Component Based Design of Multitolerance

Anish Arora *Sandeep S. Kulkarni*

Department of Computer and Information Science [1]
The Ohio State University
Columbus, Ohio 43210 USA

Abstract

The concept of multitolerance abstracts problems in system dependability and provides a basis for improved design of dependable systems. In the abstraction, each source of undependability in the system is represented as a class of faults, and the corresponding ability of the system to deal with that undependability source is represented as a type of tolerance. Multitolerance thus refers to the ability of the system to tolerate multiple fault-classes, each in a possibly different way.

In this paper, we present a component based method for designing multitolerance. Two types of components are employed by the method, namely detectors and correctors. A theory of detectors, correctors, and their interference-free composition with intolerant programs is developed, that enables stepwise addition of components to provide tolerance to a new fault-class while preserving the tolerances to the previously added fault-classes. We illustrate the method by designing a fully distributed, multitolerant program for a token ring.

Keywords : formal methods, compositional design, interference-freedom, stepwise design, detectors, correctors, dependability, fault-tolerance, graceful degradation

[1] Email: {anish,kulkarni}@cis.ohio-state.edu ; Web: http://www.cis.ohio-state.edu/{~anish,~kulkarni };
Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ; Research supported in part by
NSA Grant MDA904-96-1-0111, NSF Grant CCR-93-08640, and OSU Grant 221506

1 Introduction

Dependability is an increasingly relevant system-level requirement that encompasses the ability of a system to deliver its service in a desirable manner, in spite of the occurrence of faults, security intrusions, safety hazards, configuration changes, load variations, etc. Achieving this ability is difficult, essentially because engineering a system for the sake of one dependability property, say the availability in the presence of faults, often interferes with another desired dependability property, say the security in presence of intrusions.

In this paper, to effectively reason about multiple dependability properties, we introduce the concept of multitolerance. Each source of undependability is treated as a class of "faults" and each dependability property is treated as a type of "tolerance". Thus, multitolerance refers to the ability of a system to tolerate multiple classes of faults, each in a possibly different way.

It is worthwhile to point out that multitolerance has other related applications as well. One is to reason about graceful degradation with respect to a progressively increasing fault-classes. Another is to guarantee different qualities of service (QoS) with respect to different user requirements and traffics. A third one is to reason about adaptivity of systems with respect to different modes of environment behavior.

Although there are many examples of multitolerant systems in practice [1 − 3] and there exists a growing body of research that presents instances of multitolerant systems [4−10], we are not aware of previous work that has considered the systematic design of multitolerance. Towards redressing this deficiency, we present in this paper a formal method for the design of multitolerant systems.

To deal with the difficulty of interference between multiple types of tolerances, our method employs the use of components. More specifically, a multitolerant system designed using the method consists of an intolerant system and a set of components, one for each desired type of tolerance. Thus, the method reduces the complexity of the design to that of the designing the components and that of correctly adding them to the intolerant system. Moreover, it enables reasoning about each type of tolerance and the interferences between different types of tolerance at the level of the components themselves, as opposed to involving the whole system.

The method further reduces the complexity of adding multiple components to an intolerant system by adding each component in a stepwise fashion. In other words, the method considers the fault-classes that an intolerant system is subject to in some fixed total order, say $F1$ .. Fn. A component is added to intolerant system so that it tolerates $F1$ in a desirable manner. The resulting system is then augmented with another component so that it tolerates $F2$ in a desirable manner *and* its tolerance to $F1$ is preserved. This process of adding a new tolerance and preserving all old tolerances is repeated until all n fault-classes are accounted for. It follows that the final system is multitolerant with respect to $F1$ .. Fn.

Our method employs two types of components: *detectors* and *correctors*. Intuitively, a detector is a component that "detects" whether some predicate is satisfied by the system state; and, a corrector is a component that detects whether some predicate is satisfied by the system state and also "corrects" the system state in order to satisfy that predicate whenever the predicate is not satisfied. Informally speaking, detectors are used to ensure that each step of the system is safe, while correctors are used to ensure that the system eventually reaches a state from where its specification is (re)satisfied.

The rest of this paper is organized as follows. In Section 2, we give formal definitions of programs, faults, fault-tolerances, and types of fault-tolerances. In Sections 3 and 4, we define detectors and correctors, discuss their role in the design of fault-tolerant systems, and illustrate how they can be designed hierarchically and efficiently. In Section 5, we present a theory of non-interference for composing detectors and correctors components with intolerant programs. In Section 6, we define multitolerance and present our formal method for designing multitolerance. In Section 7, we illustrate our method by designing a multitolerant token ring program. Finally, we discuss some issues raised by our methodology in Section 8 and make concluding remarks in Section 9.

2 Preliminaries

In this section, we recall a formal representation of programs, faults, and fault-tolerances. The formalization is adapted from earlier work of the first author with Mohamed Gouda [10], with the exception of the portion on fail-safe tolerance which is new.

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

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

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

For convenience in specifying an action as a restriction of another action, we use the notation

$$\langle \text{name}' \rangle \ :: \ \langle \text{guard}' \rangle \ \wedge \ \langle \text{name} \rangle$$

to define an action $\langle \text{name}' \rangle$ whose guard is obtained by restricting the guard of action $\langle \text{name} \rangle$ with $\langle \text{guard}' \rangle$, and whose statement is identical to the statement of action $\langle \text{name} \rangle$. Operationally speaking, $\langle \text{name}' \rangle$ is executed only if the guard of $\langle \text{name} \rangle$ and the guard $\langle \text{guard}' \rangle$ are both true. Likewise, to conveniently specify a program as a restriction of another program, we use the notation $\langle \text{guard} \rangle \wedge \langle \text{program} \rangle$ to define a program consisting of the set of actions $\langle \text{guard} \rangle \wedge ac$ for each action ac of $\langle \text{program} \rangle$.

State and State Predicate. Let p be a program. A state of p is defined by a value for each variable of p, chosen from the predefined domain of the variable. A state predicate of p is a boolean expression over the variables of p. An action of p is enabled in a state iff its guard (state predicate) evaluates to true in that state.

Computation. A computation of p is a fair, maximal sequence of steps; in every step, an action of p that is enabled in the current state is chosen and the statement of the action is executed atomically. 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, i.e., the computation continues until a state is reached in which no action is enabled. Note that the set of computations is suffix closed.

Problem Specification. The problem specification that p satisfies consists of a "safety" specification and a "liveness" specification[11]. A safety specification identifies a set of "bad" finite computation prefixes that should not appear in any program computation. Dually, a liveness specification iden-

tifies a set of "good" computation suffixes such that every computation has a suffix that is in this set. (Incidentally, our definition of liveness is stronger than Alpern and Schneider's definition [11]: the two definitions become identical if the liveness specification is fusion closed; i.e., if computations $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ satisfy the liveness specification then computations $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$ also satisfy the liveness specification, where α, β are finite computation prefixes, γ, δ are computation suffixes, and x is a program state.)

A computation of p is "correct" with respect to the problem specification iff it satisfies both the safety specification and the liveness specification.

We assume that problem specifications are suffix closed. It follows from this assumption that suffixes of correct computations are also correct computations with respect to the problem specification.

Some examples of safety specifications, namely Hoare-triples and "closure", and some examples of liveness specifications, namely "leads to" and "converges to" are defined below.

Hoare-triples. Let S and R be state predicates of p. The triple $\{S\}\ ac\ \{R\}$ holds iff in any state where S holds and ac is enabled, executing the statement of ac yields upon termination a state where R holds. We use the notation $\{S\}\ p\ \{R\}$ to abbreviate $(\forall ac : ac \text{ is an action of } p : \{S\}\ ac\ \{R\})$. That is, $\{S\}\ p\ \{R\}$ holds iff in any state where S holds, executing the statement of any enabled action of p yields upon termination a state where R holds.

Closure. An action ac of p "preserves" S iff $\{S\}\ ac\ \{S\}$. S is "closed" in a set of actions iff each action in that set preserves S. Note that, by definition, the predicates *true* and *false* are trivially closed in any program.

Leads to. Let T be a state predicate of p. T leads to S in p iff starting from any state where T holds, every computation of p reaches a state where S holds. Note that the leads to relation is reflexive (T leads to T) and transitive (if T leads to S and S leads to R, then T leads to R).

Converges to. T converges to S in p iff both T and S are closed in p and T leads to S in p. Note that the converges to relation is transitive.

Invariant. One state predicate is distinguished as the invariant of p. The invariant of p is a closed state predicate such that every computation of p that starts at an invariant state is correct with respect to the problem specification.

Note that, since the invariant is closed, the invariant holds at all states in all computation that start at invariant states. Thus, the the invariant characterizes the set of all states that p reaches from some set of states. It follows that reachability analysis is one way of calculating the invariant. Note also that while many state predicates other than the invariant may be closed in p, only some of them may hold at states where the invariant holds; *true* and *false* respectively provide an example and a counterexample of such a predicate (the latter assuming that the invariant is not itself *false*).

Techniques for the design of an invariant of the program have been articulated by Dijkstra [12], using the notion of auxiliary variables, and by Gries [13], using the heuristics of state predicate ballooning and shrinking. Techniques for the mechanical calculation of an invariant predicate have been discussed by Alpern and Schneider [14].

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).

Fault-span. To formalize the behavior of a program p that tolerates a fault-class F, we observe that, just as the invariant of p characterizes the set of states that p reaches in the absence of F, there exists a predicate that likewise characterizes the set of states that p reaches in the presence of F. We call this latter predicate the "fault-span" of p with respect to F.

Note that a fault-span of p characterizes a set of states that includes the set of states characterized by the invariant of p. Hence, the fault-span holds at each state where the invariant of p holds. Also, like the invariant, a fault-span of p is, by definition, closed in p. Moreover, if an action in F is executed in a state where the fault-span holds, the resulting state is also one where the fault-span holds, i.e., the fault-span of p with respect to F is preserved by each action in F. (In the rest of the paper, we will ambiguously abbreviate the phrase "preserved by each action in F" by "closed in F".)

Fault-tolerances. We are now ready to define what it means for a program p to tolerate fault-class F. We say that p is F-tolerant for the invariant S iff there exists a predicate T that satisfies the following three requirements:

- At any state where S holds, T also holds; i.e., $S \Rightarrow T$
- Starting from any state where T holds, if any action in p or F is executed, the resulting state is also one where T holds; i.e., T is closed in p and T is closed in F.
- Starting from any state where T holds, every computation of p alone eventually reaches a state where S holds; i.e., T converges to S in p.

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

It is important to note that there may be multiple such predicates T for which p satisfies the above three requirements. Each of these multiple T predicates captures a (potentially different) type of fault-tolerance of p. We will exploit this multiplicity in Section 6 in order to define multitolerance.

Types of fault-tolerances. We now classify three types of fault-tolerances that a program can exhibit, namely *masking, nonmasking*, and *fail-safe* tolerance, using the above definition of F-tolerance. This classification of tolerances is based upon how the problem specification is met in the presence of faults.

Of the three, masking is the strictest type of tolerance: in the presence of faults, the program always satisfies its safety properties and, when faults stop occurring, the program eventually resumes satisfying its liveness properties. Nonmasking is less strict than masking: in the presence of faults, the program need not satisfy its safety properties but, when faults stop occurring, the program eventually resumes satisfying both its safety and liveness properties. Fail-safe is also less strict than masking: in the presence of faults, the program always satisfies its safety properties but, when faults stop occurring, the program need not resume satisfying its liveness properties. Formally, these three types of tolerance may be expressed in terms of the definition of F-tolerance, as follows:

1. p is *masking tolerant* to F for S iff p is F-tolerant for S and S is closed in F. (In other words, if a fault in F occurs in a state where S holds, p continues to be in a state where S holds.)

2. p is *nonmasking tolerant* to F for S iff p is F-tolerant for S and S is not closed in F. (In other words, if a fault in F occurs in a state where S holds, p may be perturbed to a state where S is violated. However, the program recovers to a state where S holds.)

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

3. p is *fail-safe tolerant* to F for S iff there exists a predicate R such that p is F-tolerant for $S \vee R$, $S \vee R$ is closed in p and in F, and all computations of p that start in a state where R holds satisfy the safety properties of the problem specification that p satisfies. (In other words, if a fault in F occurs in a state where S holds, p may be perturbed to a state where S or R holds. In the latter case, the subsequent execution of p satisfies the safety properties of p but not necessarily the liveness properties.)

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

3 Detectors

In this section, we define the first of two building blocks which are sufficient for the design of fault-tolerant programs, namely detectors. We also present the properties of detectors, show how to construct them in a hierarchical and efficient manner, and discuss their role in the design of fault-tolerance.

As mentioned in the introduction, intuitively, a detector is a program that "detects" whether a given state predicate holds in the current system state. Implementations of detectors abound in practice: Wellknown examples include comparators, error detection codes, consistency checkers, watchdog programs, snoopers, alarms, snapshot procedures, acceptance tests, and exception conditions.

Definition. Let X and Z be state predicates of a program d and U be a state predicate that is closed in d. We say that "Z detects X in d for U" iff the following three conditions hold:

- $(Safeness)$ $U \Rightarrow (Z \Rightarrow X)$; i.e., at any state where U holds, if Z holds then X also holds,
- $(Progress)$ $(U \wedge X)$ leads to $(Z \vee \neg X)$ in d; i.e., starting from any state where $U \wedge X$ holds, every computation of d either reaches a state where Z holds or a state where X is violated,
- $(Stability)$ $\{U \wedge Z\}\ d\ \{Z \vee \neg X\}$, i.e., starting from any state where $U \wedge Z$ holds, d violates Z only if it also violates X.

The *Safeness* condition implies that a detector d never lets the predicate Z "witness" the detection predicate X incorrectly when executed in states where U holds. The *Progress* condition implies that in any computation of d starting from a state where U holds, if X holds continuously then d eventually detects X by truthifying Z. The *Stability* condition implies that once Z is truthified, it continues to hold unless X is violated.

Remark. If the detection predicate X is closed in d, our definition of the detects relation reduces to one given by Chandy and Misra [15]. 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.)*

In the sequel, we will implicitly assume that the specification of a detector d (dn) is "Z detects X in d for U" (respectively, "Zn detects Xn in dn for Un"). Also, we will implicitly assume that U (Un) is closed in d (respectively, dn).

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

Lemma 3.0 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 for U
- If Z detects X in d for U, and X detects Z in d for U
  then $U \Rightarrow (Z \equiv X)$
- If Z detects Y in d for U, and Y detects X in d for U
  then Z detects X in d for U □

Lemma 3.1 Let V be a state predicate such that $U \wedge V$ is closed in d. The following statements hold.

- If Z detects X in d for U
  then Z detects X in d for $U \wedge V$
- If Z detects X in d for U, and $V \Rightarrow X$
  then $Z \vee V$ detects X in d for U
- If Z detects X in d for U, and $Z \Rightarrow V$
  then Z detects $X \wedge V$ in d for U □

Compositions. Regarding the construction of detectors, there are cases where a detection predicate X cannot be witnessed atomically, i.e., by executing at most one action of a detector program. To detect such predicates, we give compositions of "small" detectors that yield "large" detectors in a hierarchical and efficient manner. In particular, given two detectors, $d1$ and $d2$, we compose them in two ways: (i) in parallel and (ii) in sequence.

Parallel composition of detectors. In the parallel composition of $d1$ and $d2$, denoted by $d1 [] d2$, both $d1$ and $d2$ execute in an interleaved fashion. 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 3.2 Given $Z1$ detects $X1$ in $d1$ for U and $Z2$ detects $X2$ in $d2$ for U.
If the variables of $d1$ and $d2$ are mutually exclusive
then $Z1 \wedge Z2$ detects $X1 \wedge X2$ in $d1 [] d2$ for U

Proof. The *Safeness* condition of $d1 [] d2$ follows trivially from the *Safeness* of $d1$ and of $d2$. For the *Progress* condition, we consider two cases, (i) a computation of $d1 [] d2$ violates $X1 \wedge X2$ and (ii) a computation of $d1 [] d2$ does not violate $X1 \wedge X2$: In the first case, *Progress* holds trivially. In the second case, eventually $Z1$ is satisfied, and by *Stability* of $d1$, $Z1$ continues to hold in the execution of $d1$. Moreover, since the variables of $d1$ and $d2$ are disjoint, $Z1$ continues to be satisfied in $d2$.

Likewise, $Z2$ is eventually satisfied and continues to be satisfied. Thus, *Progress* holds. Finally, the *Stability* condition holds since $Z1 \wedge Z2$ can be falsified only if either $X1$ or $X2$ is violated. □

In $d1 [\!] d2$, since $d1$ and $d2$ perform their detection concurrently, the time required for detection of $X1 \wedge X2$ is the maximum time taken to detect $X1$ and to detect $X2$. Also, the space complexity of $d1 [\!] d2$ is the sum of the space complexity of $d1$ and $d2$, since the state space of $d1 [\!] d2$ is the union of the state space of $d1$ and of $d2$.

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 $Z1$ holds. Formally, the sequential composition of $d1$ and $d2$ is the program whose set of variables is the union of the variables of $d1$ and $d2$ and whose set of actions is the union of the actions of $d1$ and of $Z1 \wedge d2$.

We postulate 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 ';' ($g \wedge (d1; d2) = (g \wedge d1); (g \wedge d2)$).

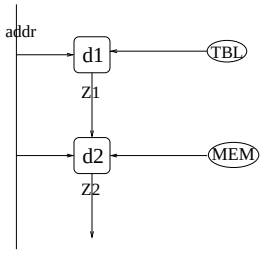
Suppose, again, that the variables of $d1$ and $d2$ are mutually exclusive. In this case, starting from any state where $X1 \wedge X2$ holds continuously, $d1$ eventually truthifies $Z1$. Only after $Z1$ is truthified are the actions of $d2$ executed; these actions eventually truthify $Z2$. Since $Z2$ is truthified only when $Z1$ (and, hence, $X1$) and $X2$ hold, it also follows that $U \Rightarrow (Z2 \;\Rightarrow\; X1 \wedge X2)$ provided we assume $U \Rightarrow (Z2 \;\Rightarrow\; X1)$.

Theorem 3.3 Given $Z1$ detects $X1$ in $d1$ for U and $Z2$ detects $X2$ in $d2$ for $U \wedge X1$.

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

In $d1; d2$, the time (respectively, space) taken to detect $X1 \wedge X2$ is the sum of the time (respectively, space) taken to detect $X1$ and to detect $X2$. The extra time taken by $d1; d2$ as compared to $d1 [\!] d2$ is warranted in cases where the witness predicate $Z2$ can be witnessed atomically but $Z1 \wedge Z2$ cannot.

Example: Memory access. Let us consider a simple memory access program that obtains the value stored at a given address (cf. Figure 1). The program is subject to two fault-classes: The first consists of protection faults which cause the given address to be corrupted so that it falls outside the valid address space, and the second consists of page faults which remove the address and its value from the memory.



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

Figure 1: **Memory access**

For tolerance to the first fault-class, there is a detector $d1$ that uses the page table TBL to detect whether the address $addr$ is valid. For tolerance to the second fault-class, there is another detector $d2$ that detects if the given address is in the memory MEM. Formally, these detectors are as follows:

(for simplicity, we assume that TBL is a set of valid addresses and MEM is a set of objects of the form $\langle addr, value \rangle$):

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

Proposition 3.4

- $Z1$ detects $X1$ in $d1$ for $U1$
- $Z2$ detects $X2$ in $d2$ for $U1$
- $Z2$ detects $X2$ in $d2$ for $U1 \wedge X1$

Note that an appropriate choice of initial state in $U1$ would be one where both $Z1$ and $Z2$ are false. Note also that, in $U1$, $Z1$ is truthified only when $X1$ holds and that $Z2$ is truthified only when $Z1$ and $X2$ both hold.

To detect $X1 \wedge X2$, we may compose $d1$ and $d2$ sequentially: $d1$ would first detects $X1$, and then $d2$ would detect $X2$. From Theorem 3.2 and Proposition 3.4, we get:

Proposition 3.5 $Z2$ detects $X1 \wedge X2$ in $d1; d2$ for $U1$

Application to design of fault-tolerance. Detectors suffice to ensure that execution of a program preserves its safety specification. To see this, recall that a safety specification essentially rules out certain finite prefixes of program computation. Now, consider any prefix of a computation that is not ruled out by the safety specification. Execution of a program action starting from this prefix preserves the safety specification iff the elongated prefix is not ruled out by the safety specification. It follows that for each program action there exists a set of computation prefixes from which execution of that action preserves the safety specification.

In other words, for each program action there exists a state predicate such that execution of that action in any state satisfying that predicate preserves the safety specification. (This translation from a set of computation prefixes to a state predicate assumes that auxiliary state variables may be introduced, which in the worst case record the history of the computation steps.) Now, if detectors can be added to the program that witness each of these state predicates, and each program action can be restricted to execute only if its corresponding witness predicate is true, the resulting program satisfies the safety specification.

To design fail-safe tolerance to F, we need to ensure that execution of p, in the presence of F, always satisfies the safety specification of p. It follows that detectors suffice for the design of fail-safe tolerance. Likewise, to design masking tolerance to F, we need to ensure that execution of p, in the presence of F, always satisfies its safety specification and, after execution of actions in F, resumes satisfying both its safety and liveness specification. Regarding safety, it suffices that detectors be added to p. (Regarding liveness, it suffices that correctors be added to p, as discussed in the next section.)

Detectors can also play a role in the design of nonmasking tolerance: They may be used to detect whether the program is perturbed to a state where its invariant does not hold. As discussed in the next section, such detectors can be systematically composed with correctors that restores the program to a state where its invariant is satisfied.

4 Correctors

In this section, we discuss the second set of building blocks, namely correctors, in a manner analogous to our discussion of detectors.

As mentioned in the introduction, intuitively, a corrector is a detector that also "corrects" the program state whenever it detects that its detection predicate does not hold in the current system state. Implementations of correctors abound in practice: Wellknown examples include voters, error correction codes, reset procedures, rollback recovery, rollforward recovery, constraint (re)satisfaction, exception handlers, and alternate procedures in recovery blocks.

Definition. Let X and Z be state predicates of a program c and U be closed in c. We say that "Z corrects X in c for U" iff Z detects X in c for U and the following condition holds:

> (*Convergence*) U converges to X in c, i.e., starting from any state where U holds, every computation of c eventually reaches a state where X holds.

The convergence condition implies that the correction predicate X is eventually truthified and is then continuously satisfied by the corrector program c, since X is closed in c. The *Progress* condition implies that c also eventually truthifies the witness predicate Z. Finally, the stability condition implies that Z once established remains true subsequently.

Remark. If the witness predicate Z is identical to the correction predicate X, our definition of the detects relation reduces to one given by Arora and Gouda [10]. 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.)

Properties. The corrects relation is antisymmetric and transitive in its first two arguments:

Lemma 4.0 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.

- If Z corrects X in d for U, and X corrects Z in d for U
  then $U \Rightarrow (Z \equiv X)$
- If Z corrects Y in d for U, and Y corrects X in d for U
  then Z corrects X in d for U □

Lemma 4.1 Let V be a state predicate such that $U \wedge V$ is closed in c. Then the following statements hold.

- If Z corrects X in c for U
  then Z corrects X in c for $U \wedge V$
- If Z corrects X in c for U and $V \Rightarrow X$
  then $Z \vee V$ corrects X in c for U □

Compositions. Analogous to detection predicates that cannot be witnessed atomically, there exist correction predicate X cannot be corrected atomically, i.e., by executing at most one step (action) of a corrector. To correct such predicates, we construct "large" correctors from "small" correctors just as we did for detectors.

Parallel composition of correctors. The parallel composition of two correctors $c1$ and $c2$, denoted by $c1 [\!] c2$, is the union of the (variables and actions of) programs $c1$ and $c2$.

Theorem 4.2 Given $Z1$ corrects $X1$ in $c1$ for U and $Z2$ corrects $X2$ in $c2$ for U.
If the variables of $c1$ and $c2$ are mutually exclusive
then $Z1 \wedge Z2$ corrects $X1 \wedge X2$ in $c1 [\!] c2$ for U □

The time taken by $c1 [\!] c2$ to correct $X1 \wedge X2$ is the maximum of the time taken to correct $X1$ and to correct $X2$. The space taken is the corresponding sum.

Sequential composition of correctors. The sequential composition of correctors $c1$ and $c2$, denoted by $c1; c2$, 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 $Z1 \wedge c2$.

Theorem 4.3 Given $Z1$ corrects $X1$ in $d1$ for U and $Z2$ corrects $X2$ in $d2$ for $(U \wedge X1)$.
If the variables of $c1$ and $c2$ are mutually exclusive, and $U \Rightarrow (Z2 \Rightarrow X1)$
then $Z2$ corrects $X1 \wedge X2$ in $c1; c2$ for U □

The time (respectively, space) taken by $c1; c2$ to correct $X1 \wedge X2$ is the sum of the time (respectively, space) taken to correct $X1$ and to correct $X2$.

As mentioned in the previous section, one way to design a corrector for X is by sequential composition of a detector and a corrector: the detector first witnesses whether $\neg X$ holds and, using this witness, the corrector then establishes X.

Theorem 4.4 Given Z detects $\neg X$ in d for U, $Z \Rightarrow Z'$, and X corrects X in c for $U \wedge Z'$.
If X is closed in d, and $\{U \wedge Z\}\ c\ \{Z \vee X\}$
then X corrects X in $(\neg Z \wedge d); c$ for U □

If c is atomic, i.e., c satisfies *Progress* and *Convergence* in at most one step, the following corollary holds.

Corollary 4.5 Given Z detects $\neg X$ in d for U, $Z \Rightarrow Z'$, and X corrects X in c for $U \wedge Z'$.
If X is closed in d, and c is atomic
then X corrects X in $d; c$ for U □

Another way to design a corrector for X is by sequential composition of a corrector and a detector: the corrector first satisfies its correction predicate X and, then the detector satisfies the desired witness predicate Z.

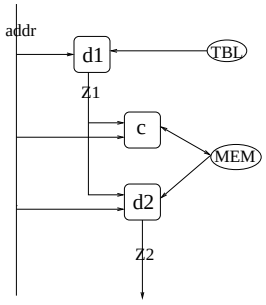
Theorem 4.6 Given X corrects X in c for U, and Z detects X in d for U.
If X is closed in d
then Z corrects X in $(\neg X \wedge c); d$ for U □

Again, if d is atomic, the following corollary holds.

Corollary 4.7 Given X corrects X in c for U, and Z detects X in d for U.
If X is closed in d and c is atomic
then Z corrects X in $c; d$ for U □

Example: Memory access (continued). If the given address is valid but is not in memory, an object of the form $\langle addr, - \rangle$ has to be added to the memory. (We omit the details of how this object is obtained, e.g., from a disk, a remote memory, or a network.) Thus, there is a corrector, c, which is formally specified as follows:

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



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

Figure 2: **Memory access**

Proposition 4.8

- $X2$ corrects $X2$ in c for $true$
- $X2$ corrects $X2$ in c for $U1$
- $X2$ corrects $X2$ in c for $U1 \wedge X1$

Before detector $d2$ can witness that the value of the address is in memory, corrector c should execute. Hence, we compose c and $d2$ sequentially. From Corollary 4.7 and Proposition 4.8, we have

Proposition 4.9

- $Z2$ corrects $X2$ in $c; d2$ for $U1$
- $Z2$ corrects $X2$ in $c; d2$ for $U1 \wedge X1$

Moreover, detector $d1$ should witness that the address is valid, before either corrector c or detector $d2$ execute. Hence, we compose $d1$ and $c; d2$ sequentially. Recall from Section 3 that $Z1$ detects $X1$ in $d1$ for $U1$. Also, $X1$ is closed in $d1$. Hence, if $d1$ is started in a state satisfying $U1 \wedge X1$, it will eventually satisfy $Z1$. It follows that $Z1$ corrects $X1$ in $d1$ for $U1 \wedge X1$. Therefore, from Theorem 4.3 and Proposition 4.8, we have

Proposition 4.10 $Z2$ corrects $X1 \wedge X2$ in $d1; (c; d2)$ for $U1 \wedge X1$

Application to design of fault-tolerance. Correctors suffice to ensure that execution of a program eventually resumes satisfying its safety and liveness properties. To see this, observe that if the correction predicate of a corrector is chosen to be the invariant of the program, the corrector ensures the program will eventually reach a state where the invariant holds and henceforth the program satisfies its safety and liveness properties.

To design nonmasking tolerance to F, we need to ensure that execution of p, after execution of actions in F, eventually reaches a state from where its specification is satisfied. Thus, correctors whose correction predicate is the invariant suffice for the design of nonmasking tolerance. Likewise, to design masking F-tolerance of p, we need to ensure that execution of p, in the presence of F, always satisfies its safety specification and, after execution of actions in F, resumes satisfying both its safety and liveness specification. For the the latter guarantee, it suffices that correctors be added to p (and for the former, it suffices that detectors be added to p, as discussed in the previous section).

5 Composition of Detector/Corrector Components and Programs

In this section, we discuss how a detector/corrector component is correctly added to a program so that the resulting program preserves 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, thereby preserving the specification of the latter.

Compositional proofs of interference-freedom have received substantial attention in the formal methods community [16–20] 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 (respectively a corrector) q, the safety specification of q, *viz Safeness* and *Stability*, and liveness specification, *viz Progress* and *Convergence*, are preserved.

Sufficient conditions for preserving 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 preserving 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 preserving *Progress* of q is to require that q be "atomic", i.e., that q satisfies 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 satisfied. 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 the 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 satisfied. 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 1. Sufficient conditions for the case of a corrector are similar.

Example: Memory access (continued). Consider an intolerant program p for memory access that assumes that the address is valid and is currently present in the memory. For ease of exposition, we let p perform only one memory access instead of repeated memory accesses. Thus, p is as follows:

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

For p to not interfere with the specification of the corrector $d1; (c; d2)$, it suffices that p execute only after $Z2$, the witness predicate of $d1; (c; d2)$, is satisfied. Hence, $d1; (c; d2)$ and p should be composed in sequence. From Theorem 5.3, we have that p does not interfere with $d1; (c; d2)$, i.e.

Proposition 5.6 $Z2$ corrects $X1 \wedge X2$ in $d1; (c; d2); p$ for $U1 \wedge X1$

In the following theorems, let Z detect X in q for U, and let U be closed in p.

Theorem 5.0 (Superposition)
If q does not read or write the variables of p, and p only reads the variables of q
then Z detects X in $q[]p$ for U

Theorem 5.1 (Containment)
If actions of p are a subset of q
then Z detects X in $q[]p$ for U

Theorem 5.2 (Atomicity)
If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$, and q is atomic
then Z detects X in $q[]p$ for U

Theorem 5.3 (Order of execution)
If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$
then Z detects X in $q;p$ for U

Theorem 5.4 (Termination)
If $\{U \wedge Z\}\ p\ \{Z \vee \neg X\}$, and U converges to V in $p[]q$
then Z detects X in $(\neg V \wedge p)[]q$ for U

Theorem 5.5 (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 $q[]p$ for U

Table 1 : Sufficient conditions for interference-freedom

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. Likewise, correctors such as reset, rollback recovery, and forward recovery procedures are typically restricted to execute only in states where the invariant does not hold. Thus, these correctors do not interfere with the program.

6 Designing Multitolerance

In this section, we first define multitolerance and then present our method for compositional, stepwise design of multitolerant programs.

Definition. Let p be a program with invariant S, $F1..Fn$ be n fault-classes, 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 S iff for each fault-class $Fj, 1 \leq j \leq n$, p is lj-tolerant to Fj for S.

The definition may be understood as follows: In the presence of faults from class Fj, p is perturbed only to states where some fault-span predicate Tj holds. Note that there exists a potentially different fault-span for each fault-class. After faults from Fj stop occurring, subsequent computation of p satisfies the problem specification as prescribed by the type of tolerance lj. For example, if lj is fail-safe, each computation of p starting from a state where Tj holds satisfies the safety specification.

Example: Memory access (continued). Observe that the memory access program, $d1; (c; d2); p$, discussed in Section 5, 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.

Compositional and stepwise design method. As outlined in the introduction, 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 that it is $l1$ tolerant to $F1$. The resulting program is then augmented with other detector/corrector components, in the second step, so that it is $l2$ tolerant to $F2$ and its $l1$ tolerance to $F1$ is preserved. And so on until, in the n-th step, the ln tolerance to Fn is added while preserving the $l1..ln-1$tolerances to $F1..Fn-1$. The multitolerant program designed thus has the structure shown in Figure 3.

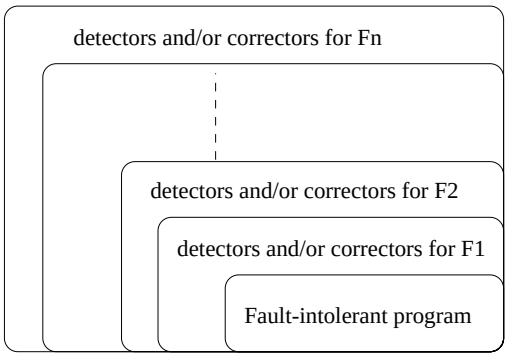


Figure 3: **Structure of a multitolerant program designed using our method**

First step. Let p be the intolerant program and S its invariant. By calculating the fault-span of p with respect to $F1$, detector and corrector components can be designed for satisfying $l1$ tolerance to $F1$. As discussed in Section 3 and 4, it suffices to add detectors to design fail-safe tolerance to $F1$, correctors to design nonmasking tolerance to $F1$, and both detectors and correctors to design masking tolerance to $F1$.

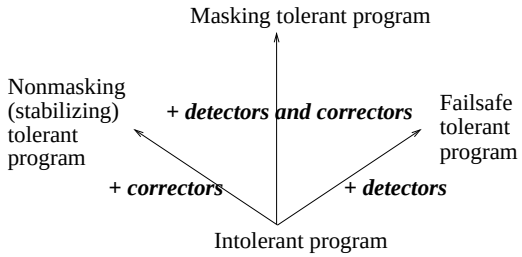


Figure 4: **Components that suffice for design of various tolerances**

Note that the detectors and correctors added to p are also be subject to F. Hence, they themselves have to be tolerant to F. But it is not necessary that they be masking tolerant to F. More specifically, it suffices that detectors added to design fail-safe tolerance be themselves fail-safe tolerant to F, since if the detectors fail to satisfy their liveness specification, the resulting program can be made to fail to satisfy the liveness specification of p. Likewise, it suffices that correctors added to design nonmasking tolerance be themselves nonmasking tolerant to F, since as long as the correctors eventually satisfy their safety and liveness specification, the resulting program can be made to eventually satisfy the safety and liveness specification of p. And, as can be expected, it suffices that the detectors and correctors added to design masking tolerance be themselves masking tolerant to F. (See Figure 5.)

In practice, the detectors and correctors often possess the desired tolerant to F trivially. But if they do not, one way to design them to be tolerant to F is by the analogous addition of more detectors and correctors. Another way is to design them to be self tolerant, without using any more detector and corrector components, as is exemplified by self-checking, self-stabilizing, and inherently fault-tolerant programs.

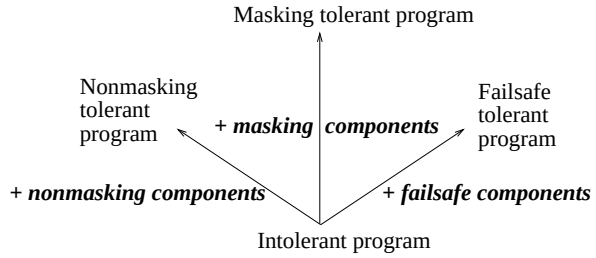


Figure 5: **Tolerance requirements for the components**

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, the invariant, and fault-span of $p1$ may differ from that of p. Therefore, letting $S1$ be the invariant of $p1$ and $T1$ be the fault-span of $p1$ with respect to $F1$, we show the following.

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

with the new components.

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

The addition of the detectors and correctors 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. (See Figure 6.) For reasons of space, we refer the interested reader to [21] for more details on this two-stage approach for designing masking tolerance.

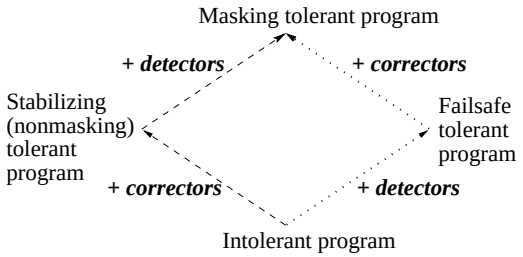


Figure 6: **Two approaches for stepwise design of masking tolerance**

Second step. This step adds $l2$ 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 $S2$ denote the invariant of the resulting program $p2$, $T21$ denote the fault-span of $p2$ in the presence of $F1$, and $T22$ denote the fault-span of $p2$ in the presence of $F2$, we show the following.

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

2. In the presence of $F2$, i.e., in states where $T22$ holds, $p1$ does not interfere with the new components, i.e., each computation of the new components satisfies 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 $T21$ holds, p does not interfere with the new components, i.e., each computation of the $p1$ satisfies the $l1$ fault-tolerance of $p1$ to $F1$ even if $p1$ execute 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.

7 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.

7.1 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):

$TR1 ::$	$j \neq 0 \;\wedge\; x.j \neq x.(j-1)$	$\longrightarrow$	$x.j := x.(j-1)$
$TR2 ::$	$j = 0 \;\wedge\; x.j \neq (x.N +_2 1)$	$\longrightarrow$	$x.j := x.N +_2 1$

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, the invariant of the program TR is

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

7.2 Adding Tolerance to 1 State Corruption

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

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 $\bot$ 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 $\bot$ 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 $x.(j-1) \neq \bot$ 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 $x.(j-1) \neq \bot$ holds. 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

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

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

$$x\text{-}corr :: \quad true \quad \longrightarrow \quad x.j := \bot$$

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

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

Consider the detector at j: Both its detection and witness predicates are $x.(j-1) \neq \bot$. Since the detects relation is trivially reflexive in its first two arguments, it follows that $x.(j-1) \neq \bot$ detects $x.(j-1) \neq \bot$ 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 5.1, 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} holds and a process state is corrupted every computation of PTR reaches a state where S_{TR} holds, 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.

7.3 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 satisfy $x.j \neq \perp$ 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 4.2, PTR reaches a state where $x.j$ and $x.k$ are not $\perp$.

For the second subcase (letting j be the predessor of k), j corrects $x.j$ from its predecessor $j-1$, satisfies $x.j \neq \perp$ and then terminates. Since the corrector at j does not read any variables written by the corrector at k, by Theorem 5.0, the corrector at j is not interfered by the corrector at k. After $x.j \neq \perp$ is satisfied, the corrector at k corrects $x.k$ from its predecessor j. By Theorem 4.4, 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.

7.4 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 when $x.j \neq \top$ holds.

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 $\bot$ nor $\top$. Action $FTR3$ is dN; it lets process N change $x.N$ from $\bot$ 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:

$FTR1 ::$	$x.(j-1) \neq \top$	$\wedge$	$PTR1$		
$FTR2 ::$	$x.N \neq \top$	$\wedge$	$PTR2$		
$FTR3 ::$	$x.N = \bot$			$\longrightarrow$	$x.N := \top$
$FTR4 ::$	$j \neq N \ \wedge \ x.j = \bot \ \wedge \ x.(j+1) = \top$			$\longrightarrow$	$x.j := \top$
$FTR5 ::$	$x.0 = \top$			$\longrightarrow$	$x.0 := 0$

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

$$S_{FTR} = X \in (\bigcup l, m, : 0 \leq l, m, l+m \leq N+1 : (\ (1 \cup \bot)^l \ (0 \cup \bot)^m (\bot \cup \top)^{(N+1-l-m)} \ \cup \ (0 \cup \bot)^l \ (1 \cup \bot)^m \ (\bot \cup \top)^{(N+1-l-m)}))$$

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 detected 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 3.3, $x.(N-1) = \top$ detects $(\forall j : N \geq j \geq (N-1) : x.j \neq 0 \wedge x.j \neq 1)$. $dN; d(N-1)$ 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 satisfy $x.0 = \top$, and execution of $c0$ will satisfy 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 5.2).

In program FTR, we have also added a detector at process j that detects $x.(j-1) \neq \top$. As described above (for 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 satisfied, i.e., no action of PTR is permanently blocked. Thus, the progress of these actions follows.

Theorem 7.0 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.

Remark. We would like to emphasize that the program FTR is masking tolerant to the fault-classes FK, with the invariant S_{FTR} and not S_{TR}. Thus, in the presence of faults in FK, S_{FTR} continues to be satisfied although S_{TR} may be violated. Process j, $j<N$, has a token iff $x.j$ differs from $x.(j+1)$ and neither $x.j$ nor $x.(j+1)$ is corrupted, and process N has a token iff $x.N$ is the same as $x.0$ and neither $x.N$ nor $x.0$ is corrupted. Thus, in a state where S_{FTR} is satisfied at most one process has a token. Also starting from such a state eventually the program reaches a state where S_{TR} is satisfied. Starting from such a state, each process can get then token. Thus, in any state in S_{FTR}, the specification of the token ring is satisfied.

8 Discussion

In this section, we address some of the issues that our method for design of multitolerance has raised. We also discuss the motivation for the design decisions made in this work.

Our formalization of the concept of multitolerance uses the abstractions of closure and convergence. Can other abstractions be used to formalize multitolerance? What are the advantages of using closure and convergence?

In principle, one can formulate the concept of multitolerance using abstractions other than closure and convergence. As pointed out by John Rushby [22], the approaches to formulate fault-tolerance can be classified into two: *specification* approaches and *calculational* approaches.

In specification approaches, a system is regarded as a composition of several subsystems, each with a standard specification and one or more failure specifications. A system is fault-tolerant if it satisfies its standard specification when all components do, and one of its failure specifications if some of its components depart from their standard specification. One example of this approach is due to Herlihy and Wing [23] who thus formulate graceful degradation, a special case of multitolerance.

In calculational approaches, the set of computations permissible in the presence of faults is calculated. A system is said to be fault-tolerant if this set satisfies the specification of the system (or a acceptably degraded version of it). Our approach is calculational since we compute the set of states reachable in the presence of faults (fault-span).

While other approaches may be used to formulate the design of multitolerance, we are not aware

of any formal methods for design of multitolerance using them. Moreover, in our experience, the structure imposed by abstractions of closure and convergence has proven to be beneficial in several ways: (1) it has enabled us to discover the role of detectors and correctors in the design of all tolerance properties (cf. Sections 3 and 4); (2) it has yielded simple theorems for composing tolerance actions and underlying actions in an interference-free manner (cf. Sections 5 and 6); (3) it has facilitated our design of novel and complex distributed programs whose tolerances exceed those of comparable programs designed otherwise [5, 10, 21, 24, 25, 26].

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. In an extended version of this paper [25], we present a case study of tree-based mutual exclusion which illustrates the analogous representation for permanent faults and for detectable or 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 auxiliary variables [5, 10]. For example, to represent Byzantine faults that affects a process j, we may introduce an auxiliary boolean variable $byz.j$ that is false iff j is Byzantine. If j is not Byzantine, it executes its "normal" actions. Otherwise, it executes some "abnormal" actions. When the Byzantine fault occurs, $byz.j$ is falsified, thus, permitting j to execute its abnormal actions. Similarly, to represent fail-stop and repair faults that affects a process j, we may introduce an auxiliary boolean variable $up.j$ that is false iff j has fail-stopped. All actions of j are restricted to be executed only when $up.j$ is true. When a fail-stop fault occurs, $up.j$ is falsified, thus preventing j from executing its actions. When a repair occurs, $up.j$ is truthified.

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 be trivially 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 which consists of three layers: the basic program that transmits the token, a corrector for the case when at least one process is not corrupted, and a

corrector 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. (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 possibly 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 impossibility to cosatisfy, it will follow that there do not exist components that can be added to the program while satisfying the interference-freedom requirements of our method.

How does our compositional method compare with the existing methods for designing fault-tolerant programs?

Our compositional method is rich in the sense that it accommodates various existing fault-tolerance design methods such as replication, checkpointing and recovery, Schneider's state machine approach, exception handling, and Randell's recovery blocks. (The interested reader is referred to [21, 25] for a detailed discussion of how properties such as replication, agreement, and order are designed by interference-free composition.)

How are fault-classes derived? Can our method be used if it is difficult to characterize the faults the system is subject to?

Derivation of fault-classes is application specific. It begins with the identification of the faults that the program may be subject to. Each of these faults is then formally characterized using state perturbations. (As mentioned above, auxiliary variables may be introduced in this formalization.) The desired type of tolerance for each fault is then specified. Finally, the faults are grouped into (possibly overlapping) fault-classes, based on the characteristics of the faults or their corresponding types of tolerance.

If it is difficult to characterize the faults in an application, a user of our method is obliged to guess some large enough fault-class that would accommodate all possible faults. It is often for this reason that designers choose weak models such as self-stabilization (where the state may be perturbed arbitrarily) or Byzantine failure (where the program may behave arbitrarily).

9 Concluding Remarks and Future Work

In this paper, we formalized the notion of multitolerance to abstract a variety of problems in dependability. Moreover, we presented a simple and stepwise method for designing multitolerant programs, that employed detector and corrector components for providing each desired type of tolerance.

The compositional nature of our method reduces the complexity of designing multitolerance. The theory for ensuring mutual interference-freedom in compositions of detectors and correctors with the intolerant program avoids re-proving the correctness of the program in every step. Furthermore, the addition of multiple components to an intolerant program is made tractable by adding tolerances to fault-classes one at a time.

To our knowledge, this is the first formal method for the design of multitolerant programs. Our method is effective for the design of quantitative as well as qualitative tolerances. As an example of quantitative tolerance, we presented a token ring protocol that recovers from upto K faults in $\Theta(K)$ time. For reasons of space, an interested reader is referred to [25] for a case study involving qualitative tolerance.

We note that we have used our method to design multitolerant programs for barrier computations, mutual exclusion, tree maintenance, leader election, bounded-space distributed reset, and termination detection [24, 25, 27]. To apply our design method in practice, we are currently developing SIEFAST, a simulation and implementation environment that enables stepwise implementation and validation of multitolerant distributed programs. We are also studying the mechanical synthesis of multitolerant concurrent programs.

Acknowledgments. We are grateful to the anonymous referees for their detailed and constructive comments on an earlier version of this paper.

References

[1] D. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation.* Digital Press, 1992.

[2] W.N. Toy and L.C. Toy. *The AT&T Case*, chapter 8. in [1].

[3] R.W. Kocsis. *The Galileo Case*, chapter 9. in [1].

[4] F. B. Bastani, I.-L. Yen, and I.-R. Chen. A class of inherently fault-tolerant distributed programs. *IEEE Transactions on Software Eng.*, 14(10):1431–1442, 1988.

[5] A. Arora. *A Foundation of Fault-Tolerant Computing.* PhD thesis, The University of Texas at Austin, 1992.

[6] A. Gopal and K. Perry. Unifying self-stabilization and fault-tolerance. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 195–206, 1993.

[7] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[8] M Gouda and M. Schneider. Maximal flow routing. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[9] I. Yen and F. Bastani. A highly safe self-stabilizing mutual exclusion algorithm. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[10] 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.

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

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

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

[14] B. Alpern and F. Schneider. Proving boolean combinations of deterministic properties. *Proceedings of the Second Symposium on Logic in Computer Scien ce*, pages 131–137, 1987.

[15] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*, chapter 9. Addison Wesley Publishing Company, 1989.

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

[17] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, pages 281–302, 1981.

[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] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[21] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. Submitted to *IEEE Transactions on Software Engineering*, a preliminary version of this paper appears in *Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 1995.*

[22] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System*, 43:180–219, 1994.

[23] M. Herlihy and J. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[24] S. S. Kulkarni and A. Arora. Multitolerance in distributed reset. Technical Report OSU-CISRC 02/96-TR13, Ohio State University, 1996. Submitted to *Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization.*

[25] A. Arora and S. S. Kulkarni. Multitolerance and its design. Technical Report OSU-CISRC 07/96 TR-37, Ohio State University, 1996.

[26] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):292–306, 1996.

[27] S. S. Kulkarni and A. Arora. Stepwise design of tolerances in barrier computations. Technical Report OSU-CISRC 03/96 TR-17, Ohio State University, 1996.

Appendix 1: Notation

Symbols	
$p, q, c, c1, d, d1$	program
$d, d1$	detector
$c, c1$	corrector
$F, F1$	faults
$R, S, T, U, V, X, X1, Z, Z1$	state predicate
$X, X1$	detection/correction predicate
$Z, Z1$	witness predicate

Compositions	
$guard \wedge action$	restriction of *action*
$guard \wedge program$	restriction of *program*
$p \llbracket q$	parallel
$p; q$	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