$, the program guarantees that starting from any arbitrary state, the program will reach a state where there is exactly one token which is circulated along the ring.

To decompose Dijkstra's program into a fault-intolerant program and a corrector, we first consider the following question: If we are only interested in a token circulation along an initialized ring, how can the token ring program be simplified? The answer to this question identifies the fault-intolerant program. Next we ask the question about fault-tolerance: what are the ideal states of the resulting fault-intolerant program, and how can it be recovered to these ideal states if the faults perturb it? The answer to this question identifies the corrector. Then, we show how the fault-intolerant program and the corrector can be independently verified in PVS and how they can be shown to be interference-free.

The rest of the chapter is organized as follows: In Section 9.2, we present Dijkstra's token ring program and its decomposition into a fault-intolerant program and a corrector. In Section 9.3, we show how the token ring program is modeled in PVS. In Section 9.4 and 9.5, we present the correctness proof for the fault-intolerant program and the corrector respectively. In Section 9.6, we show that the corrector and the fault-intolerant program do not interfere with each other. Finally, in Section 9.7, we discuss the advantages of component-based verification over non-component-based verification.

9.2 The Token Ring Program and its Decomposition

In this section, we first present the decomposition of Dijkstra's token ring program into a fault-intolerant program and a corrector. Then, we argue that they work in isolation and that they do not interfere with each other. We use the same arguments for mechanical verification in Sections 9.4, 9.5 and 9.6.

Fault-intolerant program. If we are not interested in fault-tolerance, a token ring program can be designed by maintaining a variable $x.j$ (in the range 0..(M-1), where $M > 1$) as follows: Each non-zero process j checks whether $x.j$ is different from $x.(j-1)$. If this condition is true then $x.j$ is set to $x.(j-1)$. Process 0 checks whether $x.0$ is the same as $x.N$. If this condition is true, process 0 increments $x.0$. Thus, the actions of the fault-intolerant program are as follows:

$$
\begin{array}{lcl}
j \neq 0 \ \wedge \ x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1) \\
x.0 = x.N & \longrightarrow & x.0 := x.0 + 1
\end{array}
$$

The invariant of this program is S, where

$$S \;=\; (\exists j,v : 0 \le j \le N, 0 \le v < M \;: \\ (\forall k : 0 \le k < j : x.k = v) \;\wedge\; (\forall k : j \le k \le N : x.k = v-1 \; mod \; M))$$

The invariant S characterizes the states where there exists a process j such that the x values of processes $0..(j-1)$ are equal to v, and the x values of processes $j..N$ are equal to $v-1 \; mod \; M$. Thus, process j has the unique token, and only the action at j is enabled in that state. The execution of this action results in a state where process $j+1 \; mod \; N+1$ has the token. Thus, starting from a state where S is true, the fault-intolerant program circulates a unique token along the ring.

Corrector. If the faults perturb the x values maintained at the processes, we need to recover the program to a state where S holds in order to ensure that the token circulation is re-established. This can be achieved by the corrector that lets each non-zero process copy the x value of its predecessor. Thus, the actions of the corrector are as follows:

$$j \neq 0 \;\wedge\; x.j \neq x.(j-1) \quad \longrightarrow \quad x.j := x.(j-1)$$

Observe that if the corrector actions execute in isolation, a state is reached where all x values are same, and at that state S is true. Also, if the corrector executes in any state where S is true, S continues to be true in the resulting state.

Note that although the actions of the fault-tolerant program and that of the fault-intolerant program are the same, when dealing with the fault-intolerant program we can assume that the invariant S is true. In this sense, the fault-intolerant program is simpler than the fault-tolerant program. Of course, the actions of the corrector are a subset of the fault-tolerant program and, hence, the corrector is simpler than the fault-tolerant program.

Interference-freedom between the fault-intolerant program and the corrector. Since the corrector is a subset of the fault-intolerant program, it is trivial that the corrector does not interfere with the fault-intolerant program. Likewise, the actions of the fault-intolerant program at non-zero processes are a subset of the corrector and, hence, do not interfere with the corrector. Thus, we only need to show that the action at process 0 does not interfere with the corrector. We prove the interference-freedom as follows:

1. For process 0 to interfere with the corrector, it must execute infinitely often. Otherwise, after 0 stops executing, convergence to S will be achieved.

2. If the action at process 0 executes infinitely often, $x.0$ will take all possible values in the range $0..(M-1)$.

3. If the domain of x is large enough, specifically $M \geq N+1$, then in the initial state, there must be a value in the range $0..(M-1)$ which is not present at any non-zero process.

4. From 2 and 3, it follows that eventually $x.0$ will obtain a value missing in the initial state.

5. After $x.0$ is equal to this missing value, process j will obtain this missing value only after processes $0..(j-1)$ obtain this missing value. Thus, when process 0 executes next (from 1, we know that process 0 will execute next), all processes will have the same x value. Thus, a state where S is true is reached.

9.3 Modeling of the Token Ring in PVS

In this section, we discuss how we modeled Dijkstra's token ring program in PVS. More specifically, we first define program independent concepts such as states, state predicates, actions, program compositions, etc. Then, we define the actions of the token ring program and its invariant.

Definition *(State).* The state of the program consists of the x values at processes $0..N$, each x value is in the range $0..(M-1)$.

Definition (Trace). A trace is an infinite sequence of states. If seq is a trace and i is a natural number then $seq(i)$ denotes the i^{th} element in seq.

Definition *(Assertion).* An assertion is a predicate over states. If P is an assertion and s is a state then $P(s)$ denotes whether P is true in state s.

Definition *(Action).* An action is a relation over states. If A is an action and $s1, s2$ are states then $A(s1, s2)$ denotes whether state $s2$ can be reached by executing A in state $s1$.

Definition *(Property).* A property is a predicate over traces. If R is a property and seq is a trace then $R(seq)$ denotes whether the property R is true of seq.

Notation. Henceforth, we use p and q to denote programs; $s, s0, s1$ and $s2$ to denote program states; seq to denote a trace; S and T to denote assertions; R to denote a property; m, n to denote natural numbers; j, k to denote processes; and $v, v1, v2$ to denote the x values at processes. Moreover, given a state s, $x(s)(j)$ denotes the value of $x.j$ in state s. These notations are described in Table 9.1.

Definition *(Program compositions).* In the base case, a program is just a single action. The parallel composition of programs p and q, denoted as $p \| q$, is a program consisting of the actions of p and the actions of q. (While we have defined

Variable	Used as
p, q	program
$s, s0, s1, s2$	state
seq	trace
S, T	assertion
R	property
m, n	natural number
j, k	process, domain $0..N$
$v, v1, v2$	x value for a process, domain $0..(M-1)$

Expression	Meaning
$x(s)(j)$	The value of $x.j$ in state s
$seq(n)$	n^{th} state in the sequence seq

Table 9.1: PVS Notations used in this chapter

other program compositions used for fault-tolerant programs, we omit them here as they are not used in Dijkstra's token ring program.)

Definition *(CanExecute).* Program p can execute in state $s1$ iff there exists a state $s2$ such that $p(s1, s2)$ is true.

$$CanExecute(p)(s1) \;=\; (\exists s2 :: p(s1, s2))$$

Definition *(Next).* The predicate $Next(p)(s1, s2)$ denotes whether state $s2$ can be reached by execution of some action of p. If no action of p is enabled in state $s1$ then $Next(p)(s1, s2)$ is true iff $s1 = s2$.

$$\begin{aligned} Next(p)(s1, s2) \;=\; & \;(CanExecute(p)(s1) \;\wedge\; p(s1, s2)) \\ & \vee\; (\neg CanExecute(p)(s1) \;\wedge\; s1 = s2) \end{aligned}$$

Definition *(Computation).* A computation of a program p is a trace $s_0, s_1, ...$ such that for each n, $Next(p)(s_n, s_{n+1})$ is true. Thus, the predicate characterizing 'seq is a computation of program p' is represented as follows:

$$run(p)(seq) \;=\; \forall n :: Next(p)(seq(n), seq(n{+}1))$$

Definition *(Satisfies).* Program p satisfies a property R iff for every trace that is a computation of p, $R(seq)$ is true. Thus, $satisfies(p)(R)$ is defined as follows:

$$satisfies(p)(R) \;=\; \forall seq :: run(p)(seq) \Rightarrow R(seq)$$

We use two types of properties in the proof of Dijkstra's token ring program, closure and convergence.

Definition *(Closure).* The property $closed(S)$ is the set of all traces $s_0, s_1, ...$ where for each $n, n \geq 0$, if S is true at s_n then S is true s_{n+1}. Thus, $closed(S)$ is defined as follows:

$$closed(S)(seq) \;=\; \forall n :: S(seq(n)) \Rightarrow S(seq(n{+}1)))$$

Definition *(Convergence).* The property $converges(T, S)$ is the set of all traces $s_0, s_1, ...$ where $closed(S)$ and $closed(T)$ are true, and if there exists $n, n \geq 0$, for which T is true at s_n then there exists m, $m \geq n$, for which S is true at s_m. Thus, $converges(T, S)$ is defined as follows:

$$\begin{aligned} converges(T,S)(seq) \;=\;& closed(T)(seq) \;\wedge\; closed(S)(seq) \;\wedge \\ & \forall n :: T(seq(n)) \;\Rightarrow\; (\exists m : m{\geq}n : S(seq(m))) \end{aligned}$$

Definition *(Num_steps).* Given an action ac, a trace seq, and a natural number n, the number of times action ac is executed until the n^{th} state is defined as follows:

$$num\_steps(ac)(seq, n) \;=\; \begin{cases} 0 & \text{if } n{=}0 \\ num\_steps(ac)(seq, n{-}1) + 1 & \text{if } ac(seq(n{-}1), seq(n)) \\ num\_steps(ac)(seq, n{-}1) & \text{otherwise} \end{cases}$$

Definition *(Corrector).* The corrector action at a non-zero process j is executed only in states where $x.j$ differs from $x.(j-1)$. The execution of this action results in a state where $x.j$ has the same value as that of $x.(j-1)$ and the other x values remain unchanged. Thus, corrector action at j is defined as follows:

$$\begin{aligned} corr(j)(s0,s1) \quad = \quad & x(s0)(j) \neq x(s0)(j-1) \quad \wedge \quad x(s1)(j) = x(s0)(j-1) \quad \wedge \\ & \forall k : k \neq j : x(s1)(k) = x(s0)(k) \end{aligned}$$

The corrector consists of the actions at all non-zero processes. We, therefore, use parallel composition of $corr(j)$, $0 < j \leq N$, to define the corrector, $corr\_prog$, as follows :

$$corr\_prog \quad = \quad ([\!] j : j \neq 0 : corr(j))$$

Action at process 0. The action at process 0 is executed only in states where $x.0$ is the same as $x.N$. The execution of this action results in a state where the value of $x.0$ is one greater than its initial value (in mod M arithmetic) and the other x values remain unchanged. Thus, the action at process 0 is defined as follows:

$$\begin{aligned} action\_zero(s0,s1) \quad = \quad & x(s0)(0) = x(s0)(N) \;\; \wedge \\ & x(s1)(0) = x(s0)(0) + 1 \; mod \; M \;\; \wedge \\ & \forall j : j \neq 0 : x(s1)(j) = x(s0)(j) \end{aligned}$$

Note that the fault-intolerant program consists of the parallel composition of the action at process 0 and the corrector. Thus, the fault-intolerant program is $action\_zero [\!] corr\_prog$.

j has a token. We define the predicate, 'j has a token in state s' as follows:

$$\begin{aligned} token(s)(j) \;=\; & \quad (j=0 \;\wedge\; x(s)(0)=x(s)(N)) \\ & \vee \; (j\neq 0 \;\wedge\; x(s)(j)\neq x(s)(j-1)) \end{aligned}$$

Invariant of the fault-intolerant program. Finally, we define the invariant of the fault-intolerant program, $corr\_pred$, as follows:

$$\begin{aligned} corr\_pred(s) \;=\; (\exists j, v :: \quad & \forall k : k<j : x(s)(k) = v \;\wedge \\ & \forall k : k\geq j : x(s)(k) = v-1 \; mod \; M) \end{aligned}$$

Remark. Although in this presentation, we have given a specific instantiation for the program state, it is initially defined as an uninterpreted type, and then instantiated suitably for the token ring program. This allows program independent concepts such as traces, assertions to be reused for different programs.

9.4 Verification of the Fault-Intolerant Program

To prove the correctness of the fault-intolerant program, we need to show (1) $corr\_pred$ is closed in the fault-intolerant program, and (2) if the token is at process j and an action of the fault-intolerant program is executed then the token is at process $j+1 \; mod \; N+1$ in the resulting state.

In Theorem 9.3, we show that $corr\_pred$ is closed in the fault-intolerant program. In this proof, we use Lemmas 9.1 and 9.2 which show that $corr\_pred$ is closed in the action of process 0 and the actions of non-zero process respectively. Finally, we show the token circulation property in Theorem 9.5.

Lemma 9.1 In the computation of $action\_zero$ alone, $corr\_pred$ is closed. Formally,

$$satisfies(action\_zero)(closed(corr\_pred))$$

Proof. After eliminating the quantifiers and expanding the definitions, we need to show that if *corr_pred* is true in the n^{th} state of the computation then it is true in $(n{+}1)^{th}$ state of that computation. To this end, we first do a case split on the process that has the token in the n^{th} state: In the case, where process 0 has the token, i.e., the x values of processes $0..N$ are $v-1$ mod M for some v, we show that execution of *action_zero* results in a state where the x values of $1..N$ remain $v-1$ mod M and the x value of 0 is v, i.e., *corr_pred* is true. In the case, where process j, $j \neq 0$, has the token, we show that $x.0$ is v and $x.N$ is $v-1$ mod M for some v and, hence, *action_zero* is disabled, i.e., the $(n{+}1)^{th}$ state is identical to the n^{th} state and, hence, *corr_pred* is true in $(n+1)^{th}$ state. □

Lemma 9.2 In the computation of the action at a non-zero process, *corr_pred* is closed. Formally,

$$\forall j :: satisfies(corr(j))(closed(corr\_pred))$$

Proof. The proof of this lemma is similar to that of Lemma 9.1. We show that if process j has the token then in the resulting state process $j+1$ mod $N+1$ has the token, and if any other process has the token, the execution results in a stuttering. In either case, *corr_pred* is true. □

Theorem 9.3 In the computation of the fault-intolerant program, *corr_pred* is closed. Formally,

$$satisfies(action\_zero \| corr\_prog)(closed(corr\_pred))$$

Proof. This lemma is proved by using Lemmas 9.1 and 9.2 and the following property about parallel composition: if an assertion S is closed in programs p and q then it is closed in $p \| q$. □

Lemma 9.4. At least one action of the fault-intolerant program is enabled in any program state. Formally,

$$\forall s :: CanExecute(action\_zero \| corr\_prog)(s)$$

Proof. We prove this lemma by first doing a case-split on whether all x values are equal. If all x values are equal, it follows that $x.0 = x.N$ and, hence, *action_zero* is enabled. If all x values are not equal, we induct on the processes to find the first process, say j, such that $x.j$ differs from $x.0$. Since $x.(j-1) = x.0$ and $x.j \neq x.0$, it follows that process j is enabled. □

Theorem 9.5 Starting from a state where *corr_pred* is true, if the token is at process j then the execution of an action of the fault-intolerant program results in a state where the token is at process $j+1$ mod $N+1$. Formally,

$$\begin{array}{ll} \forall j :: & token(s1)(j) \;\wedge\; corr\_pred(s1) \;\wedge\; Next(action\_zero \| corr\_prog)(s1, s2) \\ \quad\Rightarrow & \\ & (j \neq N \Rightarrow token(s2)(j+1)) \;\wedge\; (j = N \Rightarrow token(s2)(0)) \end{array}$$

Proof. Lemma 9.4 shows that execution of the fault-tolerant program does not result in stuttering. We then show that if process j has the token no other process is enabled. Finally, we show, as in Lemmas 9.1 and 9.2, that the execution of the action at j results in a state where $j+1$ mod $N+1$ has the token. □

9.5 Verification of the Corrector

To prove that *corr_prog* satisfies its specification, we need to show (1) *corr_pred* is closed in *corr_prog*, and (2) starting from any state, in the execution of *corr_prog* alone a state is reached where *corr_pred* is true. Note that (1) follows from Lemma

9.2. In this section, we prove that the corrector satisfies property (2) based on the following observation:

Observation. If only the actions at processes $j..N$ execute in a computation then eventually the x values of processes $j-1..N$ will be identical and the actions of processes $j..N$ will be disabled. $\square$

If $j=1$, the actions at processes $j..N$ are the same as the actions of the corrector and, hence, in the execution of the corrector, eventually the x values of all processes will be identical. Thus, convergence to *corr_pred* is achieved.

In order to obtain the proof of the above observation in PVS, we first define the program consisting of actions of processes $j..N$ and an assertion characterizing the states where the x values of processes $j..N$ are equal. Then, we provide a proof of the above observation in Lemma 9.8. Finally, we prove the convergence property in Theorem 9.9.

We define $corr\_above(j)$, and $same\_as\_N(j)$ as follows:

$corr\_above(j)$. For any j, $j \neq 0$, $corr\_above(j)$ is the program consisting of the actions at processes $j..N$. Formally,

$$corr\_above(j) = ([\!]\, k : j \leq k \leq N : corr(k)) \qquad \square$$

$same\_as\_N(j)$. For any j, $same\_as\_N(j)$ is an assertion which is true in state s iff the x values of processes $j..N$ are identical in state s. Formally,

$$same\_as\_N(j)(s) = \forall k : j \leq k \leq N : x(s)(k) = x(s)(N) \qquad \square$$

Lemma 9.7 If $x.j$ is the same as $x.(j-1)$ in some state in the computation of $corr\_above(j)$, then this condition continues to be true in the rest of the computation. Formally,

$$\begin{array}{lll} \forall seq, j, n : j{\neq}0 : & & run(corr\_above(j))(seq) \;\wedge \\ & & x(seq(n))(j-1) = x(seq(n))(j)) \\ & \Rightarrow & \\ & & \forall m : m{\geq}n : x(seq(m))(j-1) = x(seq(m))(j) \end{array}$$

Lemma 9.8 In the computation of the corrector actions at processes $j..N$, a state is reached where the x values of processes $(j-1)..N$ are identical. Formally,

$$\forall seq, j :: \quad run(corr\_above(j))(seq) \quad \Rightarrow \quad \exists n :: same\_as\_N(j-1)(seq(n))$$

Proof. We prove this lemma by measure-induction on the j, where the measure used is $N-j$. In the base case, $j = N$, we do a case split on whether $corr(N)$ is enabled in the initial state: If $corr(N)$ is enabled, we show that in the successor state, $same\_as\_N(N-1)$ is true. If $corr(N)$ is disabled, we show that in the initial state $same\_as\_N(N-1)$ is true.

In the induction case, we do a case-split on whether the action at process j executes in the computation of $corr\_above(j)$. If j executes in the n^{th} state, we show that the suffix of the computation from the $(n{+}1)^{th}$ state is a computation of $corr\_above(j{+}1)$. Therefore, there exists a state, say s, where $same\_as\_N(j)$ is true. Moreover, since the values of $x.j$ and $x.(j{-}1)$ are equal in the $(n+1)^{th}$ state, by Lemma 9.7, it follows that the values of $x.j$ and $x.(j-1)$ are equal in state s. Thus, $same\_as\_N(j-1)$ is true in state s.

If j never executes in the computation of $corr\_above(j)$, we show that that computation is also a computation of $corr\_above(j+1)$. Therefore, there exists a state, say s, in this computation where $same\_as\_N(j)$ is true. Thus, the actions of $j+1..N$ are disabled in state s. Since the program tries to execute an action unless all its actions are disabled, it follows that the action at j must also be disabled. It follows that $same\_as\_N(j-1)$ is true in s. $\square$

Theorem 9.9 The computation of the corrector eventually converges to $corr\_pred$. Formally,

$$satisfies(corr\_prog)(converges(true, corr\_pred))$$

Proof. We prove this lemma by instantiating $j=1$ in Lemma 9.8. From Lemma 9.8, it follows that in the computation of the corrector alone, eventually a state is reached where all x values are identical and, hence, $corr\_pred$ is true in that state. Moreover, the closure of $corr\_pred$ follows from Lemma 9.2. $\square$

9.6 Interference-Freedom Between the Corrector and the Fault-Intolerant Program

After we showed that the fault-intolerant program and the corrector satisfy their specification in isolation, we proceed to show that they do not interfere with each other. As mentioned in Section 9.2, towards this end, we show that the action at process 0 does not interfere with the corrector. Our proof follows the outline discussed in Section 9.2. More specifically, in Lemma 9.10, we show that process 0 executes infinitely often. Then, in Lemma 9.13, we show that there exists a value that is different from the x values of all non-zero processes. Subsequently, in Lemma 9.15, we show that eventually process 0 gets this missing value, and in Theorem 9.17, we conclude that the action at process 0 does not interfere with the corrector.

Lemma 9.10 In the computation of the corrector and the action at process 0, either process 0 executes infinitely often or a state is reached where *corr_pred* is true. Formally,

$$\begin{array}{ll} \forall seq, n :: & run(action\_zero [] corr\_prog)(seq) \\ \quad \Rightarrow & \\ & \quad \forall m :: (\exists n : n \geq m : action\_zero(seq(n), seq(n+1))) \\ & \vee \; \exists m :: corr\_pred(seq(m)) \end{array}$$

Proof. Note that process 0 executes infinitely often iff given any number m, it executes in the n^{th} state for some $n \geq m$. Thus, to prove this lemma we need to show that either (1) there exists a number n, $n \geq m$, such that *action_zero* executes in the n^{th} state, or (2) there exists a state where *corr_pred* is true. We prove this lemma by a case-split on whether the suffix of the computation starting from m^{th} state is a computation of *corr_prog*. If that suffix is a computation of *corr_prog*, by Theorem 9.9, it is straightforward to show that (2) is true. If the suffix is not a computation of *corr_prog*, there exists a number n, $n \geq m$, such that the $(n+1)^{th}$ state is not obtained by executing *corr_prog* in the n^{th} state. Since in the n^{th} state either *action_zero* executes or *corr_prog* executes, it follows that in the n^{th} state *action_zero* executes, i.e., (1) is true. □

Lemma 9.11 In the computation of the corrector and the action at process 0, the value of $x.0$ in the n^{th} state of the computation is equal to the sum of the initial value of $x.0$ and the number of steps taken by process 0. Formally,

$$
\begin{aligned}
\forall seq :: \quad & run(action\_zero \| corr\_prog)(seq) \\
& \Rightarrow \\
& \quad x(seq(n))(0) \;=\; (x(seq(0))(0) \;+ \\
& \qquad\qquad num\_steps(action\_zero)(seq, n)) \ mod \ M
\end{aligned}
$$

Proof. We prove this lemma by induction on the length of the computation. In the initial state, this condition is trivially satisfied. In the induction case, we do a case-split on whether process 0 executes or whether *corr_prog* executes. In each case, the proof is straightforward. □

Lemma 9.12 In the computation of the corrector and the action at process 0, either $x.0$ takes on all possible values in the range $0..(M-1)$ or a state is reached where *corr_pred* is true. Formally,

$$
\begin{aligned}
& \forall seq :: run(action\_zero \| corr\_prog)(seq) \;\; \Rightarrow \\
& \qquad \forall v : 0 \leq v < M : (\exists n :: x(seq(n))(0) = v) \\
& \qquad \vee \; \exists n :: corr\_pred(seq(n))
\end{aligned}
$$

Proof. We prove this lemma by using Lemmas 9.10 and 9.11 . In Lemma 9.11, if the value of $x.0$ in the initial state is $v0$ then after process 0 executes $(v - v0) \ mod \ M$ steps, the value of $x.0$ will be v. By Lemma 9.10, either process 0 executes infinitely often or a state is reached where *corr_pred* is true. In the former case, we know that process 0 executes $(v - v0) \ mod \ M$ times and, hence, the value of $x.0$ is eventually v. In the latter case, the lemma is trivially true. □

Lemma 9.13 If $M \geq N+1$, then in any state there exists a value, say v, in the range $0..(M-1)$ such that the x values of all non-zero processes are different from v. Formally,

$$\forall s :: (\exists v : 0 \leq v < M : (\forall j : j \neq 0 : x(s)(j) \neq v))$$

Proof. Note that this lemma essentially states the pigeonhole principle: There are at most N distinct x values of non-zero processes and, hence, if $M \geq N+1$, there must exist a value that is different from the x values of all non-zero processes. We prove this lemma in the following steps:

(1) $|\{x.j : j \neq 0\}|$ is at most N,

(2) $(|\{v : 0 \leq v < M\} - \{x.j : j \neq 0\}|)$ is non-zero if $M \geq N+1$.

We use the set library in PVS in our proof of (1) and (2). This library defines various operations with sets such as union, intersection, difference, cardinality, etc, and provides some standard lemmas about them.

Given a state s, we define the set of x values of non-zero processes upto j, $nonz\_set\_upto(s)(j)$ as follows:

$$\begin{array}{lll} nonz\_set\_upto(s)(j) = \{\} & & \text{if } j=0 \\ & nonz\_set\_upto(s)(j-1) \cup \{x(s)(j)\} & \text{otherwise} \end{array}$$

By induction on j, we then prove that the cardinality of $nonz\_set\_upto(s)(j)$ is atmost j: The base case, $j=0$, is trivial since $nonz\_set\_upto(s)(0)$ is the empty set. For the induction case, we use the fact that $nonz\_set\_upto(j+1) = nonz\_set\_upto(j) \cup \{j+1\}$ and that the cardinality of the union is no greater than the sum of cardinalities. Now, observe that (1) is trivially true if we instantiate $j=N$.

To prove (2), we use the fact that the for any two sets X and Y, $|X - Y| \geq |X| - |Y|$. Letting X be the set $0..(M-1)$, and Y be the set x values of non-zero processes, we show that in any state there exists a missing value, i.e., a value that is different from the x values of all non-zero processes. □

To identify some missing value in state s, we define a constant $missing(s)$ which denotes some arbitrary value that is missing in state s .

Definition. Given a state s, $missing(s)$ is some arbitrary value in the set $(\{v : 0 \leq v < M\} - \{x.j : j \neq 0\})$ □

Remark. Note that the definition of $missing(s)$ is sensible only if $M \geq N+1$. Thus, all theorems that use this definition rely on this assumption. For brevity, however, we will not explicitly specify this assumption in the subsequent theorems. In PVS, we define $M \geq N+1$ as an axiom so that it can be omitted in the statement of the theorems.

Lemma 9.14 Let s be any state in the computation of the corrector and the action at process 0. The x value of any non-zero process in s is either present in the initial state of that computation or it is generated by process 0 in a state preceding s. Formally,

$$\begin{array}{ll} \forall seq, n :: & run(action\_zero \| corr\_prog)(seq) \\ \quad \Rightarrow & \\ & \forall j : j \neq 0 : \quad (\exists k :: x(seq(n))(j) = x(seq(0))(k)) \\ & \qquad\qquad\quad \vee\ (\exists m : m < n : x(seq(n))(j) = x(seq(m))(0)) \end{array}$$

Proof. We prove this lemma by induction on the length of the computation. The base case, the initial state, is trivial; the x values of all non-zero processes are present in the initial state.

For the inductive case, our proof obligation is that if the x value of a non-zero process is changed in the $(n+1)^{th}$ state then that new value is either present in the initial state or it is generated by process 0 in an earlier state. Towards this end, we first do a case-split on which process executes in the n^{th} step: If process 0 executes in the n^{th} step, the x values of non-zero processes remain unchanged. Thus, the proof obligation is trivially satisfied. If a non-zero process, say j, executes in the n^{th} step, only the value of $x.j$ is changed it is set to $x.(j-1)$. We then do a case-split on whether $j=1$ or $j \neq 1$. If $j=1$, we show that the value of $x.j$ in the $(n+1)^{th}$ state is generated by process 0 in the n^{th} state. If $j \neq 1$, by induction on the value of $x.(j-1)$, it follows that the new value of $x.j$ is either present in the initial state or it is generated by process 0 in an earlier state. $\square$

Lemma 9.15 In the computation of the corrector and the action at process 0, a state is reached that satisfies one of the following conditions: (1) $x.0$ is equal to a value missing in the initial state and the x values of non-zero processes are different from $x.0$, or (2) *corr_pred* is true. Formally,

$$
\begin{aligned}
\forall seq ::\quad & run(action\_zero \| corr\_prog)(seq) \\
\Rightarrow \quad & \\
& \exists n :: x(seq(n))(0) = missing(seq(0)) \;\wedge \\
& \qquad (\forall j : j \neq 0 : x(seq(n))(j) \neq missing(seq(0))) \\
& \vee\, \exists n :: corr\_pred(seq(n))
\end{aligned}
$$

Proof. From Lemma 9.12, in the computation of the corrector and the action at process 0, a state is reached where either $x.0 = missing(seq(0))$ is true or *corr_pred* is true. In the latter case, Lemma 9.15 is trivially satisfied. In the former case, we induct on the length of the computation to show that there exists a state, say s, such that

$x.0 = missing(seq(0))$ is true in state s, and $x.0 = missing(seq(0))$ is false in all states preceding s in the computation. We then use Lemma 9.14 to show that in state s, $x.0$ is different from the x values of all non-zero processes. By the construction of s, $x.0$ is never equal to $missing(seq(0))$ in any state preceding s. Moreover, by definition of $missing(seq(0))$, it is not present in the initial state at any non-zero process. Thus, from Lemma 9.14, it follows that in state s, the value of $x.0$ is different from the x values of non-zero processes. □

Lemma 9.16 If the corrector executes starting from a state where $x.0$ differs from the x values of all non-zero processes then in any state of that computation if $x.0$ is the same as $x.j$ then the x values of processes $0..j$ are the same as $x.0$. Formally,

$$
\begin{aligned}
\forall seq :: \quad & run(corr\_prog)(seq) \;\; \wedge \;\; (\forall j : j \neq 0 : x(seq(0))(j) \neq x(seq(0))(0)) \\
\Rightarrow & \\
& (\forall n, j :: x(seq(n))(j) = x(seq(0))(0) \Rightarrow \\
& \quad (\forall k : 0 \leq k \leq j : x(seq(n))(k) = x(seq(0))(0)))
\end{aligned}
$$

Proof. We prove this result by induction on the length of the computation as well. In the induction case, let $j1$ be a process that executes in the n^{th} step, and let $j2$ be any process that satisfies $x.j2 = x.0$ in the $(n+1)^{th}$ state. To prove that in the $(n+1)^{th}$ state the x values of $0..j2$ are the same as $x.0$, we do a case-split on whether $j2 < j1$, $j2 = j1$, or $j2 > j1$.

In the first case, we show that the x values of processes $0..j2$ remain unchanged and, hence, $x.j2 = x.0$ must be true in the n^{th} state. Therefore, in the $(n+1)^{th}$ state of the computation, the x values of processes $0..j2$ are the same as $x.0$.

In the second case, we show that it must be the case that in the n^{th} state, $x.(j2-1)$ is the same as $x.0$. Hence, in the n^{th} state, the x values of $0..(j2-1)$ are the same as

$x.0$. Since in the $(n+1)^{th}$ state $x.j2$ is the same as $x.0$ and the x values of processes $0..(j2-1)$ remain unchanged, it follows that in the $(n+1)^{th}$ state the x values of processes $0..j2$ are the same as $x.0$.

In the third case also, $x.j2$ remains unchanged. Thus, in the n^{th} state, the $x.j2 = x.0$ is true. Therefore, in the n^{th} state $x.j = x.0$ and $x.(j1-1) = x.0$ is also true. Thus, the action of process $j1$ is disabled. □

Theorem 9.17 The action at process 0 does not interfere with the corrector, i.e., the computation of $action\_zero \| corr\_prog$ converges to $corr\_pred$. Formally,

$$satisfies(action\_zero \| corr\_prog)(converges(true, corr\_pred))$$

Proof. We use Lemmas 9.10, 9.13 and 9.16 to prove the above lemma. From 9.13, a state, say s, is reached where $x.0$ differs from the x values of all non-zero processes. From Lemma 9.10, process 0 executes after s. Until 0 executes for the first time, the corresponding computation is a computation of the corrector. Moreover, when 0 executes $x.0$ is the same as $x.N$. Hence, by Lemma 9.16, the x values of all processes are identical. It follows that when 0 executes for the first time after state s, $corr\_pred$ is true. □

9.7 Discussion

9.7.1 Related Work.

Since Dijkstra presented the self-stabilizing token ring program in 1974, it has been proved using various techniques [6,27,51,56,65]. Of these, the proofs by Qadeer and Shankar [56] and Merz [51] have been verified by a theorem prover. Merz constructs a complicated variant function–consisting of the enabled processes, the distance between the x value of the process 0 and the missing value, etc.—and shows that it decreases in every step. In terms of number of interactions required with the theorem prover, it outperforms the proof presented here as well as that by Qadeer and Shankar. However, these reduced interactions come at a very high cost; the creativity required to find this variant function. Also, that proof is hard to comprehend since it does not match with the intuitive understanding of the token ring program.

Qadeer and Shankar closely follow the proof by Varghese [65], and their proof is simpler than that by Merz. However, since they try to prove the properties of the entire program, some of their proofs are more complex than they need to be. For example, they prove that each process eventually gets the token using the following variant: $p(j) = sum\{k : k \text{ has a token} : (i-j) \; mod \; N+1 \; \}$. One of the reasons they need such a variant function is that they are trying to prove that *starting from an arbitrary state* eventually each process will get the token. However, this property is more general than necessary; one only needs to prove that *after the invariant is established*, eventually each process will get the token. Since we prove the token circulation property only in the invariant states, we do not need such a variant function.

In related work on mechanical verification of self-stabilization, Prasetya [55] has verified a self-stabilizing routing program in a variant of UNITY logic [22] using the theorem prover HOL [32]. He also presents an elegant development of the theory needed in the verification but he seems to require a prohibitively high level of verification effort.

9.8 Advantages of Component-Based Mechanical Verification.

Fault-tolerant programs are often tricky and so need strong assurance; mechanical verification is a very strong form of assurance but previous examples were tours-de-force that required great insight and talent and are not readily transferable to other problems or other people. By way of contrast, our component-based approach is systematic and offers some hope of making these verifications routine. The detector-correctors theory and its application to Dijkstra's token ring program shows that the effort required as well as the amount of invention is reduced. We find that the advantages of component-based mechanical proofs are the same as that of component-based non-mechanical proofs. We discuss some of these advantages below.

Reusability for a variation of the token ring program. The modification of a component in the program preserves the correctness proofs of other components. We find that this property is useful in the mechanical verification of the resulting program as well. For example, observe that if the action at process 0 is changed so that $x.0$ is incremented by k (instead of 1), where k is relatively prime to M, the self-stabilization is preserved. After we proved the correctness of Dijkstra's token ring program, we verified the self-stabilization property of this new program and found that it took approximately 30 additional minutes to obtain the new proof (compared to approximately 4-5 days for the initial proof), and most of the proof was reused.

Reusability of proofs for other fault-tolerant programs. Lemma 9.10 shows that either process 0 executes infinitely often or the correction predicate is established. This proof only depends on the fact that the corrector satisfies its specification in isolation, and not on the actual programs and predicates involved. We, therefore, have extracted a simple interference-freedom lemma that is applicable in other programs. Likewise, Lemma 9.8, only depends upon the ordering between the corrector actions. Such a ordering exists in various programs—including most tree based programs. Therefore, the same proof technique can be used in those programs as well. Also, lemmas that relate to program compositions or interference-freedom techniques such as superposition and eventual termination can be reused in other fault-tolerant programs.

Role of assumptions. Observe that our proof clearly shows the assumption $M \geq N+1$ is not required for the correctness of the fault-intolerant program or the corrector; it is required only to prove that they do not interfere with each other. Thus, if we were to weaken this assumption—say because it is possible to prove stabilization when $M \geq N$— we will need to redo only the proofs that depend on this assumption, namely Lemmas 9.13, 9.15 and 9.17. Likewise, if we could relax this assumption, say by providing higher atomicity to process 0, we could reuse most of the proof.

9.9 Chapter Summary

In this chapter, we presented a component-based proof of Dijkstra's self-stabilizing token program that has been verified in PVS. To prove correctness of this self-stabilizing program, we needed to show two properties: (1) in the absence of faults, the program circulates a token along the ring, and (2) in the presence of faults, the program eventually recovers to a state from where the token circulation is restored. Following our philosophy of program decomposition, we decomposed the fault-tolerant program into the corresponding fault-intolerant program and the corrector. Then, we proved that property (1) is satisfied by focusing on the fault-intolerant program, and considering its execution starting from the invariant states. Subsequently, we proved property (2) by focusing on the corrector, and considering its execution starting from all states. Finally, we showed that the fault-intolerant program and the corrector do not interfere with each other.

Our case study illustrates that the advantages of program decomposition in non-mechanical proofs also apply to mechanical verification. It shows that by focusing on the component responsible for satisfying the property at hand, the proof of the required property is simplified. Also, it shows that the component-based approach readily supports design exploration as modifications to a program often permits the reuse of proofs. Moreover, it demonstrates that mechanical verification of fault-tolerant programs is less of a tour-de-force and more of a straightforward activity.

CHAPTER 10

CONCLUSION AND FUTURE WORK

In this dissertation, we presented a methodology that permits a uniform design of a rich class of fault-tolerant systems. Such a methodology is bound to raise several questions. We first answer some of these questions (cf. Section 10.1). Then, we summarize the contributions in this thesis and discuss how it will affect the design of fault-tolerant systems (cf. Sections 10.2 and 10.3). Finally, we outline possible extensions of this work (cf. Section 10.4).

10.1 Discussion

We have represented faults as state perturbations. This representation readily handles transient faults, but does it also handle permanent faults? intermittent faults? detectable faults? undetectable faults?

All these faults can indeed be represented as state perturbations. The token ring case study illustrates the use of state perturbations for various classes of transient faults. The repetitive Byzantine agreement example and the multitolerant reset example analogously illustrate the representation of permanent faults, detectable faults and undetectable faults.

It is worth pointing out that representing permanent and intermittent faults, such as Byzantine faults and fail-stop and repair faults, may require the introduction of auxiliary variables [6, 8]. For example, to represent Byzantine faults that affects a process j, in Chapter 7, we introduced an auxiliary boolean variable $b.j$ that is true iff j is Byzantine. Similarly, to represent fail-stop and repair faults that affects a process j, in Chapter 8, we have introduced an auxiliary boolean variable $up.j$ that is true iff j has not fail-stopped.

How would our method of considering the fault-classes one-at-a-time compare with a method that considers them altogether?

There is a sense in which the one-at-a-time and the altogether methods are equivalent: programs designed by the one method can also be designed by the other method. To justify this informally, let us consider a program p designed by using the altogether method to tolerate fault-classes $F1$, $F2$, ... , Fn. Program p can also be designed using the one-at-a-time method as follows: Let $p1$ be a subprogram of p that tolerates $F1$. This is the program designed in the first stage of the one-at-a-time method. Likewise, let $p2$ be a subprogram of p that tolerates $F1$ and $F2$. This is the program designed in the second stage of the one-at-a-time method. And so on, until p is designed. To complete the argument of equivalence, it remains to observe that a program designed by the one-at-a-time n-stage method can trivially be designed by the altogether method.

In terms of software engineering practice, however, the two methods would exhibit differences. Towards identifying these differences, we address three issues: (i) the structure of the programs designed using the two methods, (ii) the complexity of using them, and (iii) the complexity of the programs designed using them.

On the first issue, the stepwise method may yield programs that are better structured. This is exemplified by our hierarchical token ring program (cf. Chapter 6 which consists of three layers: the basic program that transmits the token, components for the case when at least one process is not corrupted, and components for the case when all processes are corrupted

On the second issue, since we consider one fault-class at a time, the complexity of each step is less than the complexity of the altogether program. For example, in the token ring program, we first handled the case where the state of some process is not corrupted. Then, we handled the only case where the state of all processes is corrupted. Thus, each step was simpler than the case where we would need to consider both these cases simultaneously.

On the third issue, it is possible that considering all fault-classes at a time may yield a program whose complexity is (in some sense) optimal with respect to each fault-class, whereas the one-at-a-time approach may yield a program that is optimal for some, but not all, fault-classes. This suggests two considerations for the use of our method. One, the order in which the fault-classes are considered should be chosen with care. In our experience, we have found that it is desirable to deal with the 'common case' fault first. Also, it is desirable to deal with masking fault-tolerance first and stabilizing fault-tolerance last. (Again, in principle, programs designed with one order can be designed by any other order. But, in practice, different orders may yield different programs, and the complexity of these programs may be different.) And, two, in choosing how to design the tolerance for a particular fault-class, a "lookahead" may be warranted into the impact of this design choice on the design of the tolerances to the remaining fault-classes.

How does our compositional method affect the trade-offs between dependability properties?

Our method makes it possible to reason about the trade-offs locally, i.e., focusing attention only on the components corresponding to those dependability properties, as opposed to globally, i.e., by considering the entire program. Thus, our method facilitates reasoning about trade-offs between dependability properties.

Moreover, as can be expected, if the desired dependability properties are impossible to cosatisfy, it will follow that there do not exist components that can be added to the program while complying with the interference-freedom requirements of our method.

How do detectors in our work compare with the failure detectors [20] of Chandra and Toueg?

In [20], Chandra and Toueg introduced the concept of failure detectors that are necessary and sufficient in solving various problems in asynchronous distributed systems [31]. Each of these failure detectors guarantees that any process that crashes is eventually suspected by every correct processes. Depending upon how processes may be suspected incorrectly, these failure detectors are classified into four categories:

- **Perfect (P)** A process is suspected only if it has crashed.
- **Strong (S)** Some non-failed process is never suspected.

- **Eventually perfect ($\diamond$ P)** Eventually the program reaches a state from where a process is suspected only if it has crashed.

- **Eventually strong ($\diamond$ S)** Eventually the program reaches a state from where some non-failed process is never suspected.

The failure detectors in each of these categories can be expressed in terms of detectors defined in Chapter 3. The detection predicate (X) for all the failure detectors is $\neg up.j$, where $up.j$ is true iff j has not failed, and the witness predicate (Z) is $wit.k.j$ which is true iff k suspects that j has failed. Observe that the detection predicate is stable.

Note that P always satisfies the *Safety, Progress* and *Stability* of the detector. Thus, we have "$wit.k.j$ detects $\neg up.j$" in P from $true$. Also, $\diamond P$ eventually reaches a state from where further computation satisfies the *Safety, Progress* and the *Stability* of the detector. Therefore, $\diamond P$ refines the nonmasking tolerance specification of "$wit.k.j$ detects $\neg up.j$" from $true$. S and $\diamond S$ also satisfy similar detector specification for a limited number of processes. It follows that the failure detectors used in [20] are instances of our detectors.

Our detectors are, however, more general than in [20] because the detection predicate in our detector can be more general. Also, unlike our 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.

10.2 Contributions

In this dissertation, we showed that the design of fault-tolerance includes two basic concepts: detection and correction. We identified the components that are necessary and sufficient for them, showed how these components can be designed in an hierarchical manner, and how to design components required to transform a given fault-intolerant program into a fault-tolerant one.

Regarding the generality of components, in Chapters 3-5, we identified, the class of fault-tolerant programs for which detectors and correctors are necessary and sufficient. Also, in Chapter 7, we showed that the components used in existing methods such as replication and Schneider's state machine approach can be alternatively designed in terms of detectors and correctors. These results, in turn, imply that the use of detectors and correctors generalizes these methods in that the class of programs that can be made fault-tolerant using detectors and correctors is a superset of the class programs that can be designed using these methods.

Using the detectors and correctors, in Chapter 6, we presented a systematic method for designing multitolerant programs. We illustrated this method in the design of repetitive Byzantine agreement (cf. Chapter 7) and distributed reset (cf. Chapter 8). This method has also been used in design of several other multitolerant programs such as leader election [10], mutual exclusion [12], network management [42], resource synchronization [41]. It follows that the method can deal with multiple types of faults, can be used to incrementally add fault-tolerance to a new type of fault and is not application dependent. Also, due to the generality of detectors and correctors, the method can be used to make a rich class of systems fault-tolerant.

We also showed that the decomposition of a fault-tolerant program into its components is useful in verification of that fault-tolerant program. We illustrated this by presenting a mechanically verified proof of Dijkstra's token ring program that is based on decomposing the program into a fault-intolerant program and a corrector.

10.3 Impact

We would like to note that the use of detectors and correctors not only generalizes the existing fault-tolerance methods but it also differs in the design philosophy. Existing methods such as replication, Schneider's state machine and checkpointing-and-recovery add components of certain type to a fault-intolerant program. These methods, however, do not provide a formal basis as to why the added components are the right type of components in designing fault-tolerance. For example, the basic components used in replication based methods are comparators and voters. However, these replication based methods (apart form intuition and experience) do not address why comparators and voters are the right components to use in designing fault-tolerance. By way of contrast, we justify the components presented in this dissertation by showing that they are both necessary and sufficient. Subsequently, we also illustrate that these components are useful in the design and analysis of multi-tolerant programs. Moreover, by designing the components used in replication and Schneider's state machine approach, we also identify the role of the components used in those methods. In sum, our work identifies the components that are useful in the design and analysis of fault-tolerance and that provide a validation for the components used in existing methods.

10.4 Future Directions

Our work on component based design of fault-tolerance has opened up several new directions for further research. Some of these are outlined below.

From the fault-tolerant programs we developed using detectors and correctors, we have observed that detectors and correctors required in one program as well as across different programs are often similar. Therefore, we will be 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 will also be working on mechanized verification and synthesis of component based fault-tolerant programs. Towards mechanized verification, we will extend the case study discussed in Chapter 9 and encode the theory of detectors and correctors into PVS. Also, we plan to investigate whether other techniques such as phased reasoning [60] based on convergence stairs [33] and hierarchical design of components offer the same advantage in mechanical verification as they do in non-mechanical verification. We also plan to investigate the use of program decomposition in mechanical verification of multitolerant programs. Towards mechanized synthesis, we will be working towards automating our design method so that the required fault-tolerance components can be synthesized.

Another extension of this work lies in the application of our design method for developing network protocols for problems such as congestion control, network management and routing. We will also be working on the development of component based security protocols. We believe that protocols satisfying security properties such as authentication, authorization, privacy, non-repudiation and intrusion detection is possible using detectors and correctors alone, and we plan to apply our component based method for designing such protocols. Higher order security properties such as information flow may, however, require additional components, and we plan to investigate the components required for such security properties.

BIBLIOGRAPHY

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] Y. Afek and E. Gafni. Bootstrap network resynchronization. *Proceedings of 10th ACM Symposium on Principles of Distributed Computing*, pages 295–307, 1991.

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

[4] B. Alpern and F. B. Schneider. Proving boolean combinations of deterministic properties. *Proceedings of the Second Symposium on Logic in Computer Science*, pages 131–137, 1987.

[5] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer-Verlag LNCS:725*, pages 174–188, 1993.

[6] A. Arora. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, 1992.

[7] A. Arora. Efficient reconfiguration of trees: A case study in the methodical design of nonmasking fault-tolerance. *Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault-Tolerance*, pages 110–127, 1994. *Science of Computer Programming* to appear.

[8] 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.

[9] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[10] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995, pages 174–185*, 14:174–185, 1995.

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

[12] 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.

[13] 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. An extended version of this paper has been invited to *IEEE Transactions on Computers.*

[14] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 254–263, 1994.

[15] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[16] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilizing by local checking and global reset. *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857*, pages 326–339, 1994.

[17] O. Babaoglu. On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, November 1987.

[18] R.J.R. Back and K. Sere. Stepwise refinement of parallel programs. *ACM Transactions on Software Engineering and Methodology*, 3(4):133–180, 1994.

[19] F. Bastani, I.-L Yen, and I. Chen. A class of inherently fault-tolerant diffusing programs. *IEEE Transactions on Software Engineering*, 14:1432–1442, 1988.

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

[21] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

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

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

[24] D. M. Dhamdhere and S. S. Kulkarni. A token based k resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50:151–157, 1994.

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

[26] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[27] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[28] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gastern. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[29] S. Dolev and T. Herman. Superstabilzing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[30] S. Finn. Resynch procedures and fail-safe network protocol. *IEEE Transactions on Communication*, 27(6):840–845, 1979.

[31] M. J. Fischer, N. .A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):373–382, 1985.

[32] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[33] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

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

[35] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *ACM Symposium on Principles of Distributed Computing*, 1996.

[36] S. Katz and K. Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7:17–26, 1993.

[37] R. Koo and S. Toueg. Checkpointing and rollback-recovery in distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, Jan 1987.

[38] S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science*, 1998(4), December 1998.

[39] 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, dec 1997.

[40] S. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, October 1997.

[41] S. S. Kulkarni and A. Arora. Low-cost fault-tolerance in barrier synchronizations. *International Conference on Parallel Processing*, pages 132–139, August 1998.

[42] S. S. Kulkarni and A. Arora. Once-and-forall management protocol (OFMP). *Proceedings of the Fifth International Conference on Network Protocols*, October 1997.

[43] S. S .Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.

[44] T. H. Lai and L. F. Wu. An (n-1) resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):63–78, 1995.

[45] Y. Lakhnech and M. Siegel. Deductive verification of stabilizing systems. *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 201–216, 1997.

[46] L. Lamport and N. Lynch. *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods.* Elsevier Science Publishers B. V., 1990.

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

[48] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, pages 281–302, 1981.

[49] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. *ACM Symposium on the Principles of Distributed Computing*, 1986.

[50] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[51] S. Merz. Mechanical verification of self-stabilizing token ring. Personal communication, 1998.

[52] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[53] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[54] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[55] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, pages 399–415, 1997.

[56] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

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

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

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

[60] M. Siegel and F. Stomp. Extending the limits of sequentially phased reasoning. In P. S. Thiagarajan, editor, *Foundations of software technology and theoretical computer science, Lecture Notes in computer science 880*, 1994.

[61] M. Spezialetti and P. Kearns. Efficient distributed snapshots. *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, May 1986.

[62] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computing Systems*, 1985.

[63] G. Tel. *Structure of Distributed Algorithms.* PhD thesis, University of Utrecht; also published by Cambridge University Press, 1989.

[64] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, June 1992.

[65] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

[66] Y. Zhao and F. B. Bastani. A self-adjusting algorithm for Byzantine agreement. *Distributed Computing*, 5:219–226, 1992.

APPENDIX A

NOTATION

Symbols	
p, q, c, d	programs
d	detectors
c	correctors
F	faults
s, st	state
ac	action
R, S, T, U, V, X, Z, g	state predicates
X, sf	detection/correction predicates
Z	witness predicates
$SPEC$	problem specification
$SSPEC$	safety specification

Compositions	
$guard \ \wedge \ action$	restriction of *action*
$guard \ \wedge \ program$	restriction of *program*
$p \ \| \ q$	parallel
$g \wedge p$	restriction
$p \ ; \ q$	sequential

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