On Systematic Design of Fast and Perfect Detectors

Arshad Jhumka[*]
Dept. of Computer Engineering
Chalmers University of Technology
412 96, Göteborg, Sweden
Email: arshad@ce.chalmers.se

Felix C. Gärtner[†]
Laboratoire de Programmation Distribuée
Departement de Systémes de Communications
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
Email: fgaertner@lpdmail.epfl.ch

Christof Fetzer
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932, USA
Email: christof@research.att.com

Neeraj Suri
Department of Computer Science
Darmstadt University of Technology
64283 Darmstadt, Germany
Email: suri@informatik.tu-darmstadt.de

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
Technical Report 200263
September 22, 2002

Abstract

We present a theory of fast and perfect detector components that extends the theory of detectors and correctors of Arora and Kulkarni, and based on which, we develop an algorithm that automatically transforms a fault-intolerant program into a fail-safe fault-tolerant program. Apart from presenting novel insights into the working principles of detectors, the theory also allows the definition of a detection latency efficiency metric for a fail-safe fault-tolerant program. We prove that in contrast to an earlier algorithm by Kulkarni and Arora, our algorithm produces fail-safe fault-tolerant programs with optimal detection latency. The application area of our results is in the domain of distributed embedded applications.

Keywords: detectors, fail-safe fault-tolerance, program transformation, formal methods, safety specification.

Technical areas: Fault Tolerant and Dependable Systems, Software Engineering and Formal Methods, Distributed Algorithms.

[*] Work by Arshad Jhumka was supported in part by TFR grant and Saab Endowment.

[†] Work by Felix Gärtner was supported by an Emmy Noether scholarship from Deutsche Forschungsgemeinschaft.

1 Introduction

Safety-critical applications need to satisfy stringent dependability requirements in their provision of services. To reduce the complexity of designing such applications, Arora and Kulkarni [2] proposed a transformational approach, whereby an initially fault-intolerant program is systematically transformed into a fault-tolerant one. The main step involved in designing a fault-tolerant program is composing the corresponding fault-intolerant program with components that (i) detect and/or (ii) correct errors that arise as a result of faults, depending on the level of fault-tolerance to be achieved. The class of programs that achieves the first goal is termed *detectors* while the class of programs that achieves the second goal is called *correctors* [4].

In this paper we restrict our attention to designing *fail-safe* fault-tolerance. Intuitively this means that it is acceptable that the program "halts" if faults occur as long as it always remains in a "safe" state. This type of fault-tolerance is often used in (nuclear) power plants or train control systems where safety (avoidance of catastrophic events) is more important that continuous provision of service. In the context of the Arora/Kulkarni approach, fail-safe fault-tolerance can be achieved by merely employing detectors.

Detectors can be regarded as an abstraction of many different existing fault-tolerance mechanisms. For example, a common way to achieve fault-tolerance is to replicate a critical task and schedule it on different processors. The outputs of these tasks are brought together in a voter which outputs a consistent value. The voter contains a comparator which is an instance of a detector. Another (maybe more obvious) example of a detector is the use of error detecting codes. Other error handling mechanisms like acceptance tests or executable assertions can also be formulated as detectors in the sense of Arora and Kulkarni [4]. Hence, reasoning on the level of detectors makes an approach applicable to many different practical settings.

In an earlier work, Kulkarni and Arora [11] presented an algorithm which automates the addition of fail-safe fault tolerance to an initially fault-intolerant program. This algorithm is based on an analysis of the state transition representation of the program in the presence of faults. The algorithm is sound and complete meaning that (i) the transformed program is in fact a fail-safe fault-tolerant version of the original program, and (ii) if a fail-safe fault-tolerant version of the program exists, then the algorithm will find it. In this paper, we also present a sound and complete algorithm that automates addition of fail-safe fault-tolerance, but the resulting program is more *efficient* than that from the Kulkarni/Arora algorithm [11]. More efficient means that the fail-safe fault-tolerant program has a lower *detection latency* and hence detect faults *faster* (in this respect, we will also say that the program contains *fast detectors*). In fact, we show that our algorithm generates fail-safe fault-tolerant programs where the detection latency is *optimal.* A low detection latency means that the effects of faults are contained, i.e., errors do not propagate and contaminate the system state more than necessary.

Our algorithm is derived out of a refined theory of detectors. This theory develops a terminology which we believe captures and explains the working principles of detectors better than before. The basic building block of the theory is the notion of a transition which is *inconsistent* with respect to a safety [13] specification. This can be understood as follows: Executing a transition inconsistent w.r.t. the safety specification will lead to a violation of the safety specification if no countermeasures are taken. Building upon this concept, we develop a theory of accurate, complete, perfect and fast detectors together with the necessary correctness theorems. Intuitively, a detector is *accurate* if it "preserves" correct behaviors of the system in the presence of faults. A detector is *complete* if it "rejects" incorrect behaviors. A detector is *perfect* if it is accurate and complete. Finally, a set of detectors is *fast* if one of the detectors "halts" the program as soon as possible after the occurrence of faults.

Overall, in this paper, we make the following contributions:

- We present a novel theory of fast and perfect detectors which improves on the understanding of the basic working principles of detectors in the context of the Arora/Kulkarni theory [4].
- Based on this theory, we provide an algorithm that systematically transforms a fault-intolerant program into a fail-safe fault-tolerant program with optimal detection latency.

Some of the notions which underlie the theory of perfect detectors have been studied by Gärtner and Völzer [7]. However, we are not aware of any other work which employs these concepts in an automatic transformation.

The paper is structured as follows: Section 2 recalls the basic system model of the Arora/Kulkarni theory. Section 3 provides an overview of detectors and their role in establishing fail-safe fault tolerance. Section 4 defines the problem of adding fail-safe fault-tolerance using detectors. Section 5 develops the theory of perfect and fast detectors. In Section 6 we present the algorithm that automatically generates a fail-safe fault-tolerant program from the corresponding fault-intolerant program with perfect and fast error detection capabilities. We conclude the paper in Section 7.

2 Preliminaries

In this section, we recall the standard formal definitions of programs, faults, fault tolerance (in particular, fail-safe fault-tolerance), and of specifications [4].

2.1 Programs

A *program p* consists of a set of variables V_p and a finite set of actions. Each variable stores a value of a predefined nonempty domain and is associated with a predefined set of initial values. An action has the form

$$\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

in which the guard is a boolean expression over the program variables and the statement is either the empty statement or an instantaneous assignment to one or more variables.

We define the *state space S_p of a program p* as the set of all possible assignments of values to variables. A *state predicate of p* is a boolean expression over the state space of p. The *set of initial states I_p* is defined by the set of all possible assignments of initial values to variables.

An *action ac of p is enabled in a state s* if the guard of ac evaluates to "true" in s. An action ac is represent by a set of state pairs. We assume that actions are deterministic, i.e., $\forall s, s', s'' : (s, s') \in ac \wedge (s, s'') \in ac \Rightarrow s' = s''$. Note that programs are permitted to be non-deterministic since multiple actions can be enabled in the same state. In particular, each non-deterministic action can be converted into a set of deterministic actions with an identical state transition relation.

A *computation of p* is a weakly fair (finite or infinite) sequence of states $s_0, s_1, \ldots$ such that $s_0 \in I_p$ and for each $j \geq 0$, s_{j+1} results from s_j by executing the assignment of a single action which is enabled in s_j. We require that *weak fairness* implies that if a program action ac is continuously enabled, ac is eventually chosen to be executed. Weak fairness implies that a computation is *maximal* with respect to program actions, i.e., if the computation is finite then no program action is enabled in the final state.

If α is a finite computation and β is a computation, we denote with $\alpha \cdot \beta$ the *concatenation* of both computations. A *state s occurs in a computation* $s_0, s_1, \ldots$ iff there exists an i such that $s = s_i$. Similarly, a *transition* (s, s') *occurs in a computation* $s_0, s_1, \ldots$ iff there exists an i such that $s = s_i$ and $s' = s_{i+1}$.

In the context of this paper, programs are equivalently represented as state machines, i.e., a program is a tuple $p = (S_p, I_p, \delta_p)$ where S_p is the state space and $I_p \subseteq S_p$ is the set of initial states. The state transition relation $\delta_p \subseteq S_p \times S_p$ is defined by the set of actions as follows: Every action ac implicitly defines a set of transitions which is added to δ_p. Transition $(s, s') \in \delta_p$ iff ac is enabled in state s and computation of the statement results in state s'. We say that *ac induces* these transitions. State s is called the *start state* and s' is called the *end state* of the transition.

2.2 Specifications

A *specification* for a program p is a set of computations which is fusion-closed. A *specification S is fusion-closed* iff the following holds for finite computations α, γ, a state s and computations β, δ: If $\alpha \cdot s \cdot \beta$ and $\gamma \cdot s \cdot \delta$ are in S, then so are $\alpha \cdot s \cdot \delta$ and $\gamma \cdot s \cdot \beta$. We will discuss the consequences of demanding fusion-closed specifications in Section 3.

A *computation c_p of p satisfies a specification S* iff $c_p \in S$, otherwise c_p violates S. A *program p satisfies a specification S* iff all possible computations of p satisfy S.

Definition 1 (Maintains) *Let p be a program, S be a specification and α be a finite computation of p. We say that α* maintains *S iff there exists a sequence of states β such that $\alpha \cdot \beta \in S$.*

Definition 2 (Safety specification) *A specification S of a program p is a* safety specification *iff the following condition holds: For every computation σ that violates S, there exists a prefix α of σ such that for all computations β, $\alpha \cdot \beta$ violates S.*

Proposition 1 *A specification S is a safety specification iff for all $\sigma \notin S$ there exists a prefix α of σ such that α does not maintain S.*

Proof. Follows from the Definitions 1 and 2. □

Informally, the safety specification of a program states that "something bad never happens". More formally, it defines a set of "bad" finite computation prefixes that should not be found in any computation. Alpern and Schneider [1] have shown that every specification can be written as the intersection of a safety specification and a *liveness specification*. Informally, a liveness specification determines what types of events must eventually happen. Since we are mainly interested here in safety specifications we omit the formal definition of liveness. However, liveness issues are important since any safety specification can be satisfied by the empty program, i.e., the program that does nothing.

Multiple initial states reflect the fact that a program may initially read external inputs before executing. We additionally assume a set of special variables called the *output variables* in which the program should finally write the results of a computation. This model is suitable for the domain of embedded applications (like sensors and actuators) which we aim at in this paper. Program actions that write output variables are referred to as *critical actions*.

2.3 Fault Models and Fault Tolerance

All standard fault models from practice which endanger a safety specification (transient or permanent faults) can be modeled as a set of added transitions. We focus on the subset of these fault models which can potentially be tolerated: We disallow faults to violate the safety specification directly. For example, if a safety specification constrains the output variables of a program, the fault model prevents the *fault actions* of F to modify the output variables in such way that the fault itself results in a safety violation. However, fault actions can change the program state such that subsequent program actions violate the safety specification.

Definition 3 (Fault model) *A* fault model F for program p and safety specification SS *is a set of actions over the variables of p that do not violate the SS, i.e., if transition* (s_j, s_{j+1}) *is in F and* $s_0, s_1, \ldots, s_j$ *is in SS, then* $s_0, s_1, \ldots, s_j, s_{j+1}$ *is in SS.*

Definition 4 (Computation in the presence of faults) *A* computation of p in the presence of F *is a weakly p-fair sequence of states* $s_0, s_1, \ldots$ *such that* s_0 *is an initial state of p and for each* $j \geq 0$*,* s_{j+1} *results from* s_j *by executing a program action from p or a fault action from F and there exists no program action ac such that ac is permanently enabled but never executed, i.e.,* $\forall j : (\forall k \geq j, \exists s' : (s_j, s') \in ac) \Rightarrow (\exists k \geq j : (s_k, s_{k+1}) \in ac)$.

Weakly p-fair means that only the actions of p are treated weakly fair (fault actions must not eventually occur if they are continuously enabled). We say that a *fault occurs* if a fault action is executed.

Rephrased in the transition system view, a fault model adds a set of transitions to the transition relation of p. We denote the modified transition relation by δ_p^F. Since fault actions are not treated fairly, their occurrence is not mandatory. Note that we do not rule out faults that occur infinitely often (as long as they do not directly violate the safety property).

Definition 5 (Fail-safe fault-tolerance) *Let S be a specification and SS be the smallest safety specification including S. A program p is said to be* fail-safe F-tolerant for specification S *iff all computations of p in the presence of faults satisfy SS.*

If F is a fault model and SS is a safety specification, we say that a *program p is F-intolerant for SS* iff p satisfies SS in the absence of faults F but violates SS in the presence of faults F. For brevity, we will write *fault-intolerant* instead of *F-intolerant for SS* if F and SS are clear from the context.

3 Detectors and their Role in Constructing Fail-Safe Fault-Tolerant Programs

We adopt the view of Arora and Kulkarni [4] that a fault-tolerant program is the composition of a fault-intolerant program with fault-tolerance components. Using the same system model as in this paper, Arora and Kulkarni proved that a class of program components called *detectors* are necessary and sufficient to establish fail-safe fault-tolerance. Intuitively, a detector is a program module that detects whether a given state predicate is satisfied in a given state. Instances of detectors can be executable assertions [9], error detection codes, or comparators. In this paper we are mainly interested in adding fail-safe fault-tolerance, thus we review the result stating that detectors are *sufficient* to build fail-safe fault-tolerant applications. The main idea of the result is to use detectors to simply “halt” the program in a state where it is about to violate the safety specification.

An important prerequisite for the Arora/Kulkarni sufficiency result is that specifications are fusion-closed. Fusion-closed specifications allow to characterize a safety specification as a set of disallowed “bad” *transitions* (instead of a set of disallowed computation prefixes).

Proposition 2 *Let SS be a safety specification, p an F-intolerant program for SS. If p violates SS then there exists a transition* $t \in \delta_p$ *such that for all computations* σ *of p holds: If t occurs in* σ *then* $\sigma \notin SS$*.*

Proof. Since p violates SS, there exists a computation σ which is not in SS. The fact that SS is a safety property implies that σ contains a minimal prefix, written $\alpha \cdot s \cdot s'$, which does not maintain SS (i.e., which prevents the computation from being in SS). This prefix has at least length 2 because all initial states of p maintain SS. We must now show that if (s, s') occurs in any other computation ρ of p, then $\rho \notin SS$:

1. For a contradiction, assume $\rho = \hat{\alpha} \cdot s \cdot s' \cdot \hat{\beta} \in SS$. We will show that $\alpha \cdot s \cdot s'$ maintains SS.
2. Since SS is a safety property and $\rho \in SS$ (step 1), all prefixes of ρ maintain SS.
3. From step 2 and because it is a prefix of ρ, computation $\hat{\alpha} \cdot s \cdot s'$ maintains SS.
4. From step 3 and definition of maintains: $\exists \hat{\delta} : \hat{\alpha} \cdot s \cdot s' \cdot \hat{\delta} \in SS$.
5. From assumption $\alpha \cdot s$ maintains SS, so from definition of maintains we have: $\exists \delta : \alpha \cdot s \cdot \delta \in SS$.
6. Because of fusion-closure of SS and the steps 4 and 5 construct: $\alpha \cdot s \cdot s' \cdot \hat{\delta} \in SS$.
7. Step 6 means that $\alpha \cdot s \cdot s'$ maintains SS, which is a contradiction to the fact that $\alpha \cdot s \cdot s'$ does not maintain SS.

□

We call the transitions identified in Proposition 2 *bad transitions.* Intuitively, to maintain a safety specification now requires to keep track of the current computation and take precautions not to run into one of the bad transitions which are disallowed by the safety specification. From our restrictions of the fault model we know that bad transitions must be program transitions. A detector refines the guard of the corresponding action in such a way that the action is never executed whenever the computation could result in taking a bad transition. Formally, a detector for an action implements a state predicate d which is "true" iff execution of the action starting in d maintains the specification. In the programming notation, given an action $g \rightarrow st$, a detector for this action refines the guard to $g \wedge d$. Arora and Kulkarni formulate this fact in their original work as follows [2, Theorem 4.3]:

Theorem 1 (Sufficiency of detectors) *For each action ac of p there exists a predicate d such that execution of ac in a state where d holds maintains SS.*

Definition 6 (Detector for an action) *Let SS be a safety specification. An SS-*detector *d* monitoring program action *ac* of *p is a state predicate of p which is guaranteed to exist according to Theorem 1.*

We will simply talk about *detectors* instead of *SS-detectors* if the relevant safety specification is clear from the context.

Consider the transition system view of a program p again. We say that a *state s is reachable by p* iff starting from an initial state of p it is possible to construct a computation which contains s using only transitions from δ_p. Otherwise s is unreachable. A transition of p is reachable iff its start state is reachable. Otherwise it is unreachable. Similarly the notions of a state or transition being *reachable in the presence of faults* can be defined by referring to δ_p^F.

Using the above terminology, detectors *remove* program transitions which were unreachable by p but become reachable in the presence of faults. In a sense, composing a program with detectors means to refine the original transition relation and eliminate certain program transitions to make bad transitions unreachable.

Designing detectors is not an easy task in distributed systems and has its inherent complexities [12, 10]. Kulkarni and Arora [11] have presented an algorithm to automatically add detectors to a given program in the transition system representation. Basically, this algorithm simply refines the guards of critical actions to maintain the safety specification in the presence of faults. We will revisit and improve this method later in Section 5.

We close this section with a final remark regarding the assumption that specifications be fusion-closed. Informally spoken, fusion-closure guarantees that the entire history of a computation "is available" in the current state of the system, i.e., it is sufficient to observe the current system state to know whether the next step will result in a disallowed prefix. It has been observed [8, 2] that specifications in the popular Unity logic [6] are fusion-closed, as are low-level specifications like C programs or transition systems.

In general a specification that is not fusion-closed can be converted into a fusion-closed specification through the addition of history variables. How this can be done in a way that minimizes the number of additional states remains a topic for further research.

4 The Transformation Problem

We now formally state the problem of transforming a fault-intolerant program p into a fail-safe fault-tolerant version p' for a given safety specification SS and fault model F [11].

When deriving p' from p, only fault tolerance should be added, i.e., p' should not satisfy SS in new ways in the absence of faults. Specifically, there are two conditions to be satisfied in the transformation problem:

- If there exists a transition (s, t) in p' that is not used by p to satisfy SS, then (s, t) cannot be used by p', since this means that there are other ways p' can satisfy SS in the absence of faults. Thus, the set of transitions of p' should be a subset of the set of transitions of p.

- Also, if there exists a state s reachable by p' in the absence of faults that is not reached by p in the absence of faults, then this means that p' can satisfy SS differently from p in the absence of faults, and such a state s should not be reached by p' in the absence of faults. Thus, the set of states reachable by p' should be a subset of the set of states reachable by p.

In general, these conditions result in the requirement that both programs should have the same set of fault-free computations. Formally, we define the transformation problem as follows:

Definition 7 (Transformation for fail-safe fault tolerance) *Let SS be a safety specification, F a fault model F, and p an F-intolerant program for SS. Identify a program p' such that the following three conditions hold:*

1. *p' satisfies SS in presence of F.*
2. *In the absence of faults, every computation of p' is a computation of p.*
3. *In the absence of faults, every computation of p is a computation of p'.*

Later in Section 6 we present an algorithm which solves the above transformation problem, i.e., we present an algorithm that systematically transforms a fault-intolerant program into a program that satisfies the above three conditions. The algorithm is based on a theory for fast and perfect detectors which we introduce in the following section.

5 A Theory of Fast and Perfect Detectors

This section, which can be regarded as the heart of the paper, presents a theory of detector components which helps in the design of fail-safe applications. The theory is centered around the notion of an SS-inconsistent transition which is introduced in Section 5.1. Using this notion, we identify correctness criteria for programs composed with so-called *perfect* detectors in Section 5.2. Finally, we formulate a metric of *detection latency* for the efficiency of detectors in Section 5.3 to compare different transformation algorithms. Our algorithm to add fail-safe fault tolerance presented in Section 6 directly follows from the theory presented now.

5.1 Transition Consistency in the Context of Safety Specifications

The intuition behind the definition of inconsistency is that if a given computation violates the safety specification, then some “errorneous” transition occurred in the computation.

Definition 8 (*SS*-inconsistent transitions) *Given a fault-intolerant program p with safety specification SS, and a computation α. A transition (s, s') is* SS-inconsistent for p w.r.t. α *iff*

- *α is a finite computation of p in the presence of faults,*
- *(s, s') occurs in α, i.e., $\alpha = \sigma \cdot s \cdot s' \cdot \beta$,*
- *all transitions in $s \cdot s' \cdot \beta$ are in δ_p,*
- *α violates SS, and*
- *$\sigma \cdot s$ maintains SS.*

```
Program $P1$
    var $x$ init 1, $y$ init 1, $z$ init 10, $c$ init 1 : int

    $c = 1 \rightarrow x :=$ read(); $c := c + 1$; // value between 5 and 10
    $c = 2 \rightarrow y :=$ read(); $c := c + 1$; // value between 5 and 15
    $c = 3 \rightarrow z := x + y$; $c := c + 1$
    $c = 4 \rightarrow$ output($z$); $c := 1$ // loop forever

$F$ (faults):
    true $\rightarrow x :=$ random $[0 \ldots 25]$
    true $\rightarrow y :=$ random $[0 \ldots 50]$
```

Figure 1: Program to illustrate Definition 8.

We now illustrate this definition: Consider the program $P1$ in Figure 1 which reads two sensors, and then outputs the sum of the two readings. The safety specification SS requires the output to be always between 10 and 25. The fault transitions state that, from each state, the value of variable x (respectively, y) can be arbitrarily changed to a value in the range of $[0 \ldots 25]$(respectively, $[0 \ldots 50]$). Consider now computation α (states are given as triples $\langle x, y, z \rangle$, i.e., the program counter c is not explicitly given):

$$\alpha : \langle 1, 1, 10 \rangle \cdot \langle 10, 1, 10 \rangle \cdot \langle 10, 5, 10 \rangle \cdot \langle 10, 5, 15 \rangle$$

Obviously, α satisfies SS and so no program transition is SS-inconsistent. Now consider computation β which violates SS:

$$\beta : \langle 1, 1, 10 \rangle \cdot \langle 10, 1, 10 \rangle \cdot \langle 25, 1, 10 \rangle \cdot \langle 25, 5, 10 \rangle \cdot \langle 25, 5, 30 \rangle$$

In β, a fault transition occurs after the second state, i.e., state $\langle 10, 1, 10 \rangle$, changing the value of x to 25. The subsequent program transition from $\langle 25, 1, 10 \rangle$ to $\langle 25, 5, 10 \rangle$ is SS-inconsistent, since the execution of the following program transition to state $\langle 25, 5, 30 \rangle$ causes a violation of the safety specification. The program transition from $\langle 25, 5, 10 \rangle$ to $\langle 25, 5, 30 \rangle$ is also SS-inconsistent. The first program transition and the fault transition are not SS-inconsistent.

Intuitively, an SS-inconsistent transition for a given program computation is a program transition where the subsequent execution of a sequence of program actions causes the computation to violate the safety specification. In a sense, SS-inconsistent transitions lead the program computation on the "wrong path".

Now we define SS-inconsistency independent of a particular computation.

Definition 9 (*SS*-inconsistent transition for p) *Given a program p with safety specification SS. A transition (s, s') is* SS-inconsistent for p *iff there exists a computation α of p in the presence of faults such that (s, s') is SS-inconsistent for p w.r.t. α.*

In general, a transition can be SS-inconsistent w.r.t. a computation α_1, and not be SS-inconsistent w.r.t. α_2. This can be due to nondeterminism in program execution. To see this consider the program $P2$ in Figure 2. The safety specification SS mandates that always $10 \leq d \leq 50$. Consider now the following computation α_1 of $P2$ (a state is given as $\langle w, x, y, z \rangle$):

$$\alpha_1 = \langle 1, 5, 1, 10 \rangle \cdot \langle 1, 10, 1, 10 \rangle \cdot \langle 1, 45, 1, 10 \rangle \cdot \langle 15, 45, 1, 10 \rangle \cdot \langle 15, 45, 15, 10 \rangle \cdot \langle 15, 45, 15, 60 \rangle$$

In the second state a fault occurs setting x to 45 and effectively causing α_1 to violate SS after execution of a sequence of program transitions. Notice that the transition $t = (\langle 1, 45, 1, 10 \rangle, \langle 15, 45, 1, 10 \rangle)$ is SS-inconsistent for p w.r.t. α_1.

Now consider computation α_2 of p:

$$\alpha_2 = \langle 1, 5, 1, 10 \rangle \cdot \langle 1, 10, 1, 10 \rangle \cdot \langle 1, 45, 1, 10 \rangle \cdot \langle 15, 45, 1, 10 \rangle \cdot \langle 0, 45, 1, 10 \rangle \cdot \langle 0, 45, 0, 10 \rangle \cdot \langle 0, 45, 0, 45 \rangle$$

Here again a fault happens in the second state but due to a lucky interleaving of program actions α_2 maintains SS. Hence, the same program transition t as above is not SS-inconsistent for p w.r.t. α_2.

```
Program P2
    var w init 1, c1 init 1 : int // process a
    var x init 5, y init 1, z init 10, c2 init 1 : int // process b

process a:
    $c1 = 1 \rightarrow w :=$ read(); $c1 := c1 + 1$; // value between 15 and 25
    $c1 = 2 \wedge x \leq 15 \rightarrow w := w + 5$; $c1 := 1$; // loop
    $c1 = 2 \wedge x > 15 \rightarrow w := w - 15$; $c1 := 1$; // loop

process b:
    $c2 = 1 \rightarrow x :=$ read(); $c2 := c2 + 1$; // value between 0 and 20
    $c2 = 2 \rightarrow y := w$; $c2 := c2 + 1$;
    $c2 = 3 \rightarrow z := y + x$; $c2 := c2 + 1$;
    $c2 = 4 \rightarrow$ output($z$); $c2 := 1$; // loop

F (faults):
    true $\rightarrow x :=$ random $[10 \ldots 45]$
    true $\rightarrow w :=$ random $[1 \ldots 50]$
```

Figure 2: Program containing two concurrent processes with a transition that is both SS-inconsistent and not SS-inconsistent w.r.t. two different computations.

If we cannot find a computation in the presence of faults for which a particular transition is SS-inconsistent then we say that this transition is SS-consistent.

Definition 10 (*SS*-consistent transition for p) *Given a program p with safety specification SS. A* transition (s, s') is SS-consistent for p *iff* (s, s') *is not SS-inconsistent for p.*

The notion of SS-inconsistency is a characteristic for a computation which violates SS.

Proposition 3 *Given an fault-intolerant program p for a safety specification SS. Every computation α of p in the presence of faults that violates SS contains an SS-inconsistent transition for p w.r.t. α.*

Proof.

1. Because p is F-intolerant, there exists a computation α of p in the presence of faults such that $\alpha \notin SS$.
2. From step 1 and Proposition 2 there exists a bad transition (s, s') in α.
3. From step 2 and the restriction of F follows that $(s, s') \in \delta_p$.
4. From step 3 and Definition 8, (s, s') is SS-inconsistent for p w.r.t. α.

□

Inconsistent transitions can also be characterized through the reachability of bad transitions.

Proposition 4 *Given a fault-intolerant program p for a safety specification SS. If (s, s') is an SS-inconsistent transition for p then a bad transition is reachable starting from s using only program transitions from δ_p.*

Proof. The proof follows directly from the definition of SS-inconsistent transitions and Proposition 2. □

Reachability of bad transitions in δ_p leads to the following observation.

Proposition 5 *Given a fault-intolerant program p for safety specification SS. Every SS-inconsistent transition for p is not reachable in the absence of faults.*

Proof.

1. For a contradiction, assume the start state s of an SS-inconsistent transition (s, s') is reachable in the absence of faults.
2. Step 1 implies that there exists a computation $\alpha \cdot s \cdot s'$ of p in the absence of faults.
3. From the fact that (s, s') is inconsistent, and Proposition 4 there exists a computation $s \cdot s' \cdot \beta$ of p in the absence of faults in which a bad transition occurs.
4. From steps 2 and 3 follows that there exists a computation $\sigma = \alpha \cdot s \cdot s' \cdot \beta$ of p in the absence of faults containing a bad transition.
5. From step 4 and Proposition 2 there exists a computation of p in the absence of faults which violates SS.
6. From step 5 p violates SS in the absence of faults, a contradiction.

□

Note that the previous observation cannot be strengthened to an equivalence (a non-reachable transition in the absence of faults must not be SS-inconsistent). But it can be reformulated to characterize reachable transitions in the absence of faults as SS-consistent.

Corollary 1 *Given a fault-intolerant program p for a safety specification SS. Every reachable transition $(s, s') \in \delta_p$ in the absence of faults is SS-consistent for p*

In the next section, we introduce the notion of perfect detectors using the terminology of SS-consistency.

5.2 Perfect Detectors

From the previous section, we observed that SS-inconsistent transitions are those transitions that can lead a program to violate its safety specification in the presence of faults if no precautions are taken. Perfect detectors are a means to implement these precautions. The definition of perfect detectors follows two guidelines: A detector d monitoring a given action ac of program p needs to (1) "reject" the starting states of all transitions induced by ac that are SS-inconsistent for p, and (2) "keep" the starting states of all induced transitions that are SS-consistent for p. These two properties are captured in the definition of *accuracy* and *completeness* of detectors (the notions are defined in analogy to Chandra and Toueg [5]).

Definition 11 (Detector accuracy) *Given a program p with safety specification SS, and a program action ac of p. A* detector d monitoring ac is SS-accurate for ac in p *iff for all transitions* (s, s') *induced by ac holds: if* (s, s') *is SS-consistent for p, then* $s \in d$.

Definition 12 (Detector completeness) *Given a program p with safety specification SS, and a program action ac of p. A* detector d monitoring action ac is SS-complete for ac in p *iff for all transitions* (s, s') *induced by ac holds: if* (s, s') *is SS-inconsistent for p, then* $s \notin d$.

Definition 13 (Perfect detector) *Given a program p with safety specification SS, and a program action ac of p. A* detector d monitoring ac is SS-perfect for ac in p *iff d is both SS-complete and SS-accurate for ac in p.*

Where the specification is clear from the context we will write *accuracy* instead of *SS-accuracy* (the same holds for completeness and perfection).

Intuitively, the completeness property of a detector is related to the safety property of the program p in the sense that the detector should filter out all SS-inconsistent transitions for p, whereas the accuracy property relates to the liveness specification of p in the sense that the detector should not rule out SS-consistent transitions. This intuition is captured by the following lemmas. The first one (Lemma 1) uses the accuracy property to show that the fault free behavior of a program is not affected by adding perfect detectors. The next one (Lemma 2) uses the completeness property to show that perfect detectors indeed establish fault-tolerance.

Lemma 1 (Perfect detectors and fault-free behavior) *Given a fault-intolerant program p and a set D of perfect detectors. Consider program p′ resulting from the composition of p and D. Then the following statements hold:*

1. *In the absence of faults, every computation of p′ is a computation of p.*
2. *In the absence of faults, every computation of p is a computation of p′ .*

Proof.

1. From Corollary 1, every program transition which is reachable in p is SS-consistent.
2. From construction, p' results from adding perfect detectors to p. Because they are perfect (Definition 13), they are accurate.
3. From steps 1, 2 and the definition of accuracy, all SS-consistent transitions of p are also transitions of p'.
4. Steps 1 and 3 imply that every reachable transition in p is also reachable in p'.

5. Step 4 implies that every computation of p is also a computation of p', proving the first claim of the lemma.

6. From the definition of a detector (Definition 6) follows that composition with detectors does not introduce new state transitions.

7. Step 6 implies that $\delta_{p'} \subseteq \delta_p$.

8. Step 7 implies that every computation of p' is also a computation of p, proving the second claim of the lemma.

□

Lemma 2 (Perfect detectors and behavior in the presence of faults) *Given a fault-intolerant program p for a safety specification SS. Given also a program p′ by composing the critical actions of p with perfect detectors. Then, p′ satisfies SS in presence of faults.*

Proof.

1. For a contradiction assume that p' violates SS. From definition of violates follows that there exists a computation σ of p' which is not in SS.

2. Step 1 and Proposition 2 imply that there a bad transition (s, s') occurs in σ.

3. Because of the restrictions on the fault model (critical variables are not affected), the transition (s, s') from step 2 must be a program transition (i.e., $(s, s') \in \delta_{p'}$) induced by some critical action ac.

4. From Definition 8 and step 3 the transition (s, s') is SS-inconsistent.

5. Consider the critical program action ac (from step 3) causing the bad transition. From construction of p', ac is composed with a perfect detector d.

6. From step 5 and because d is perfect, it is also complete.

7. Because d is complete (step 6), d monitors ac (step 5) and transition (s, s') induced by ac is SS-inconsistent (step 4), the definition of completeness implies that $s \notin d$.

8. Step 7 implies that $(s, s') \notin \delta_{p'}$ which contradicts step 3.

□

5.3 Fast Detectors

Perfect detectors ensure *correctness* of the transformation. We now turn to a different aspect, namely the efficiency of perfect detectors. Intuitively, we would like a fault to be detected as early as possible to prevent further contamination of the system state. In this section, we focus on explaining the relationship between *fast* detection and SS-inconsistent transitions.

Informally, a detector is perfect if it removes an arbitrary SS-inconsistent transition for every violating execution. A fast detector should remove the "first" SS-inconsistent transition.

Definition 14 (Earliest SS-inconsistent transition) *Given an F-intolerant program p with safety specification SS, and a computation $\alpha = s_0 \cdot s_1 \cdots s_i \cdot s_{i+1} \cdots s_m$ of p in the presence of faults that violates SS. The* transition (s_i, s_{i+1}) is the earliest SS-inconsistent transition for p w.r.t. α *iff the following two properties hold:*

1. *(s_i, s_{i+1}) is SS-inconsistent for p w.r.t. α.*
2. *(s_{i-1}, s_i) is a transition induced by a fault action.*

Intuitively, when a computation α of a program p in the presence of faults violates the safety specification SS of p, there exists a suffix that contains a maximal partial computation which starts with an SS-inconsistent transition and ends in a bad transition. The earliest SS-inconsistent transition is the first SS-inconsistent transition in this suffix.

Define the set $EIT_p^F(SS)$ of earliest SS-inconsistent transitions of a program p as the union of the earliest SS-inconsistent transitions over all computations of p violating SS. Define $p \setminus EIT_p^F(SS)$ as the program p' which is the same as p except that all transitions from $EIT_p^F(SS)$ have been removed from δ_p.

Definition 15 (Fast detectors) *Let p be a fault-intolerant program. A* set of perfect detectors D for program p is fast *iff p composed with D results in $p \setminus EIT_p^F(SS)$.*

Lemma 3 (Fast perfect detectors and behavior in the presence of faults) *Let p be a fault-intolerant program for a safety specification SS. Then p composed with a set of fast perfect detectors satisfies SS in the presence of faults.*

Proof. This is a generalization of the proof of Lemma 2.

1. For a contradiction, it is again assumed that p' violates SS, i.e., that there exists a computation σ of p' which is not in SS.

2. From Definition 14 it is possible to generalize Proposition 3 to state that in every violating execution there exists an earliest SS-inconsistent transition in every violating computation. Denote this transition in σ as (s, s').

3. The fact that detectors are fast and from Definintion 15 we have that all earliest inconsistent transitions are removed from p while constructing p', which is a contradiction to the occurrence of (s, s') in a computation of p'.

□

We now define a metric to measure the "fastness" of detectors. Intuitively, the detection latency defines the number of program transitions executed until the program "halts" at a detector, after a "harmful" fault has occurred.

Definition 16 (Detection latency) *Let SS be a safety specification and p' be a program which has been made fault-tolerant for SS by composing a fault-intolerant program p with a set of detectors. Consider a finite computation $\alpha = s_0 \cdots s_i \cdot s_{i+1} \cdots s_m$ of p' in the presence of faults, such that:*

1. *(s_{i-1}, s_i) is a transition induced by a fault action,*
2. *all transitions in $s_i \ldots s_m$ are in $\delta_{p'}$, and*
3. *starting from s_m a bad transition in SS is reachable by using only one program action of p.*

Then, the detection latency $L_p(\alpha)$ of p' w.r.t. α *is the number of transitions executed in* $s_i \dots s_m$.

Definition 17 (Maximum detection latency) *Let F be a fault model, SS be a safety specification and p be a fail-safe F-tolerant program for SS. The* maximum detection latency LM_p of *p is defined as the maximum of $L_p(\alpha)$ for all computations α of p in the presence of faults.*

Lemma 4 (Latency of fast detectors) *Given a fault-tolerant program p' which is the result of the composition of a fault-intolerant program p with a set of fast perfect detectors. Then p' has maximum detection latency* 0.

Proof. Consider any computation $\alpha = s_0 \cdots s_i \cdots s_m$ of p' which satisfies Definition 16. We need to show that $s_i = s_m$.

1. Definition 16 implies that there exists a computation σ of p which can be written as $\sigma = \alpha \cdot \beta$ (i.e., a continuation of α) which violates SS.
2. Step 1 and Definition 14 imply that (s_i, s_{i+1}) is the earliest SS-inconsistent transition of p w.r.t. σ.
3. Step 2 and the definition of fast detectors imply that p' evolved from p by removing (among other transitions) also (s_i, s_{i+1}).
4. Step 3 implies that $s_i = s_m$, which in effect means that $L_{p'}(\alpha) = 0$.

Since we have not restricted the choice of α, the statement holds for all α. This implies that $LM_{p'} = 0$. □

Since the maximum detection latency of a fail-safe fault-tolerant program p' must be at least 0, composition of a fault-intolerant program p with fast perfect detectors results in a fail-safe fault-tolerant program p' with optimal detection latency. It remains to be shown that this composition preserves the correctness.

Lemma 5 (Fast perfect detectors and fault-free behavior) *Given a program p' that is the composition of a fault-intolerant program p and a set of fast perfect detectors. For p and p' holds:*

1. *In the absence of faults every computation of p' is a computation of p.*
2. *In the absence of faults every computation of p is a computation of p'.*

Proof. The proof is the same as that of Lemma 1. □

Lemmas 3 (behavior in the absence of faults), 5 (behavior in the presence of faults), and 4 (optimal detection latency) will form the basis for deriving the transformation algorithm for adding fast, perfect detectors in Section 6.

5.4 Summary

In this section, we have presented two main themes, namely (i) a theory for design of perfect detectors, and (ii) a theory for design of fast detectors. With Proposition 3 we have shown that when a program p violates its safety specification SS, this is due to so-called SS-inconsistent transitions. Given that we want the fail-safe fault-tolerant program p' to behave as the corresponding fault-intolerant program p in the absence of faults, we have shown that composing p with perfect detectors in a certain way achieves this. Perfect detectors are defined directly using the notion of SS-inconsistent transitions. Further, we developed the concept of an earliest SS-inconsistent transition that underpins fast error detection and allows to prove optimal detection latency in our case (Lemma 4).

We close this section with a final remark concerning an observation made by Arora and Kulkarni [3, Sect. 3]. The authors observed that "based on their experience", when designing fail-safe fault-tolerant programs, the detectors for non-critical actions are trivial, i.e., "true", whereas the detectors for critical actions are non-trivial. Lemma 3 formally justifies the validity of Arora and Kulkarni's observation since it shows that it is sufficient to compose critical actions with perfect detectors to ensure fail-safe fault-tolerance. Proving this statement was made possible by our notions of accuracy and completeness of detectors. In this sense these properties can be regarded as a concretization of "non-trivial".

In the next section, based on the results developed in this section, we provide an algorithm that automates synthesis of a fail-safe fault-tolerant algorithm with perfect and fast error detection capabilities.

6 Adding Efficient Fail-Safe Fault Tolerance

In this section we give an algorithm to solve the transformation problem of Definition 7 which follows from the theory presented in Section 5

The basic idea of the algorithm is to remove the set of earliest inconsistent transitions from the input program p. Intuitively, the algorithm works as follows: It takes as parameters the fault-intolerant program p (in the form of its transition relation δ_p) and the fault model F (in the form of the set of fault transitions). The safety specification SS is encoded as the set of bad transitions and passed to the algorithm in variable ss.

Starting from the set of bad transitions in ss, the algorithm constructs the set it of all inconsistent transitions. From this set, it constructs the set eit of earliest inconsistent transitions. This set of transitions is removed from δ_p yielding the transition relation of the transformed program. The algorithm is presented in Figure 3.

```
add-efficient-fail-safe($\delta_p, \delta_F, ss$: set of transitions):
    $it := \{(s_0, s_1) \mid \exists \alpha = s_0 \cdot s_1 \cdot s_2 \cdots$ of program transitions:
            $\exists (s, s') \in ss : (s, s')$ occurs in $\alpha$ $\}$
    $eit := \{(s_0, s_1) \mid (s_0, s_1) \in it \wedge \exists s \in S_p : (s, s_0) \in \delta_F\}$
return ($p' = p$ where transition relation is $\delta_p \setminus eit$)
```

Figure 3: Algorithm to add efficient fail-safe fault-tolerance.

Theorem 2 (Correctness the of transformation algorithm) *The algorithm in Figure 3 solves the transformation problem of Definition 7. Furthermore, the resulting program has optimal detection latency.*

Proof. Since the algorithm constructs p' by removing the set of all earliest inconsistent transitions, we can apply the lemmas from Section 5. Lemma 5 ensures that p and p' have the same fault-free behavior which proves the second and third requirements of Definition 7. Lemma 3 ensures that p' satisfies the specification in the presence of faults, which proves the first requirement of Definition 7. Lemma 4 ensures that the maximum detection latency is 0, meaning that it is trivially optimal. □

In contrast, the algorithm of Kulkarni and Arora [11] generates programs with detection latency equal to the maximum length over all partial computations considered when computing set it, since they remove only bad transitions, i.e., the last transition in the partial execution, whereas we remove the first one.

We now provide a brief analysis of the complexity of our algorithm:

1. Assume that the number of bad transitions specified by ss be m. Thus, to compute set it, the number of partial computations visited is $O(m)$.
2. Assume that the maximum number of transitions visited in any partial computation when computing set it is n. Then, the number of states visited is $O(n)$.
3. Therefore, maximum number of states visited when computing set it is $O(mn)$.
4. Computing set eit means going through set it, thus this step has complexity $O(mn)$.
5. Overall, the algorithm in Figure 3 has complexity $O(mn + mn) = O(mn)$, where m is the number of bad transitions specified by ss, and n is the maximum number of transitions considered in any partial computation.

The complexity of our algorithm is no more than the complexity of the algorithm presented by Kulkarni, and Arora [11], which has polynomial complexity in the state space of the program.

7 Discussion and Conclusions

In this paper, we have presented a novel theory of fast and perfect detector components for the design of fail-safe fault-tolerant programs. This theory allows to derive a transformation algorithm which automatically adds fault-tolerance abilities with optimal detection latency. This theory also solves the previously open problem of systematic design of fast and perfect error detection [14].

Our algorithm is intended mainly for a certain class of programs, termed as *bounded programs*. In bounded programs, the length of the partial executions to be considered when calculating the set it is finite. This means that in the program, there are no infinite loops, rather all loops are bounded. An instance of bounded programs can be found in the domain of embedded applications, more specifically applications where the output is to be written within a bounded number of steps.

The motivation for perfect error detection is obvious in adaptive systems. In adaptive systems, usually (in periods of non-perturbation) a fault-intolerant program p executes. During periods of perturbation, a fault-tolerant version p' of p (with possibly lower efficiency) is switched in. If a detector is not accurate, then p' may be switched in, even when there is no perturbation, lowering the efficiency of the system. If the detector is not complete, it might fail to detect an error entirely. Hence, perfect detection is necessary if the system is to be correct and efficient.

The motivation of optimal detection latency is for fault containment. The earlier an error is detected, the higher is the error containment. If an error is not contained, more sophisticated error recovery mechanisms may be required to correct the fault than if the error is contained. Specifically, if an error is contained, a local recovery procedure may be initiated, but if the error is not contained and the state of several processes is corrupted, local recovery mechanisms may not be adequate.

References

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

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

[3] Anish Arora and Sandeep S. Kulkarni. Designing masking fault tolerance via nonmasking fault tolerance. *IEEE Transactions on Software Engineering*, 24(6), June 1998.

[4] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, Reading, Mass., 1988.

[7] Felix C. Gärtner and Hagen Völzer. Defining redundancy in fault-tolerant computing. In *Brief Announcement at the 15th International Symposium on DIStributed Computing (DISC 2001)*, Lisbon, Portugal, October 2001.

[8] H. Peter Gumm. Another glance at the Alpern-Schneider characterization of safety and liveness in concurrent executions. *Information Processing Letters*, 47(6):291–294, 1993.

[9] Martin Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the International Conference on Dependable Systems and Network (DSN 2000)*, pages 24–33, 2000.

[10] Arshad Jhumka, Martin Hiller, Vilgot Claesson, and Neeraj Suri. On systematic design of consistent executable assertions for distributed embedded software. In *Proceedings of the ACM Joint Conference on Languages, Compilers and Tools for Embedded Systems/Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 74–83, 2002.

[11] Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings*, number 1926 in Lecture Notes in Computer Science, pages 82–93, Pune, India, September 2000. Springer-Verlag.

[12] Sandeep S. Kulkarni and A. Ebnenasir. Complexity of adding failsafe fault-tolerance. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 337–344. IEEE Computer Society Press, July 2002.

[13] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[14] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990. 29 refs.