A Framework for the Design and Validation of Efficient Fail-Safe Fault-Tolerant Programs

Arshad Jhumka, Neeraj Suri
Department of Computer Science
TU - Darmstadt, Germany
Email: {arshad,suri}@informatik.tu-darmstadt.de
Martin Hiller
Department of Electronics and Software
Volvo Technology Corporation
Göteborg, Sweden
Email: martin.hiller@volvo.com

Abstract. We present a framework that facilitates synthesis and validation of fail-safe fault-tolerant programs. Starting from a fault-intolerant program, with safety specification SS, that satisfies its specification in the absence of faults, we present an approach that automatically transforms it into a fail-safe fault-tolerant program, through the addition of a class of detectors termed as *SS-globally consistent detectors*. Further, we make use of the SS-global consistency property of the detectors to generate pertinent test cases for testing the fail-safe fault-tolerant program, or for fault injection purposes. The properties of the resulting fail-safe fault-tolerant program are that (i) it has minimal detection latency, and (ii) perfect error detection. The application area of our framework is in the domain of distributed embedded applications.

Keywords: detectors, software synthesis, fault tolerance, fail-safe, test cases
Contact author: Arshad Jhumka (arshad@informatik.tu-darmstadt.de)

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 the 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* [3].

In this paper we restrict our attention to designing *fail-safe* fault-tolerant programs. Intuitively this means that it is acceptable that the fail-safe fault-tolerant program "halts" when 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 than continuous provision of service. Thus, a fail-safe fault-tolerant program has to satisfy at least its safety specification[1] in presence of faults. Arora and Kulkarni, in [3], showed that fail-safe fault-tolerance can be achieved by merely employing detectors, i.e., a fail-safe fault-tolerant program can be obtained by composing the corresponding fault-intolerant program with detectors only. In safety-critical systems, detection may be the only option [10], and once faults have been detected, a back-up system takes over.

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. However, use of replication is often computationally expensive. Another (maybe more obvious) example of a detector is error detecting codes. Other error handling mechanisms like acceptance tests, self checks or executable assertions can also be formulated as detectors in the sense of Arora and Kulkarni [3]. Hence, reasoning at the level of detectors makes an approach applicable to many different practical settings.

However, the design of *efficient* fail-safe fault-tolerant programs is problematic, since design of efficient detectors is difficult, as observed in [10]. For design of fail-safe fault-tolerant programs, programmers tend to program defensively to ensure that the safety specification is not violated, i.e., they use very "restrictive" detectors. In formal terms, this means that those detectors are not *accurate*. When detectors are not accurate, the efficiency of the system (for example, response times, QoS etc) may decrease. For example, given a certain program with some valid inputs where these are flagged as erroneous by an inaccurate detector, there may not be any response at all, since the system may have "halted". Inaccurate detectors may still preserve safety, however there may be an associated decrease in performance.

Similarly, detectors may fail to detect certain erroneous situations, i.e., the detectors are not restrictive enough. In formal terms, this means that the detectors are not *complete*. Hence, for design of *efficient* fail-safe fault-tolerant systems, detectors need to be both accurate, and complete, i.e., detectors need to be *perfect*[2] (detectors are complete and accurate). However, there is a dearth of frameworks or guidelines pertaining to the design of efficient detectors (or efficient fail-safe fault-tolerant programs).

In this paper, our approach is to transform an initially fault-intolerant program with safety specification SS that satisfies its specification in the absence of faults, but violates it in the presence of faults [3], into an efficient fail-safe fault-tolerant program, through the addition of a class of detectors, termed as *SS-globally consistent detectors*. We will later show that composing a given fault-intolerant program with a SS-globally consistent detectors results in a fail-safe fault-tolerant program that has *minimal* detection latency, and *perfect detection*[4]. Once the fail-safe fault-tolerant program has been obtained, it needs to

[1] We will explain this term in Section 2.

[2] Classical measures for efficiency of detectors are detection coverage, and latency.

[3] We will refer to this program as a fault-intolerant program

[4] Hence, SS-globally consistent detectors are instances of perfect detectors

be validated, through testing or fault injection. Both approaches can be computationally expensive, since they require generation of test cases. For this case, we exploit the SS-global consistency property of detectors to efficiently and automatically generate test cases.

Thus, our contributions are the following:

1. We introduce a class of detectors called *globally consistent detectors* which are instances of perfect detectors, and show, by means of examples, how fail-safe programs are obtained.
2. We explain how the SS-global consistency property can be exploited for systematic and automatic generation of test cases for validation (i.e., testing, or fault injection)

Throughout the paper, we will use examples to illustrate the different concepts involved in our approach. Our framework allows automatic synthesis of a fail-safe fault-tolerant program, as well as automatic generation of test cases for its validation. To the best of our knowledge, this framework is novel, since no other work has addressed the design of fail-safe fault-tolerant programs with perfect detection, and minimal detection latency in a systematic manner.

The paper is structured as follows: Section 2 presents the models (system, faults) used in the paper. Section 3 explains the role of detectors in the provisioning of fail-safe fault tolerance. Section 4 explains the concept of adding fail-safe fault tolerance to a fault-intolerant program. Section 5 introduces a class of detectors, called *SS-globally consistent detectors*, for which design is polynomial in the state space of the fault intolerant program. We further show how pertinent test cases for validating the fail-safe fault-tolerant program can automatically be generated from knowledge of the SS-globally consistent detectors in Section 6. In Section 7, we perform fault injection experiments to ascertain the viability of concept of SS-global consistency. We summarize the paper in Section 8.

2 Preliminaries

In this section, we will present the basic notations and terminologies that will underpin our presentation.

2.1 Program

A program P consists of a set of variables V_P, and a set of actions A_P, both partitioned among n processes $p_1 \ldots p_n$. Each variable in V_P stores a value from an associated predefined non-empty, but finite, domain, and each action in P updates the value of one or more variables in V_P. A given value association with variables in V_P is called a *state* of P, and the set of all such possible value associations defines the *state space* of P. There also exists a subset of the state space of P that we refer to as the *set of initial states* of P. We assume actions of P to be deterministic, however execution of actions of P is non-deterministic. An *event* is said to occur when a program action executes. A given event can be *good* or *bad*, depending on whether or not it violates the safety specification.

An action defines a set of transitions, and the set of actions defines the complete transition system of the program.

Two processes p_r and p_w of P communicate as follows: there exists a set of "shared" variables V_s between p_r and p_w. In such cases, for each variable in V_s, p_r is the reader of that variable, and p_w the writer, i.e., if p_w (the writer) updates the variable, then p_r (reader) reads it. This defines the information flow between two processes, and V_s is the interface between p_i and p_j.

There exists a set of special variables, denoted by V_o, 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. Such program model is suitable for embedded applications, for which our framework is targeted.

Also, we assume programs to contain *critical actions* and *non-critical actions*. Critical actions are those that need to be monitored with detectors, while the non-critical actions do not [2]. Examples of critical actions for embedded systems are those actions that control progress, i.e., those actions that provide the output value, or commit some value to the environment.

A detector[5] D in program P monitoring an action A of P is a boolean expression over the state space of P. Specifically, when D evaluates to "True" in a given state, that state is considered a *valid* state of P, and it also means that execution of action A can safely take place. In Section 3, we will explain in more details the role of detectors in ensuring fail-safe fault-tolerance.

2.2 Specification

A specification of a program consists of two parts, namely (i) a safety specification, and (ii) a liveness specification [1]. Given our focus on fail-safe fault-tolerance, we will explain safety specifications only. The liveness specification is needed so as to rule out any trivial program, such as one that does nothing, which always satisfies the safety specification. We also assume the specification to be *fusion-closed*. Informally, fusion-closure of a specification guarantees that the entire history of a given execution of the program "is available" in the current state, such that it is possible to determine if the next action to be executed is "desirable". It has been observed [2] that low level specifications, such as C programs, are fusion-closed. In general, a specification that is not fusion-closed can be converted into a fusion-closed specification by the addition of history variables, so the fusion-closure requirement is not a hindrance.

Informally, a safety specification of a program states that "something bad never happens". Specifically, it rules out certain *sequences of events* that should never happen during execution of P. However, the fusion-closure property of the specification allows identification of a *set of events* (rather than set of sequences of events) that should not occur in any given execution of the program. Therefore,

[5] A detector in our context will be an executable assertion.

we take the safety specification of a program to specify the set of events that should not occur in any execution of the program, i.e., it specifies the set of bad events. Also, fusion-closure of a specification guarantees the existence of detectors (detection predicates) [2].

2.3 Faults

In this paper, we focus on the set of fault models that can potentially be tolerated, i.e., we do not consider faults that directly violate the safety specification of the program. For example, if the safety specification constrains the output variables of a program, as is often the case in embedded applications, then we disallow the faults to directly modify the output variables of the program that could result directly in a safety specification violation. However, faults can arbitrarily alter the state of the program in such a way that subsequent execution of program actions can lead to violation of the safety specification. Thus, safety is violated due to execution of a certain program action, such that the corresponding event is ruled out by the safety specification.

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

We adopt the view of Arora and Kulkarni [3] 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.

Recall that the safety specification of a program P specifies a set of events that should not occur during any execution of P, i.e., the set contains bad events. Intuitively, to avoid violating a safety specification requires to keep track of the current program execution (history) and take precautions so that none of the events which are disallowed by the safety specification (bad events) occurs. From our restrictions of the fault model (faults do not directly violate safety), we know that these bad events occur when program actions are executed. Thus, a detector monitoring a given action in the program works in such a way that the action is never executed whenever its execution will result in the occurrence of a bad event. Overall, a detector allows execution of a corresponding program action only if its execution is "safe" (not a bad event). Also, it was shown in [6] that a bad event cannot occur without the occurrence of faults. This means that if no fault occurs, then only good events are observed from the program. Detectors can also prevent potentially bad events from occurring [6], i.e., they prevent events that can potentially lead the program to violate its safety specification from occurring. Such potentially bad events may also be considered as bad events. Thus, the safety specification can be extended to also rule out those potentially bad events.

However, designing detectors has its inherent complexities [7, 8]. In subsequent sections, we will explain how detectors can be designed that will transform a fault-intolerant program into a fail-safe fault-tolerant one.

At this point, we provide an example to illustrate some of the concepts we have presented:

```
Program $P1$
    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 $[10 \ldots 50]$
```

Fig. 1. An example program to illustrate the different concepts

In the example in Fig. 1, the program a is written in the UNITY logic [4]. Variables c_1 and c_2 are two program counters, for process a and b respectively. For example, in process P_1, the first statement says that when the program counter c_1 is 1, then variable w is assigned a sensor value, and the program counter is incremented. In process b, when $c_2 = 4$, an actuator value is sent through *output(z)*. The faults indicate for example that, at any time, the value of variable x can be randomly changed to one within $[10 \ldots 45]$. Note that we do not consider faults affecting variable z, as per our fault model.

Processes a and b communicate as follows: variable w is written to by process a and read by process b. This defines the information flow between the two processes. An example of a safety specification for program $P1$ is $10 \leq z \leq 50$. This means that whenever the value of variable z is outside of the given range, a safety specification violation occurs. Since a fault cannot cause variable z to take values outside of the permissible range, the action that updates z (i.e., $z := y + x$) should be monitored by a detector to avoid occurrence of bad events that will lead to safety specification violation. For example, starting from a program state $P1s = (w = 15, x = 15, y = 60, z = 10)$ (we exclude the counters), executing the action $z := y + x$, will lead to a program state $P1e = (w = 15, x = 15, y = 60, z = 75)$, which violates the safety specification. Executing the program action starting from state $P1s$ to state $P1e$ is a bad event. Thus, a detector that monitors whether the sum of values of variables x

and y (i.e., $x + y$) is within 10 and 50 is needed. If the sum if outside of the range, then the detector flags an error, and the program can possibly halt.

We will use this example as a running example to explain how our framework works. In the next section, we explain what it means to transform a fault-intolerant program into a fail-safe fault-tolerant one.

4 The Transformation Problem

We now 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 [9, 6].

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 an event e in p' that did not occur in p to satisfy SS, then event e cannot be used by p' to satisfy SS, since this means that there are other ways p' can satisfy SS in the absence of faults. Thus, the set of events occurring in p' should be a subset of the set of events occurred in 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, in the presence of faults, the set of states reachable by p' should be a subset of the set of states reachable by p, and in the absence of faults, the sets of reachable states are equal.
– In the presence of faults, p' satisfies SS.

Overall, the first two conditions state that in the absence of faults, the fault-intolerant program p is "equivalent"[6] to the fail-safe fault-tolerant program p'. Also, in presence of faults, p' satisfies its safety specification, while p does not.

In [6], we showed that composing critical actions of a program with a class of detectors, called *perfect* detectors, is sufficient to solve the transformation problem. In the next section, we will define the concept of SS-globally consistent detectors, and explain that they are instances of perfect detectors. Thus, composing a fault-intolerant program with SS-globally consistent detectors will result in a program that will always satisfy its safety specification in the presence of faults, i.e., it is fail-safe fault-tolerant.

5 Adding Globally Consistent Detectors to a Program

In this section, we will explain the concept of globally consistent detectors. We then explain that a class of globally consistent detectors, called *SS-globally consistent detectors* are instances of perfect detectors.

[6] The two programs exhibit the same behavior in the absence of faults, i.e., are behavior-equivalent

5.1 Consistent Detectors and Globally Consistent Detectors

Before explaining the concept of globally consistent detectors, we will first explain the concept of *consistent detectors*. Recall that a detector d monitors the safe execution of a program action A, such that no bad event actually occurs upon execution of A. The detector d defines a set of states from which execution of A is safe, i.e., execution of A from any state defined by d will not give rise to a bad event.

A detector d_i monitoring a program action A_i is said to be *consistent* with a detector d_j monitoring program action A_j if and only if no sequence of events (thus good events since they have not been ruled out by d_i) starting from execution of A_i will cause execution of A_j to violate the safety specification. In other words, if A_i executes safely, followed by a sequence of good events, such that A_j is executing, the execution of A_j is safe. For example, see process a of Fig. 2.

```
Program $P1'$
    var $x, y$ init 1, $c1$ init 1 : int // process a
process a:
    $c1 = 1 \wedge (15 \leq read() \leq 25) \rightarrow x :=$ read(); $c1 := c1 + 1$; // value of x
between 15 and 25
    $c1 = 2 \wedge (25 \leq x + 10 \leq 35) \rightarrow y := x + 10$; $c1 := c1 + 1$; // loop
    $c1 = 3$                          $\rightarrow output(y); c1 := 1$; //loop
$F$ (faults):
    true $\rightarrow x :=$ random $[10 \ldots 45]$
```

Fig. 2. An example to showthe concept of consistent detectors

In process a, the detector $d_i, (15 \leq read() \leq 25)$, monitors action $A_i, x := read()$, while detector $d_j, (25 \leq x + 10 \leq 35)$, monitors action $A_j, y := x + 10$. If A_i executes, then it means a good event has occurred (i.e., it satisfies d_i). If no fault happens, then A_j will execute as well. Thus, d_i and d_j are consistent. In other words, if x can take a value between 15 and 25, then adding 10 (to obtain value for y) will cause y to take value between 25 and 35, hence consistency of detectors. If a set of n detectors is incorporated in a program, and each detector is consistent with the safety specification, then the set of detectors is said to be *SS-globally consistent*.

5.2 Design of SS-Globally Consistent Detectors

In this section, we introduce a class of detectors, called SS-globally consistent detectors, for which the design complexity is polynomial in the size of the fault-intolerant program, and we will argue that this class of detectors is an instance of perfect detectors, i.e., they are complete (detect all errors that will cause violation of the safety specification of the program) and accurate (no false detection).

The design of *SS-globally consistent detectors*[7], is tractable for a class of programs known as *bounded programs.* The main property of bounded programs is that the length of event sequences before the program outputs a value is bounded, i.e., there are no infinite loops within a process, nor is there some infinite communication between processes, and also that variables take values from finite domains. An example of bounded programs is embedded applications.

A set of SS-globally consistent detectors for a given program p with safety specification SS has the property that each detector in the set is consistent with SS. Recall that SS can effectively be a detector that monitors the critical action of the program. Overall, it means that if a detector d_i monitoring program action A_i is consistent with the safety specification of the program, then no sequence of (good) events, starting from execution of A_i, will violate the safety specification. Hence, d_i is accurate. Also, for any bad event ruled out by SS, there will also be a corresponding event ruled out by d_i. Hence, d_i is complete. Therefore, a globally consistent detector is indeed a perfect detector (accurate and complete).

At this point, we explain how SS-globally consistent detectors can be designed. Since each detector d_i is consistent with the safety specification of the program, we exploit this relationship, and start with the safety specification to automatically generate the SS-globally consistent detectors.

Since SS defines a detector that monitors the critical action of the program, we perform a backward propagation procedure, starting from the critical action of the program, against the flow of information. We illustrate this using a series of examples, see Fig 3–Fig. 5.

Then, for $10 \leq x+y \leq 50$ (safety specification) to be satisfied, and given that variable y is assigned the value of variable w, then the detector that monitors the program action $y := w$ should ensure that $10 \leq w + x \leq 50$. In such case, it is easy to verify that detector $(10 \leq w + x \leq 50)$ is consistent with safety specification $(10 \leq y + x \leq 50)$, see Fig 3.

Likewise, the detector monitoring program action $x := read()$ should ensure that $10 \leq read() + w \leq 50$, such that when variable x is assigned value from *read()*, then this does not violate the detector $10 \leq w + x \leq 50$ monitoring program action $y := w$, see Fig 4.

As for process a, the information flow is from process a to process b, through the shared variable w. The value of variable w is used to update the value of variable y of process b. Thus, if the detector for program action $y := w$ is to be satisfied, then the value of variable w in process a should be cognizant of the fact that variable w can be updated in two different ways. The detector monitoring the *if-then* action of process a is as follows: $(10 \leq w + x - 15 \leq 50) \vee (10 \leq w + x + 5 \leq 50)$, which is equivalent to $(25 \leq w + x \leq 65) \vee (5 \leq w + x \leq 45)$. The program is shown in Fig. 5.

Depending on the fault model, some of those detectors may be excluded. For example, if faults were not to affect variables x and y say, then the detector monitoring program action $z := x + y$ will not be needed, i.e., $10 \leq x + y \leq 50$.

As can be deduced, the complexity of the procedure is polynomial in the size of the program (i.e., polynomial in the state space of the fault-intolerant

[7] Whenever it is obvious from the text, we will use the term globally consistent detectors to mean SS-globally consistent detectors.

```
**Program** $P1$
    **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
    $c2 = 2 \wedge (10 \leq w + x \leq 50) \rightarrow y := w$; $c2 := c2 + 1$;
    $c2 = 3 \wedge (10 \leq y + x \leq 50) \rightarrow z := y + x$; $c2 := c2 + 1$;
    $c2 = 4 \rightarrow$ output($z$); $c2 := 1$; // loop

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

Fig. 3. An Example to show generation of SS-globally consistent detectors. Observe that the critical action $z := y + x$ is monitored by a detector defining SS.

```
**Program** $P1$
    **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:**
    $c2 = 1 \wedge (10 \leq w + read() \leq 50) \rightarrow x :=$ read(); $c2 := c2 + 1$; // value
between 0 and 20
    $c2 = 2 \wedge (10 \leq w + x \leq 50) \rightarrow y := w$; $c2 := c2 + 1$;
    $c2 = 3 \wedge (10 \leq y + x \leq 50) \rightarrow z := y + x$; $c2 := c2 + 1$;
    $c2 = 4 \rightarrow$ output($z$); $c2 := 1$; // loop

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

Fig. 4. An Example to show generation of SS-globally consistent detectors

```
**Program** $P1$
    **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:**
    $c1 = 1 \wedge ((25 \leq x + w \leq 65) \vee (5 \leq x + w \leq 45)) \rightarrow w :=$ read();
$c1 := c1 + 1$; // value between 15 and 25
    $c1 = 2 \wedge x \leq 15 \wedge (5 \leq x + w \leq 45) \rightarrow w := w + 5$; $c1 := 1$; // loop
    $c1 = 2 \wedge x > 15 \wedge (25 \leq x + w \leq 65) \rightarrow w := w - 15$; $c1 := 1$; // loop

**process b:**
    $c2 = 1 \wedge (10 \leq w + read() \leq 50) \rightarrow x :=$ read(); $c2 := c2 + 1$; // value
between 0 and 20
    $c2 = 2 \wedge (10 \leq w + x \leq 50) \rightarrow y := w$; $c2 := c2 + 1$;
    $c2 = 3 \wedge (10 \leq y + x \leq 50) \rightarrow z := y + x$; $c2 := c2 + 1$;
    $c2 = 4 \rightarrow$ output($z$); $c2 := 1$; // loop

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

Fig. 5. The final program with SS-globally consistent detectors

program). As we have explained earlier, SS-globally consistent detectors are instances of perfect detectors, i.e., they detect errors if and only if they lead to violation of the safety specification. Thus, whenever a fault occurs, and given the fact that the detectors are perfect implies that the corresponding error will be detected earlier, i.e., it has a *lower* latency, than if the error is to be detected by the detector guarding the critical action. Specifically, given our fault model and a set D of perfect detectors for program p (resulting in fail-safe fault-tolerant p') with safety specification SS, then a detector $d_i \in D$ exists such that whenever a bad event is about to occur, d_i will flag the problem, i.e., p' has minimal detection latency (i.e., "0-step" – no bad event happens).

Overall, in this section, we have argued that SS-globally consistent detectors are perfect detectors, and we have illustrated, by means of examples, how these are generated. We also argued that when SS-globally consistent detectors are incorporated into a program, the program has a better detection latency.

6 Automatic Generation of Test Cases for Testing/Fault Injection Using Perfect Detectors

In this section, we explain how the use of perfect detectors (i.e., SS-globally consistent detectors) help in the automatic generation of test cases for testing the fail-safe fault-tolerant program, or for fault-injection purposes.

For test case generation, we use the perfect detectors to partition the state (input) space. In [5], the authors argued that for partition testing to be efficient, there is a need to group within one given partition all inputs that will cause the system to fail. The availability of SS-globally consistent detectors, i.e., perfect detectors means that those detectors will partition the input space "perfectly", i.e., one can group into one partition all inputs that will cause the program to fail. For example, the safety specification of our example program is $10 \leq z \leq 50$. If we want to test the program using test cases that will cause the program to fail (e.g., for fault-injection), we should choose test cases from the space $((x + y < 10) \vee (x + y > 50))$ just before executing the critical action.

So, when the resulting fail-safe fault-tolerant program has to be validated (e.g., testing or fault-injection), whenever execution reaches one of of the detectors, an appropriate test case can be automatically generated, and the behavior of the program observed. For example, consider Fig. 5. If the action $y := w$ in process b is about to be executed when the program is running, the detector for this action is $(10 \leq w + x \leq 50)$. So, to generate a test case for fault-injection, we need to invert the detector condition, i.e., $(w + x < 10) \vee (w + x > 50)$, and choose an appropriate value of w that will satisfy the inverted condition.

Similarly, for testing, i.e., testing the fail-safe fault-tolerant program, if the system designer wants to perform unit testing, i.e, testing of each process, the detectors can again be used to help automatically generate the required test cases (assuming all the required stubs are available). For example, considering Fig. 5 again, unit testing of process b will "force" *read()* to generate a value that will violate the detector monitoring this action. For integration testing, i.e., testing communication between processes, the detectors again help in automatically generating test cases. For example, from Fig. 5, for testing the interface between processes a and b, we reuse the detector $(10 \leq w + x \leq 50)$ to generate the relevant test cases.

7 Fault Injection Experiments to Ascertain SS-Global Consistency

We have explained that SS-globally consistent detectors are perfect detectors, i.e., they detect errors if and only if they will lead to violation of the safety specification. We have also shown how to automatically generate the perfect detectors, by means of an example. We also explained how, by exploiting the information obtained from the perfect detectors, test cases for fault-injection or testing can be automatically generated. In this section, we present the results from an experiment to ascertain the viability of the concept of SS-globally consistent detectors.

The target software is an aircraft arresting system, used on short runways, see Fig. 6. It consists of 6 modules. We focus on module *V-REG*. *V-REG* uses the signals *SetValue* and *IsValue* to control *OutValue*, the output value to the pressure valve. The value of the *OutValue* signal is calculated by evaluating a function on the difference between the *SetValue* and the *IsValue* signals. The reason for choosing module *V-REG* is that it is medium-sized, and SS-globally consistent detectors could be easily generated for the module.

To ensure that SS-globally consistent detectors are indeed perfect detectors, we compare them against detectors obtained directly from the specification of the system. The detectors obtained from the specification monitored signals *SetValue* (EA_1) and *IsValue* (EA_2), while the SS-globally consistent detector

Fig. 6. Software Architecture of the Target System

(EA_3) obtained monitored both signals at the same time. EA_4 is the safety specification of the program, monitoring the critical action of the program. EA_3 is the SS-globally consistent detector generated by our approach. Observe that we had obtained a set of SS-globally consistent detectors, however, since we are assuming that errors only get into the system via the input signals of module *V-REG*, we only need to monitor the input signals. If faults could corrupt the value of program variables, we would have included all of the SS-globally consistent detectors in module *V-REG*. We determine SS-global consistency by determining the consistency value of each detector with the safety specification. If the consistency value of all detectors is 1, then all the detectors are SS-globally consistent.

7.1 Fault Injection in V-REG Module

When performing the fault injection experiments, sometimes errors were injected after an aircraft has been arrested. We therefore use the term errors to denote whenever errors are injected before an aircraft has been arrested. The errors injected were bit-flips in the input variables of the module, at different given time instance.

First, we want to ascertain that EA_1 and EA_2 are not SS-globally consistent. Thus, we try to ascertain the fact that there are cases where EA_4 detects an error whereas EA_1 and EA_2 do not, or vice versa.

From the fault injection experiments, we calculated the consistency of a given EA, EA_i, with respect to the safety specification (EA_4) by calculating (i) the number of concurrent error detection by EA_4 and EA_i *conc-det*, and (ii) number of error detection by EA_4, *ss-det*. The consistency values in Table 1 is then calculated as follows: (1 - abs *(ss-det - conc-det)/(ss-det)*). For example, there were 3840 error injections into $SetValue$, and of these, EA_1 detected 1932, giving a detection coverage of 0.50313 for EA_1. Also, there were 2051 corruptions of the system state, leading to violations of the safety specification. Of these, 1561 errors were detected by EA_1. Using the consistency equation above gives a value of 0.76109 for consistency. The same is repeated for the consistency value of EA_2

and EA_3. The consistency value of EA_4 is NA since we assume it to be 1 (by default).

Metrics	EA_1	EA_2	EA_3	EA_4
Consistency	0.76109	0.44369	1	NA

Table 1. Consistency Values of detectors for errors injected in *SetValue*

For data in Table 2, errors injected in $IsValue$, the following values were obtained:

Metrics	EA_1	EA_2	EA_3	EA_4
Consistency	0.082714	0.90441	0.99951	NA

Table 2. Consistency Values of detectors for errors injected in *IsValue*

We also note that the consistency value of EA_3 from Table 2 is less than 1. From closer inspection, we found that this mismatch is due to the fact that sometimes error is detected after the aircraft has been arrested, and when the system is performing some reset action. So, this slight mismatch can be safely ignored, since we did not consider the case when the system is actually resetting. The overall consistency value of each detector is summarized in Table 3.

Overall, we have found that the detector generated by our approach is indeed a perfect detector, since it has consistency value of almost 1 (it is consistent with the safety specification of the system). However, the specification-based detectors EA_1 and EA_2 sometimes allow errors to go undetected, and violate the safety specification, which can have disastrous consequences for safety critical systems, or are inaccurate leading to performance degradation. Also, these detectors also seem to detect errors even though those errors are "harmless" (will not lead to violation of safety specification). These observations corroborate those made by Leveson et. al [10], i.e., specification-based detectors are likely to be inaccurate and/or incomplete. Hence, our approach can be seen as a first step in addressing the problem of generating perfect detectors.

Metrics	EA_1	EA_2	EA_3	EA_4
Consistency	0.42457	0.67224	0.99975	NA

Table 3. Consistency values of detectors for errors injected in V-REG module inputs

8 Discussion and Conclusions

In this paper, we have presented an approach for designing efficient fail-safe fault-tolerant program, i.e., programs with perfect error detection and optimal detection latency. We have explained, through examples, how such detectors (perfect detectors) can be generated. We also explained how, by using the perfect detectors, test cases for validating the fail-safe fault-tolerant program can be automatically obtained. The complexity of our method is polynomial in the size of the program (state space) [6]. Our approach is novel, and to the best of our knowledge, there is no work that has addressed the design of perfect detectors for software. Our work also solves some of the problems posed in [10], and the observations made when running the fault-injection experiments corroborate all the findings presented in [10].

We have shown that SS-globally consistent detectors are instances of perfect detectors, and we have presented an experimental analysis of how SS-global consistency can be verified. Our approach works for a class of programs, known as bounded programs, of which embedded applications are instances.

In this work, we have looked at continuous signals. As future work, we will look at including discrete signals in our framework. An initial possible approach is to partition the space of the continuous signal into disjoint sets of continuous values, where each set can represent one discrete value.

A final note on our approach: note that our approach is not based on inverting the code of the program, rather it makes use of the computation itself to generate detectors. Thus, the problem with having non-invertible functions, such as hash functions, do not apply, and the way our approach deals with such situations is to include the function call inside the detector, e.g., $0 \leq x + F(y) \leq 25$.

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. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, Reading, Mass., 1988.
5. B. Jeng and E.J. Weyuker. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, July 1991.
6. A. Jhumka, F. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast, and perfect detectors. Technical report, Ecole Polytechnique Federale de Lausanne (EPFL), School of Computer and Communication Sciences, Technical Report 200263, September 2002.
7. A. Jhumka, M. Hiller, V. Claesson, and N. Suri. *On Systematic Design of Globally Consistent Executable Assertions in Embedded Software.* In *Proceedings LCTES/SCOPES*, pages 74–83, 2002.
8. S. Kulkarni and A. Ebnenasir. *"Complexity of Adding Fail-Safe Fault Tolerance".* In *Proceedings International Conference on Distributed Computing Systems*, 2002.
9. 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.
10. N. Leveson, S. S. Cha, J. C. Knight, and T. 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.