An Approach to Synthesize Safe Systems

Arshad Jhumka[1], Felix Freiling[2], Christof Fetzer[3] and Neeraj Suri[4*]

1 Dept of Computer Science, University of Warwick, UK
2 Dept of Computer Science, University of Mannheim, Germany
3 Dept of Computer Science, TU Dresden, Germany
4 Dept of Computer Science, TU Darmstadt, Germany
*Corresponding author

Abstract: Detectors are system components that identify whether the system is in a particular state. Detectors can be used to ensure arbitrary safety properties for systems, i.e., they can be used to prevent the system from reaching a "bad" state. Detectors have found application in the area of fault-tolerant systems but can also be used in the area of security. We present here a theory of detectors that identifies the class of *perfect detectors* and explains their importance for fault-tolerant systems. Based on the theory, we develop an algorithm that automatically transforms a fault-intolerant program into a fault-tolerant program that satisfies its safety property even in the presence of faults. We further show how to use some of the results for adding security properties to a given insecure program. We provide examples to show the applicability of our approach.

Keywords: Safe systems, formal methods, security, reliability, program transformation

Copyright © 200x Inderscience Enterprises Ltd.

1 Introduction

Critical applications need to satisfy stringent dependability requirements in their provision of services. Such applications need to remain *safe* in spite of external perturbations from the environment such as fault occurrences or security intrusions. There are various formal approaches to ensure the correctness of these applications, the more prominent being model checking (or some other formal verification techniques) and synthesis approaches.

Synthesis approaches are inherently transformational in nature, in that they transform a given program such that the resultant program has some additional properties. Several such approaches exist in the fault tolerance area, for example, Liu and Joseph [21, 22], Arora *et al.* [2], Peled and Joseph [23] (for a survey of transformational approaches see Gärtner [9]). Similarly, transformational approaches have been developed for the security area, e.g., Schneider [25], Ligatti *et.al* [20], Erlingsson and Schneider [8]. These are sometimes referred to as *program rewriting*. Though both fault tolerance and (part of) security focuses on similar properties, they have grown into separate research threads. In this paper, we will develop a theory that can be applied to both fault tolerance and security areas.

In the realm of transformational approaches for fault tolerance, Arora and Kulkarni [4] showed that the fundamental mechanisms used by these fault tolerance mechanisms can be factorized along two dimensions: (1) prevention of taking bad steps, and (2) guaranteeing that necessary good steps are eventually performed. They thus proposed a modular transformational approach [3] to design fault-tolerant systems from fault-intolerant systems. It was shown [4] that to achieve the first goal, it is necessary and sufficient to compose the program with components called *detectors* that detect the effects of faults, i.e., errors, on the system state, and, to achieve the second goal, it was necessary and sufficient to compose the system with components called *correctors* that correct the errors on the system state. Intuitively, a detector is a program component that detects whether a given predicate holds in a given program state. Examples of a detector are error detecting codes, acceptance tests, self checks, snapshot procedures and executable assertions [13]. Analogous concepts can be found in the security literature. For example, Hamlen *et.al* [12] defined a detector to be a predicate that induces a security policy, and Schneider [25] identifies a set of security automata that detect whether the system is about to violate its security policy. Using detectors it is possible to prevent a program from reaching an unrecoveravly bad state, i.e., they can be used to ensure what Lamport called a *safety property* [18]. In this paper, we focus on the setting where a program should never violate its safety specification, i.e., its desired safety property. This should be true in spite of external perturbations. Thus, it is acceptable for a program to halt when it is about to violate its safety specification. In fault tolerance, such as program is called *fail-safe fault-tolerant*.

The design of efficient detectors is difficult. For example, Leveson *et al.* [19] remark that the "process of writing self checks is obviously difficult" and that often detectors were ineffective, i.e., the detectors did not detect errors when they were present, and some other detectors signalled false alarms, i.e., they would flag an error when no such error is present. Leveson *et al.* concluded that, to design effective detectors, "more training or experience might be helpful". These remarks indicate that sound methodological approaches or frameworks are needed to possibly guide

a programmer in developing effective detectors. Further, it is difficult to effectively locate the detectors such that they effectively detect the erroneous program runs that can violate safety.

To address these problems (design and location of detectors), we develop a theory that guides in the effective design and location of those detector predicates. In this paper, we will develop a theory intended mainly for fault tolerance, but an interesting fact is that a part of the theory is also applicable to enforcing security properties on a program. For fault tolerance, we endeavour to develop automated synthesis approaches to design efficient fail-safe fault tolerance, i.e., they do not trigger false positives and false negatives. Automated synthesis approaches developed on the settings used in this paper have been developed by Kulkarni and Arora [16], however they do not address the problems we focus on in this paper, i.e., their approaches focus on fail-safe fault tolerance design, whereas we focus on *efficient* fail-safe fault tolerance design. Hence, the theory we develop offers more insight into the workings of detector components in fail-safe fault tolerance. To ease understanding of our approach, we will adopt a dual view of programs, namely the syntactic view (guarded program notation) and the semantic view (state transition system). In general, if a security requirement can be formulated as a safety property, our approach can also be used to synthesize secure programs. However, the model of programs we used in security will differ slightly from these, since we will look at programs that cross function boundaries, i.e., programs that consist of function calls. But, some of the fault tolerance results will still apply to the security area.

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

- We present a novel theory of detectors that accurately describes the working principles of detectors in fault-tolerant programs.
- We identify various classes of detectors, and identify their role in the design of efficient fail-safe fault tolerance.
- Based on our theory, we provide a polynomial-time algorithm that systematically transforms a fault-intolerant program into an efficient fail-safe fault-tolerant.
- We show the applicability of the theory through a case study.
- We further show the applicability of the theory in the design of secure programs.

The paper is structured as follows: Section 2 recalls the basic model and terminology we use in the paper. 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 detectors, along with correctness theorems. In Section 6 we present an algorithm that automatically generates a fail-safe fault-tolerant program from the corresponding fault-intolerant program with perfect detection capabilities. To show the applicability of our theory, we develop fault-tolerant program in Section 7. We then discuss the applicability of our approach to the area of security in Section 8. We summarize and conclude the paper in Section 9.

2 Preliminaries

2.1 Programs

A *program* p consists of a finite set of processes $\{p_1, \ldots, p_n\}$. Each p_i contains a finite set of actions, and variables. The set of variables V_p of p is the union of all process variables. Each variable stores a value from a nonempty finite domain and each variable stores an *initial* value drawn from their respective domain.

A *state* (resp. initial state) of p is a function that assigns a value (resp. initial value) to every variable in p. The *state space* S_p *of* p (resp. *initial state space* I_p *of* p) is the set of all possible states (resp. initial states) of p. A *state predicate of* p is a boolean expression over the state space of p. Syntactically, a program action has the form

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

in which the guard is a state predicate of p and the statement is either the empty statement or an instantaneous assignment to one or more variables.

An *action* ac *of* p *is enabled in a state* s if the guard of ac evaluates to "true" under s. An action ac can be semantically represented 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.

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. *Weak fairness* means that if a program action ac is continuously enabled along the states of a computation, then 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 $s' = s_{i+1}$. State s is *reachable by* p iff starting from an initial state of p it is possible to construct a computation which contains s by executing a sequence of guarded actions of p. Otherwise s is unreachable. A transition (s, s') is reachable by p iff s is reachable by p. Otherwise it is unreachable.

2.2 Finite State Machines

Every program can uniquely be represented as a finite state machine, i.e., the tuple (S_p, I_p, δ_p) where S_p is the state space, $I_p \subseteq S_p$ is the set of initial states, and $\delta_p \subseteq S_p \times S_p$ is the state transition relation. When designing systems it is preferable to use the notation of guarded actions since such actions are close to the level of abstraction which is offered by programming languages [7, 5]. Every guarded action ac is an abstraction of a set of transitions in δ_p. The transition (s, s') is *member* of δ_p iff ac is enabled in state s and computation of the statement results in state s'. We say that ac *induces* all these transitions (s, s'). State s is called the *start state* and s' is called the *end state* of the transition.

Finite state machines can be considered an implementation level representation of programs since such machines can be directly implemented in hardware. Given a program p it is possible to compute the corresponding finite state machine in $O(|S_p| \cdot A)$ steps where A is the total number of deterministic actions of p.

2.3 Communication

Two processes p_r and p_w of a program p communicate as follows: for each pair of processes p_r and p_w there exists a set of *shared variables* $V_s \subseteq V_p$. Both processes can read the contents of any variable in V_s, but only p_w can update these variables. This defines the information flow and variable access across two processes. The set V_s represents the interface between processes p_w and p_r.

There exists a set of special variables $V_o \subseteq V_p$ that are shared by some processes (that write to the variables), and the environment that reads them. These special variables are commonly referred to as the *output variables*. There exists also a special set of variables, denoted by V_i, where each of the variables is written to by the environment, and read by a process in p. Such variables are known as *input variables*. Input and output variables represent the interface of the program p with its environment. Whenever V_i and V_o are non-empty, such a program model reflects the system assumptions of distributed embedded applications (like sensors and actuators).

For embedded applications, the fact that a program may initially read external inputs before executing is modeled by providing multiple initial states. We expect the program to periodically write the results of a computation to the output variables.

2.4 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, ϵ: If $\alpha \cdot s \cdot \beta$ and $\gamma \cdot s \cdot \epsilon$ are in S, then so are $\alpha \cdot s \cdot \epsilon$ and $\gamma \cdot s \cdot \beta$. 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.

Intuitively, a fusion-closed specification allows a program to make decisions about future state transitions by looking at its current state only. This means that programs do not need to have a larger state space (i.e., more variables) than the specification. Fusion-closed specifications are non-restrictive in the sense that every specification which is not fusion-closed can be transformed into an equivalent fusion-closed specification by adding history variables [10]. Furthermore, "low-level" specifications like C programs, state-transition diagrams, and specifications written in the widespread Unity logic [5] are fusion-closed [11, 3]. We discuss the consequences of requiring fusion-closed specifications below.

Alpern and Schneider [1] have shown that every specification can be written as the intersection of a *safety specification* and a *liveness specification*. A safety specification demands that "something bad never happens" [18]. This defines a set of "bad" finite computation prefixes that should not be found in any computation.

A liveness specification determines what types of events must eventually happen. Since we are mainly interested in safety specifications we specifiy a formal definition of safety.

Definition 1 (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.*

The notion of a finite computation of not being "bad", i.e., the possibility to extend it to remain in the specification, is captured by the definition of *maintains* which can also be used to define safety specifications.

Definition 2 (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 $\alpha \cdot \beta \in S$.*

2.5 Fault Models and Fault Tolerance

A fault model precisely describes the way operations/components of the system may fail. Fault models have been classified into different domains [24]: time faults (including stopping faults), and value faults. Traditional stopping faults (like silent component crashes) cannot lead *by themselves* to a violation of safety. To violate a safety specification, a system must exhibit one of the disallowed computation prefixes. However, the standard value faults from practice (i.e., bit-flips, stuck-at faults, memory perturbation faults) can directly or indirectly lead to a violation of safety.

At the program level, any fault assumption which endangers a safety specification can be modeled as a set of added actions. In this context, we can broadly categorize faults into two classes: (i) faults that do not directly cause violation of safety, and (ii) faults that directly violate safety. We focus on the first class above, i.e., the subset of fault models which have the desirable property that they can potentially be tolerated: We disallow faults to violate the safety specification *directly*. For example, in the case of distributed embedded applications, if a safety specification constrains the output variables of a program, the fault model may not modify the output variables directly in such way that the fault itself results in a safety violation. We stress again that it is impossible to achieve fail-safe fault-tolerance if we do not make this assumption. In practice, of course, it must still be ensured that the expected faults are covered by the fault model with high probability. This can be attempted by adding redundancy and therefore applying fault-tolerance techniques at a different level of abstraction.

Definition 3 (Fault model) *A* fault model F for program p and safety specification $SSPEC$ *is a set of actions over the variables of p that do not violate $SSPEC$, i.e., if state sequence $s_0, s_1, \ldots, s_j$ maintains $SSPEC$, then the state sequence $s_0, s_1, \ldots, s_j, s_{j+1}$ which results from executing any action from F in s_j also maintains $SSPEC$.*

We call the members of F the *fault actions* (or *faults*). We say that a *fault occurs* if a fault action is executed. We denote by $F(p)$ the program p in presence of faults F, or by δ_p^F when considered in the state transition system.

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 .*

Note that by weakly p-fair we mean that only the actions of p are treated weakly fair, i.e., a fault action might never be executed even if it is continuously enabled. The notions of a state or transition being *reachable in the presence of faults* can be defined analogously to the terms for fault-free computations.

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

If F is a fault model and *SSPEC* is a safety specification, we say that a *program p is F-intolerant for SSPEC* iff p satisfies *SSPEC* but $F(p)$ violates *SSPEC*. We will also write *fault-intolerant* instead of *F-intolerant for SSPEC* if F and *SSPEC* are clear from the context.

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

All standard fault-tolerance mechanisms (like error-correcting codes, state machine replication, rollback-recovery) can be modeled as the composition of a fault-intolerant program with fault-tolerance components [4]. It also can be shown that a class of program components called *detectors* is necessary and sufficient to establish fail-safe fault-tolerance in the context of fusion-closed specifications [3]. Intuitively, a detector is a program module that detects whether a given state predicate is satisfied in a given state. Given our focus in adding fail-safe fault-tolerance, we recall [3] a 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 this 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).

Definition 6 (bad transition) *For a program p, a transition $t \in \delta_p$ is* bad with respect to a safety specification *SSPEC if for all computations σ of p holds: If t occurs in σ then $\sigma \notin SSPEC$.*

Proposition 1 ([15]) *Let SSPEC be a safety specification. If p is an F-intolerant program for SSPEC then δ_p contains a bad transition with respect to SSPEC.*

A bad transition is a concept that is applicable in the finite state machine representation of a program only and not in the guarded action representation. While a guarded action *ac* may induce a bad transition, there may be other transitions induced by *ac* which are not bad transitions. In the implementation of a program

as a finite state machine, Proposition 1 indicates that it is sufficient—in order to maintain a safety specification—to keep track of the current computation and take precautions not to run into one of the bad transitions, which are indeed induced by program actions, which are disallowed by the safety specification.

A detector is a concept from the guarded actions representation of a program [4]. Intuitively, a detector refines the guard of the corresponding action in such a way that the action is never executed whenever the action could induce 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. Given an action $g \rightarrow st$, a detector for this action refines the guard to $g \wedge d$, which was shown to exist for every action in a program by Arora and Kulkarni [4].

Definition 7 (Detector for an action) *Let SSPEC be a safety specification. An SSPEC-*detector d monitoring program action ac of p *is a state predicate of* p *such that executing* ac *in a state where* d *holds maintains SSPEC.*

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

Intuitively, detectors ensure that bad transitions are made unreachable in the presence of faults in the state machine representation of the program. However, designing detectors is not an easy task in distributed systems and has its inherent complexities [17, 14]. Further, manual design of detectors entails that the system has to be formally verified again to verify that no desired properties had been removed and no unwanted properties introduced. To this end, we develop a theory underpinning the design of efficient detectors.

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 *SSPEC* and fault model F.

When deriving p' from p, only fault tolerance should be added. Specifically, two conditions need to be satisfied in the transformation problem:

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

- 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, this means that p' can satisfy *SSPEC* 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 8 (Fail-safe transformation problem) *Let SSPEC be a safety specification, F a fault model, and p an F-intolerant program for SSPEC. Identify a program p' such that the following three conditions hold:*

1. *p' satisfies SSPEC in the 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 any fault-intolerant program into a program that satisfies the above three conditions. The algorithm is based on a theory for perfect detectors which we introduce in the following section.

5 A Theory of Perfect Detectors

This section presents a theory of detector components which helps in the design of fail-safe applications. The theory is centered around the notion of an *SSPEC*-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.

The notion of an *SSPEC*-inconsistent transition is tied to the finite state machine representation of programs while the notion of perfect detectors refers to the guarded actions representation of a program. Hence, the theory both allows to (1) identify general principles about good detector design and (2) derive an algorithm to automatically add efficient fail-safe fault tolerance at "compile time". This algorithm will be presented in Section 6.

5.1 Transition Consistency in the Context of Safety Specifications

The intuition behind the definition of *SSPEC*-inconsistent transition is that faults do not, by definition, directly violate safety. To violate safety, one or more faults need to be followed by at least one or more program transitions. We have already identified the program transitions that violate the safety specification as bad transitions. We call the program transitions starting at the first bad transition of a computation until the preceding fault transition as *SSPEC*-inconsistent transitions. Formally, we define this as follows.

Definition 9 (*SSPEC*-inconsistent transition) *Given a fault-intolerant program p with safety specification SSPEC, and a computation α of p in the presence of faults F. A transition (s, s') is SSPEC*-inconsistent for p w.r.t. α in presence of faults F *iff*

1. *there exists a prefix α' of α such that α' violates SSPEC,*
2. *(s, s') occurs in α', i.e., $\alpha' = \sigma \cdot s \cdot s' \cdot \beta$,*
3. *all transitions in $s \cdot s' \cdot \beta$ are in δ_p, and*
4. *$\sigma \cdot s$ maintains SSPEC.*

Finite State Machine Example. Figure 1 shows a graphical explanation of Definition 9. It shows the state transition relation of a program in the presence of faults (the transition (s_3, s_4) is introduced by F). The safety specification *SSPEC* identifies a bad transition (s_6, s_7) which should be avoided. In the presence of faults, this transition becomes reachable and hence the program if F-intolerant since it exhibits a computation α_1 violating *SSPEC*. In this computation, the three transitions following the fault transition match Definition 9 and hence are *SSPEC*-inconsistent w.r.t. α_1 in the presence of F. Note that an *SSPEC*-inconsistent transition is only reachable in the presence of faults.

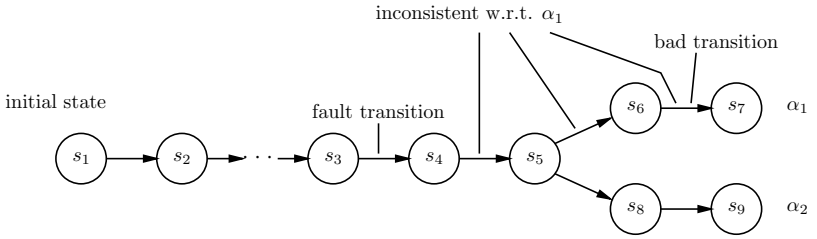


Figure 1 Graphical explanation of *SSPEC*-consistency.

Guarded Command Example. Consider the simple program $P1$ in Figure 2 which reads two sensors, and then outputs the sum of the two readings. The safety specification *SSPEC* requires the output to be always between 10 and 25. The fault transitions state that, from each state, the value of variable x and y can be arbitrarily changed to a value in the range of $[0 \ldots 25]$ and $[0 \ldots 50]$, respectively. 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 *SSPEC* and so no program transition is *SSPEC*-inconsistent. Now consider computation β which violates *SSPEC*:

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

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 *SSPEC*-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 *SSPEC*-inconsistent. The first program transition and the fault transition are not *SSPEC*-inconsistent.

Intuitively, an *SSPEC*-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, *SSPEC*-inconsistent transitions lead the program computation on the "wrong path".

Now we define *SSPEC*-inconsistency independent of a particular computation. Intuitively, we call a program transition *SSPEC*-inconsistent if there exists a "wrong path" for which the transition is *SSPEC*-inconsistent.

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

    c = 1 → x := read(); c := c + 1; // x within [5 ... 10]
    c = 2 → y := read(); c := c + 1; // y within [5 ... 15]
    c = 3 → z := x + y; c := c + 1
    c = 4 → output(z); c := 1 // loop forever

F (faults):
    true → x := random [0 ... 25]
    true → y := random [0 ... 50]
```

Figure 2 Program to illustrate SSPEC-inconsistent transitions (Definition 9).

Definition 10 (*SSPEC*-inconsistent transition for p) *Given a program p with safety specification SSPEC. A transition (s, s') is SSPEC-*inconsistent for p in presence of faults F *iff there exists a computation α of p in the presence of faults F such that (s, s') is SSPEC-inconsistent for p w.r.t. α in presence of F.*

Example. In general, a transition can be *SSPEC*-inconsistent w.r.t. a computation α_1, and not be *SSPEC*-inconsistent w.r.t. α_2 (see Fig. 1). This can be due to nondeterminism in program execution. For example, consider the program $P2$ in Figure 3. The safety specification *SSPEC* mandates that always $10 \leq z \leq 50$. 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 *SSPEC* 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 *SSPEC*-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 *SSPEC*. Hence, the same program transition t as above is not *SSPEC*-inconsistent for p w.r.t. α_2.

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

Definition 11 (*SSPEC*-consistent transition for p) *Given a program p with safety specification SSPEC. A* transition (s, s') is *SSPEC*-consistent for p in presence of faults F *iff (s, s') is not SSPEC-inconsistent for p in presence of F.*

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

Proposition 2 *Given a fault-intolerant program p for a safety specification SSPEC. Every computation α of p in the presence of faults F that violates SSPEC contains an SSPEC-inconsistent transition for p w.r.t. α in presence of F.*

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(); $c1 := c1 + 1$; // w within $[15 \ldots 25]$
  $c1 = 2 \wedge x \leq 15 \rightarrow w := w + 5$; $c1 := 1$; // loop
  $c1 = 2 \wedge x > 15 \rightarrow w := w - 15$; $c1 := 1$; // loop

process b:
  $c2 = 1 \rightarrow x :=$ read(); $c2 := c2 + 1$; // x within $[0 \ldots 20]$
  $c2 = 2 \rightarrow y := w$; $c2 := c2 + 1$;
  $c2 = 3 \rightarrow z := y + x$; $c2 := c2 + 1$;
  $c2 = 4 \rightarrow$ output(z); $c2 := 1$; // loop

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

Figure 3 Program containing two concurrent processes with a transition that is both *SSPEC*-inconsistent and not *SSPEC*-inconsistent w.r.t. two different computations.

Proof.

1. Proposition 1 states that there exists a bad transition (s, s') in α.
2. From proof step 1 and the definition of F follows that $(s, s') \in \delta_p$.
3. From proof step 2 and Definition 9, (s, s') is *SSPEC*-inconsistent for p w.r.t. α.

□

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

Proposition 3 *Given a fault-intolerant program p for a safety specification SSPEC. If (s, s') is an SSPEC-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 *SSPEC*-inconsistent transitions and Proposition 1. □

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

Proposition 4 *Given a fault-intolerant program p for safety specification SSPEC. Every SSPEC-inconsistent transition for p in presence of faults F is not reachable in the absence of F.*

Proof.

1. For a contradiction, assume the start state s of an *SSPEC*-inconsistent transition (s, s') is reachable in the absence of faults.
2. Proof 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 3 there exists a computation $s \cdot s' \cdot \beta$ of p in the absence of faults in which a bad transition occurs.

4. From proof 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 proof step 4 and Proposition 1 there exists a computation of p in the absence of faults which violates *SSPEC*.

6. From proof step 5 p violates *SSPEC* 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 *SSPEC*-inconsistent). However, it can be reformulated to characterize reachable transitions in the absence of faults as *SSPEC*-consistent.

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

In the next section, we introduce the notion of perfect detectors using the terminology of *SSPEC*-consistency and *SSPEC*-inconsistency. The concept of perfect detectors can be regarded as a programming level abstraction of *SSPEC*-inconsistency.

5.2 Perfect Detectors

From the previous section, we observed that *SSPEC*-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 *SSPEC*-inconsistent for p, i.e., starting states that may lead to a violation of safety, and (2) keep the starting states of all induced transitions that are *SSPEC*-consistent for p in presence of faults. We call the first property *completeness* and the second property *accuracy* (see Figure 4). In the following definitions, p is a program with safety specification *SSPEC*, ac is a program action of p, and F is a fault model.

Definition 12 (Detector completeness) *A* detector d monitoring ac is *SSPEC*-complete for ac in p in presence of F *iff for all transitions (s, s') induced by ac holds: if (s, s') is SSPEC-inconsistent for p in presence of F then $s \notin d$.*

Definition 13 (Detector accuracy) *A* detector d monitoring ac is *SSPEC*-accurate for ac in p in presence of F *iff for all transitions (s, s') induced by ac holds: if $s \notin d$ then (s, s') is SSPEC-inconsistent for p in presence of F.*

Definition 14 (Perfect detector) *A* detector d monitoring ac is *SSPEC*-perfect for ac in p in presence of F *iff d is both SSPEC-complete and SSPEC-accurate for ac in p in presence of F.*

In other words, a perfect detector guarantees the equivalence that (s, s') is *SSPEC*-inconsistent for p in presence of F iff $s \notin d$.

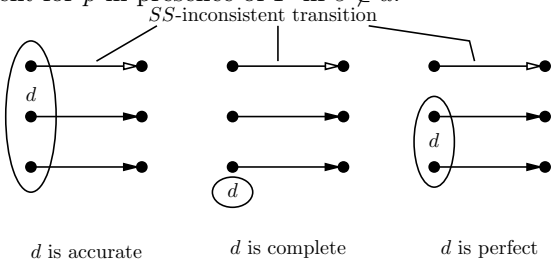


Figure 4 Accurate, complete and perfect detector for an action. The arrows stand for the set of transitions induced by that action.

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

We now raise the low level definitions of perfect detectors to the program design level of guarded actions. 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 "dangerous" *SSPEC*-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 *SSPEC*-consistent transitions in the absence of faults. 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 (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 *SSPEC*-consistent.
2. From construction, p' results from adding perfect detectors to p. Because they are perfect (Definition 14), they are accurate.
3. From proof steps 1, 2 and the definition of accuracy, all *SSPEC*-consistent transitions of p are also transitions of p'.
4. Proof steps 1 and 3 imply that every reachable transition in p is also reachable in p'.
5. Proof 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 7) follows that composition with detectors does not introduce new state transitions.

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

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

□

To understand the behavior in the presence of faults, we make use of the notion of a *critical action*. Intuitively, critical actions are program actions that may directly violate the safety specification. In effect these can only be actions that write output variables.

Definition 15 (Critical and non-critical actions) *An action ac is* critical *iff there exists a transition (s, t) induced by ac such that (s, t) is a bad transition that is reachable in presence of faults F. Otherwise n action is* non-critical.

The set of bad transitions specified by the safety specification $SSPEC$ of a program p defines a set of critical actions in p. The following lemma gives a programming level design guideline for adding fail-safe fault-tolerance.

Lemma 2 (Behavior in the presence of faults) *Given a fault-intolerant program p for a safety specification SSPEC,and fault class F. Given also a program p' by composing the critical actions of p with their corresponding perfect detectors. Then, p' satisfies SSPEC in presence of faults F.*

Proof.

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

2. Proof step 1 and Proposition 1 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 proof step 2 must be a program transition (i.e., $(s, s') \in \delta_{p'}$)

4. From proof step 3, and Definition 15, there exists a critical action ac that induces the bad transition from proof step 3

5. From Definition 9 and proof step 3 the transition (s, s') is $SSPEC$-inconsistent.

6. Consider the critical program action ac (from proof step 4) causing the bad transition. From construction of p', ac is composed with a perfect detector d.

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

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

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

□

Taken together, Lemmas 1, and 2 show that composing the critical actions of a fault-intolerant program with perfect detectors is sufficient to solve the transformation problem defined in Section 4. The final question to answer is whether a perfect detector exists for every critical action in program p? We assert a positive answer to this as shown by the following lemma.

Lemma 3 (Existence of perfect detectors) *Given a program p with safety specification SSPEC, and fault model F. For each critical action ac in p, there exists a detector d such that d is perfect for ac in p in presence of F.*

Proof.

1. From assumption, action ac in p is critical.
2. From 1, and Definition 15, there exists a set B of reachable bad transitions in presence of F, i.e., $B = \{(s,t) : (s,t)$ *is a reachable bad transition in* $\delta_p^F \wedge (s,t)$ *is induced by* $ac\}$.
3. Let ac_r be the set of all transitions induced by ac reachable in the presence of faults.
4. From proof steps 2, and 3, the set $O = ac_r \setminus B$ is the set of all transitions induced by ac reachable in presence of faults that will not cause violation of $SSPEC$ when executed.
5. From proof step 4, set O does not contain any reachable transition (s,t) induced by ac that is $SSPEC$-inconsistent (bad) for p.
6. From proof step 4, set O contain all reachable transitions (s,t) induced by ac that are $SSPEC$-consistent for p.
7. From proof steps 5 (completeness), and 6 (accuracy), the set $OS = \{s : (s,t) \in O\}$ defines a state predicate (thus a detector) that is perfect for ac in p.

□

Thus, we have shown that composing critical actions of a program with perfect detectors is sufficient to ensure that the resulting program is fail-safe fault-tolerant, as well as preserving its behavior in absence of faults. We have also shown that such detectors always exist for each critical action.

5.3 Constructing Perfect Detectors

Finally, we study how to automate the construction of perfect detectors for critical actions. The following theorem shows how to implement the design guideline derived from Lemmas 1, and 2 on the finite state machine level.

Theorem 1 (Constructing perfect detectors) *Given a fault-intolerant program p with safety specification SSPEC, and fault model F, and a program p′ that satisfies SSPEC in the presence of F. The following two statements are equivalent:*

1. *The program p' can be obtained by composing each critical action ac of p with a perfect detector for ac in p in presence of F.*

2. *Each SSPEC-inconsistent transition induced by the critical action ac of p reachable in the presence of F is unreachable in p' in presence of F, and each SSPEC-consistent transition of p reachable in the presence of F is also reachable in p' in presence of F.*

Proof. We assume a given critical action ac of p being composed with a detector that is perfect for ac in p in presence of F, and show the implication of the second statement from the first statement.

1. ac is composed with a detector d that is perfect for ac in p in presence of F.

2. Since d is perfect, it causes all $SSPEC$-inconsistent transitions induced by ac to be unreachable in p' in presence of F (Definition 12).

3. Since d is perfect, it causes all $SSPEC$-consistent transitions induced by ac to still be reachable in p' in presence of F (Definition 13).

4. From proof steps 2 and 3, we have statement 2.

The proof for the implication of statement 1 from statement 2 is straightforward. The first part of statement 2 (unreachability of $SSPEC$-inconsistent transitions induced by a critical action ac in presence of faults) implies completeness of a detector d monitoring ac), and the second part of statement 2 (reachability of $SSPEC$-consistent transitions induced by a critical action ac in presence of faults) implies accuracy of a detector d monitoring ac. Taken together, d is perfect for ac in p in presence of F.

□

The algorithm for synthesizing perfect detectors (or fail-safe fault-tolerant programs with perfect detection) is based directly on Theorem 1 and presented in the following section.

6 Algorithm for Adding Perfect Fail-Safe Fault Tolerance

In this section, we present a sound and complete algorithm for synthesizing fail-safe fault-tolerant programs with perfect detection. It is based on the fact that composing critical actions of a fault-intolerant program p with perfect detectors results in a fail-safe fault-tolerant program p' whose behavior in the absence of faults is identical to that of p.

6.1 The Algorithm

Theorem 1 suggests that it is both necessary and sufficient to remove all reachable bad transitions in presence of faults, while keeping all reachable "non-bad" transitions in presence of faults. Algorithm *add-perfect-fail-safe* exploits this construction method by first computing the set of reachable bad transitions in presence of faults, and then making these unreachable by removing them. The input to the

transformation algorithms are three sets of transitions: δ_p is the state transition relation of the fault-intolerant program p, δ_F is the set of transitions induced by the fault actions of F, and ss is the safety specification given as the set of bad transitions. The procedure returns the state transition relation of the fault-tolerant program p'.

```
% Synthesize perfect detectors
add-perfect-fail-safe(δ_p, δ_F, ss: set of transitions)
{
  % calculate set of reachable ss-inconsistent
  % transitions induced by critical actions
  ss_r := {(s,t)| (s,t) is induced by a critical action of p and
       (s,t) is reachable using transitions in δ_p^F and
       (s,t) is SSPEC-inconsistent for p in presence of F}
  return (δ_p \ ss_r)
}
```

Figure 5 Algorithm to synthesize fail-safe fault-tolerant program with perfect detection.

6.2 Correctness of the Algorithm

We now prove that the algorithm is sound and complete. Soundness means that any result found by the algorithm is correct with respect to the transformation problem. Completeness means that if there exists a solution to the transformation problem then the algorithm will find it.

Theorem 2 (Soundness of the transformation algorithm) *The returned program of the transformation algorithm in Figure 5 satisfies the properties of the transformation problem of Definition 8.*

Proof. Since the algorithm constructs p' by removing the set ss_r of all $SSPEC$-inconsistent transitions induced by critical actions reachable by using transitions in δ_p^F, hence, from Theorem 1, p' is obtained by composing each critical action of p with a perfect detector. From Lemma 1 we then have that p' maintains the fault-free behavior of p. From Lemma 2 we have that p' satisfies the safety specification in the presence of faults. Hence, both requirements of Definition 8 are satisfied. □

Theorem 3 (Completeness) *If there exists a solution to the transformation problem, then the algorithm in Figure 5 will find it.*

Proof. The algorithm in Figure 5 works by removing all $SSPEC$-inconsistent transitions induced by critical actions that are reachable in presence of faults. Hence, for the algorithm to find the solution, the algorithm needs to be able to construct the set ss_r of Figure 5. To compute ss_r, we need to compute the set ss, and this can be achieved in $O(|S_P|)^2$. As we assume programs with finite state space, the construction of the set ss, and hence of ss_r, always terminates. Hence, the algorithm is complete.

□

6.3 Algorithm Complexity

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

1. Assume that the number of bad transitions given ss be m.
2. Assume that the maximum number of transitions visited to determine reachability of a bad transition is n. Then, the number of transitions visited is $O(n)$.
3. Therefore, maximum number of transitions visited when computing set ss_r is $O(m \cdot n)$.
4. Removing set ss_r has complexity $O(m)$, since the size of set ss_r is $O(m)$.
5. Overall, the algorithm in Figure 5 has complexity $O(m \cdot n + m) = O(m \cdot n)$, where m is the number of bad transitions specified by ss, and n is the maximum number of transitions considered to ascertain reachability.

The complexity of our algorithm is no more than the complexity of another algorithm presented by Kulkarni, and Arora [16], which also has polynomial complexity in the state space of the program.

In the next section, we present an example to show the applicability of our approach.

7 Fault-Tolerance Case Study

In this section, we demonstrate the practical applicability of the automatic transformation procedure of section 6 when applied to a fault-intolerant distributed mutual exclusion program. In this section, we use the notation of guarded commands to present the example. It should be noted that the actual transformation is computed on the transition system representation of the program and that presenting the resulting fail-safe fault-tolerant programs in the guarded command notation is for expository purposes only (this representation has to be manually extracted from the transition system representation in practice).

To start the example, first recall the problem of mutual exclusion which can be regarded as the specification of a token ring. In the mutual exclusion problem, multiple processes have a special section of their code which is called *critical section*. Processes may wish to enter the critical section, e.g., to access a shared resource. Processes leave the critical section in finite time. A protocol solving mutual exclusion guarantees that at any point in time at most one process is in its critical section. This is the safety specification of mutual exclusion. The liveness specification states that if a process wants to enter its critical section, it will manage to do this in finite time.

We implement mutual exclusion by using a token ring. For this, we assume that the processes are arranged in a ring and these processes circulate a token in a particular direction. Whenever a process wants to access its critical section, it waits for the token to arrive. After accessing the critical section, it forwards the token to the next process in sequence.

In our example, there are $N+1$ processes, numbered from 0 to N, which are arranged in sequence a ring. Process k with $0 \leq k < N$ passes the token to process $k+1$, whereas process N passes the token to process 0. Each process k has a binary variable, $t.k$. All variables are initialized to the same value. Every process has just one action. If it executes this action, it is said to "pass the token" and execute its critical section. The safety property of mutual exclusion translates to the property that there are never two guarded statements in the program that could execute concurrently.

All processes k $(0 < k \leq N)$ compare their value $t.k$ with that of the predecessor $t.(k-1)$ in the ring. If both values are not equal, they are made equal by executing the action. Similarly, process 0 compares its value $t.0$ with the value $t.N$ of process N. If both values are the same, they are made unequal by executing the action. The fault-intolerant program for the token ring is as follows:

ITR1 ::	$k \neq 0 \wedge t.k \neq t.(k-1)$	$\rightarrow$	$t.k := t.(k-1)$
ITR2 ::	$k = 0 \wedge t.k = t.N$	$\rightarrow$	$t.k := \neg t.N$

We consider faults that may corrupt the token variables at the processes in a detectable way, i.e., by setting t to a "bad" value $\perp$. If we would allow direct modifications to t, then no fail-safe tolerant program would exist since it is easy to construct a computation in which multiple faults lead to multiple tokens in the ring. The fault actions we consider are formalized as follows (one such action exists for every k):

F ::	$t.k \neq \perp$	$\rightarrow$	$t.k := \perp$

In the presence of faults, the values of several processes can be set to $\perp$ and, following their program, they may execute actions independently, violating the safety property. The set of bad transitions ss therefore contains all transitions of process k that start in a state where $t.(k-1) = \perp$. Note that every action of the processes induces such a transition and so all actions are critical actions (according to Definition 15). Hence, we expect our transformation algorithm to modify all actions with a detector.

Applying the transformation procedure to the fault-intolerant token ring program yields the resulting program:

FSTR1 ::	$t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1)$	$\rightarrow$	$t.k := t.(k-1)$
FSTR2 ::	$t.N \neq \perp \wedge k = 0 \wedge t.k = t.N$	$\rightarrow$	$t.k := \neg t.N$

Observe that now there are never two guarded statements that could execute concurrently in the system, resulting in the program to be fail-safe F-tolerant.

8 Theory Applied to Security

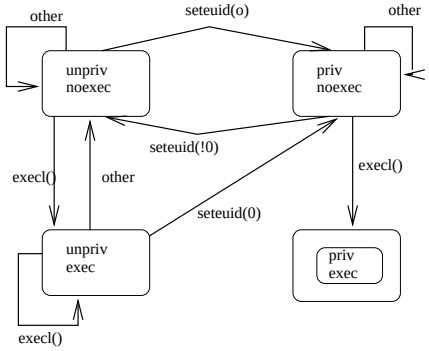
In the rest of the paper, we show how some of the concepts developed earlier for the fault tolerance case can be adapted to deal with security.

The model used in the security section differs from the one previously introduced. Rather than dealing with a state-based view of system execution, we focus on an event-based perspective. Here, we present a brief overview of the models.

Let Σ be the set of security-related operations. Let $R \subseteq \Sigma^*$ be the set of security operations sequences that violate the security property. A trace $\tau \in \Sigma^*$ represents a sequence of operations executed by a possible path through the program. We denote by $T_p \subseteq \Sigma^*$ the set of traces generated by a program p. As before, the problem is to decide if $T_p \cap R$ is empty. If the set is non-empty, them the program p does not satisfy the security property.

In this model, R and T_p are arbitrary languages. Deciding $T_p \cap R$ is an undecidable problem since T_p is in general an uncomputable set. Hence, we have to restrict the form of R and T_p to make the problem decidable. In general, a program p will comprise function calls, whose return addresses need to be recorded on a stack. The language generated by a stack in context-free. Further, we assume that the set R is a regular language. Since we model R as a regular language, it implies that there exists a finite state automaton A_R that accepts R, i.e., $R = L(A_R)$. Also, since we model T_p as a context-free language, it implies that there exists a Pushdown Automaton P_p that accepts T_p, i.e., $T_p = L(P_p)$. Hence, we now need to decide whether $L(P_p) \cap L(A_R)$ is empty. Using context free languages to model the set of traces introduces some imprecision, since in general $T_p \subseteq L(P_p)$. Now that the original problem of determining $R \cap T_p$ is now that of determining $L(P_p) \cap L(A_R)$, and given that $T_p \subseteq L(P_p)$, we have $R \cap T_p \subseteq L(P_p) \cap L(A_R)$. The impact of this result is that any analysis performed on the program is *sound*, i.e., the analysis may return some false positives, but never false negatives. In light of the theory developed earlier, this means that for safety, we only need to use detectors that are complete, and not necessarily accurate.

We show this via an example.




```
int main(int argc, char *argv[])
{//do something with privilege
m0: do_something_with_priv();
m1: drop_priv();
m2: execl("/bin/sh","/bin/sh",NULL);//risky system call
m3:}

void drop_priv()
{
    struct passwd *passwd;

d0: if ((passwd = getpwuid(getuid())) == NULL)
d1          return;// forget to drop priv
d2  fprintf(log,"drop priv for %s",passwd->pw_name);
d3: seteuid(getuid()); // drop priv
d4: }
```

Figure 6 A Security Example.

The example in Fig. 6 comes from UNIX applications, and have been used by Chen and Wagner [6]. A privileged process has full access to permission to the system, and it should not make certain system calls that run untrusted programs without first dropping all privileges. One such system call is *execl()*. The program in Fig. 6 makes a bad system call by calling *execl()* without having dropped the privileges, hence the program is unsafe. The automaton specifies the safety property that the program has to satisfy, and the program will cause the automaton to end up in a final (unwanted) state..

For a program to be safe, it is sufficient to compose its critical actions with complete detectors. Further, since the theory is developed under the assumption of fusion closure, we enhance the program with a variable *state* that keeps track of the history of the trace. The critical action in the program is the *execl()* system

call. To execute this system call from state 0 (i.e., priv,noexec) means that the automaton will end up in a final state (unwanted). Hence, we need to enhance the program with the movement of the automaton. Hence, we refine the program by incorporating (state = 0) at the beginning since the automaton starts in that state. Also, as code is executed, we keep track of the movement of the automaton. To this end, we add (state = 1) after the program executes the *seteuid* system call. This causes the automaton to enter state 1 (i.e., unpriv,noexec).

Once the program has been refined with code that encodes the automaton, adding a complete detector to check whether the automaton is in state 0 and about to execute a *execl* system call is trivial. The detector is *if state == 0*. Since we focus on safety, the system can halt, so when the detector evaluates to True, we cause the program to halt. This is shown in Fig. 7. The instrument program will no longer cause a violation of safety.

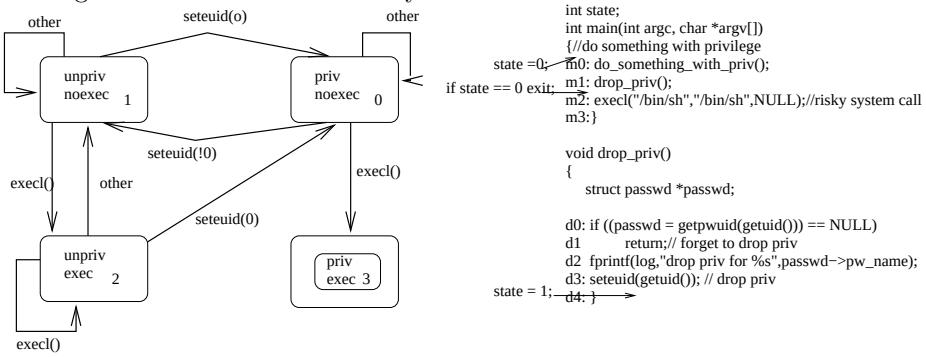


Figure 7 A Security Example.

9 Conclusion

In this paper, we have developed a novel theory of detectors and identified perfect detectors as a class of detectors that preserves the efficiency of fail-safe fault-tolerant programs. The theory is based on the notion of a transition that is inconsistent with the safety specification of the program. We have developed necessary and sufficient conditions for the addition of efficient fail-safe fault tolerance. We also show how some results from the fault tolerance area can be reused to enforce a given security property.

References and Notes

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

2 Anish Arora, Paul C. Attie, and E. Allen Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 173–182, 1998.

3 Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 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 K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, Reading, Mass., 1988.

6 Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *SIGSAC: 9th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2002.

7 Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

8 Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

9 Felix C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science (J.UCS)*, 5(10):668–692, October 1999. Special Issue on Dependability Evaluation and Assessment.

10 Felix C. Gärtner and Arshad Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Grenoble, France, September 2004.

11 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.

12 Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *j-TOPLAS*, 28(1):175–205, January 2006.

13 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.

14 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.

15 Sandeep S. Kulkarni. *Component Based Design of Fault-Tolerance*. PhD thesis, Department of Computer and Information Science, The Ohio State University, 1999.

16 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.

17 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.

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

19 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.

20 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec*, 4(1-2):2–16, 2005.

21 Zhiming Liu. *Fault-tolerant programming by transformations*. PhD thesis, University of Warwick, Department of Computer Science, 1991.

22 Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

23 Doron Peled and Mathai Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 128:99–125, 1994.

24 David Powell. Failure mode assumptions and assumption coverage. In Dhiraj K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 386–395, Boston, MA, July 1992. IEEE Computer Society Press.

25 Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security*, 3(1):30–50, February 2000.