An Approach for Designing and Assessing Detectors for Dependable Component-Based Systems

Arshad Jhumka[†] Martin Hiller[‡] and Neeraj Suri[†]

[†]Dept of CS, TU-Darmstadt, Germany
Email: {arshad,suri}@informatik.tu-darmstadt.de

[‡]Volvo Tech. Dept, Göteborg, Sweden
Email: martin.hiller@volvo.com

Abstract

In this paper, we present an approach that helps in the design and assessment of detectors. A detector is a program component that asserts the validity of a predicate in a given program state. We first develop a theory of error detection, and identify two main properties of detectors, namely *completeness* and *accuracy*. Given the complexity of designing efficient detectors, we introduce two metrics, namely *completeness* (C) and *inaccuracy* (I), that capture the operational effectiveness of detector operations, and each metric captures one efficiency aspect of the detector. Subsequently, we present an approach for experimentally evaluating these metrics, and is based on fault-injection. The metrics developed in our approach also allow a system designer to perform a cost-benefit analysis for resource allocation when designing effecient detectors for fault-tolerant systems. The applicability of our approach is suited for the design of reliable component-based systems.

Keywords: detectors, fail-safe, formal methods, metrics, cost/benefit analysis, fault injection.

1 Introduction

Safety-critical applications need to satisfy stringent dependability requirements in their provision of services. To reduce the complexity of designing such applications, transformational approaches have been developed in which an initially fault-intolerant program is systematically transformed into a fault-tolerant one. In one such approach, Arora and Kulkarni [3] showed that, to transform a given fault-intolerant program into a *fail-safe fault-tolerant* one, it is both necessary and sufficient to compose the intolerant program with *detectors*[1]. A detector is a program component that detects whether a given predicate holds in a given program state. Examples are self checks, comparators etc.

In this paper we restrict our attention to designing such fail-safe fault-tolerant programs. Intuitively this means that it is acceptable that the program "halts" if faults occur as long as it always remains in a "safe" state. This type of fault-tolerance is useful in application areas such as train control systems where safety (avoidance of catastrophic events) is more important that continuous provision of service in the presence of faults. In fact, in safety-critical systems, fail-safe fault tolerance may be the only desired type of fault tolerance, since in case of imminent failure, the system is shut down in a safe state, and another (possibly mechanical) backup system takes over.

However, the design of efficient detectors is difficult. Leveson *et al.* [10] remarked 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.* [10] concluded that, to design effective detectors, "more training or experience might be helpful". These also indicate that sound methodological approaches or frameworks are needed to guide a programmer in developing effective detectors. However, there is a dearth of frameworks to guide a programmer in designing efficient detectors.

In this paper, we develop a theory that underpins the design of detectors. In our theory we are able to characterize two fundamental aspects of detectors: (1) their *accuracy*, i.e., ability to minimize the number of "mistakes" they make, and (2) their *completeness*, i.e., ability to detect *all* harmful faults, i.e., faults that can threaten the safety of the system. These two properties form the basis of efficient detectors.

The basis of our theory is the notion of a transition which is *inconsistent* with respect to a safety [9] specification. This can be understood as follows: Executing a transition inconsistent w.r.t. the safety specification of a program may lead to a violation of the safety specifi-

[1] Henceforth, we will use the phrases "design of efficient fail-safe fault tolerance" and "design of efficient detectors" interchangeably, depending on the context.

cation if no countermeasures are taken. Building upon this concept, we develop a theory of accurate, complete, and perfect detectors together with the necessary correctness theorems. Intuitively, a detector is *accurate* if it "preserves" all correct behaviors of the system. The accuracy property of a detector is related to the "rate of false alarms" of a detector, i.e., the more accurate a detector is, the less the false alarm rate. A detector is *complete* if it "rejects" all incorrect behaviors of the system in the presence of faults. The completeness property characterizes the ability to detect "harmful" faults, i.e., the more complete a detector is, the more "harmful" faults it detects. A detector is *perfect* if it is accurate and complete. Intuitively, a perfect detector is an efficient detector since it does not make any mistake, as well detecting all harmful errors.

In previous work [7], we had developed an algorithm that automatically generates perfect detectors for a given fault-intolerant program. However, the algorithm assumes implementation knowledge. There are many cases where such an assumption breaks down, e.g., component-based systems, OO systems. For these systems, the algorithm in [7] may not readily apply.

To circumvent this problem, based on the identified efficiency properties of a detector, we present two metrics, namely *completeness* and *inaccuracy* of the detector that capture its effectiveness. The idea is to use these metrics as guide in refining the detectors so as to "emulate" perfect detectors. However, without these metrics, "emulating" perfect detectors would still be hard, since the refinement process will be adhoc. Hence, these metrics help in the refinement of the detectors, and also help in their assessment.

In this paper, we make the following contributions:

- We explain what it means to transform a fault-intolerant program into a fail-safe fault-tolerant one, and we present a novel theory of detectors, and identify a class of detectors called perfect detectors that solves the transformation problem.
- We argue that, for component-based systems, it may be difficult to design perfect detectors.
- We propose two metrics, called *completeness*(C), and *inaccuracy* (I), that help in designing and assessing detector "perfectness".
- We explain how these metrics can be experimentally evaluated, and present a small case study on a real software used for aircraft arrestment in aircraft carriers (short runways), and we discuss their relevance in design of efficient detectors.

Intuitively, the completeness metric (C) captures the completeness property of a detector, i.e., it captures the fraction of harmful errors detected. The inaccuracy metric (I) captures the amount of mistakes (false detections) a detector makes. Thus, the composite metric (C,I) characterizes how "far" the detector being designed is from being perfect.

Our approach works for a class of programs called *bounded programs*. The property of such programs is that the set of reachable states is finite (bounded). Typically, embedded programs are bounded programs.

The paper is structured as follows: Sect. 2 introduces the formal notations used in the paper. In Sect. 3, we explain the addition of fail-safe fault tolerance, and present a theory of detectors. We introduce the metrics in Sect. 4. In Sect. 5, we present a target system used for our experimental measurement of these metrics, which are detailed in Sect. 6. We summarize and conclude in Sect. 7.

2 Preliminaries

In this section, we recall the standard definitions of programs, faults, fault tolerance (in particular fail-safe fault tolerance), and of specifications [3].

2.1 Programs

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

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

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

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

An *action* ac *of* p *is enabled in a state* s if the guard of ac evaluates to "true" in s. An action ac can be 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 as multiple actions can be enabled in the same state. In particular, each non-deterministic action can be converted into a set of deterministic actions with an identical state transition relation. We also

assume that each non-deterministic action has only finite non-determinism.

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* implies that if a program action *ac* is continuously enabled, *ac* is eventually chosen to be executed. Weak fairness implies that a computation is *maximal* with respect to program actions, i.e., if the computation is finite then no program action is enabled in the final state.

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

In the context of this paper, programs are equivalently represented as state machines, i.e., a program is a tuple $p = (S_p, I_p, \delta_p)$ where S_p is the state space and $I_p \subseteq S_p$ is the set of initial states. Transition $(s, s') \in \delta_p$ iff *ac* is enabled in state s and computation of *ac* results in state s'. We say that *ac induces* these transitions.

We assume programs to contain a set of *critical actions* (defined in Sec. 3.4), and such actions control program progress.

2.2 Specifications

A *specification* for a program p is a set of computations which is fusion-closed. A *specification S is fusion-closed*[2] 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.

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

Definition 2 (Safety specification) *A specification S of a program p is a* safety specification *iff the following condition holds: $\forall$ computation σ that violates S, $\exists$ a prefix α of σ s.t $\forall$ state sequences β, $\alpha \cdot \beta$ violates S.*

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

[2]Intuitively, fusion closure guarantees that history is available in each computation state.

Informally, the safety specification of a program states that "something bad never happens". More formally, it defines a set of "bad" finite computation prefixes that should not be found in any computation. Alpern and Schneider [2] have shown that every specification can be written as the intersection of a safety specification and a *liveness specification.* Informally, a liveness specification determines what types of events must eventually happen. As our interest is in safety specifications we omit the formal definition of liveness. However, liveness issues are important since any safety specification can be satisfied by the empty program, i.e., the program that does nothing.

2.3 Fault Models and Fault Tolerance

All standard fault models from practice which endanger a safety specification (transient or permanent faults) can be modeled as a set of added transitions. We focus on the subset of these fault models which can potentially be tolerated: We disallow faults to violate the safety specification directly. For example, for embedded applications, if a safety specification constrains the output variables of a program, the fault model prevents the *fault actions* to modify the output variables in such way that safety is violated. However, fault actions can change the program state such that subsequent program actions execution violate the safety specification. The reason for choosing such a fault model is that we target tolerable faults. If a fault can directly violate safety, then no fail-safe fault-tolerant program exists (since we cannot prevent the fault from occurring).

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

Definition 4 (Computation during 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*

Weakly *p*-fair means that only the actions of *p* are treated weakly fair (fault actions must not eventually occur if they are continuously enabled). We say that a *fault occurs* if a fault action is executed. Rephrased in the transition system view, a fault model adds a set of transitions to the transition relation of *p*. We denote the modified transition relation by δ_p^F. Since fault actions are not treated fairly, their occurrence is not mandatory. Note that we do not rule out faults that occur infinitely

often (as long as they do not directly violate the safety property).

Definition 5 (Fail-safe fault-tolerance) *Given a program p with safety specification SS, and a fault model F. The program p is said to be* fail-safe F-tolerant for SS *iff all computations of p in the presence of faults F satisfy SS.*

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

Consider the transition system view of a program p.

Definition 6 (Reachable state) *We say that a* state s is reachable by p *iff starting from an initial state of p it is possible to construct a computation α in which s occurs in α, using only transitions from δ_p. Otherwise s is unreachable.*

Definition 7 (Reachable transition) *A transition (s,t) of p is reachable iff state s is reachable by p. Otherwise it is unreachable.*

Similar definitions for *reachability in the presence of faults* can be easily obtained by replacing p with δ_p^F.

2.4 Issues of Implementation Non-Availability

When implementation details are not available, like in component-based systems, this implies the inability to read/write certain variables. The issues involved under such read/write constraints have been discussed in [8]. For completeness, we recall them here.

Write Restrictions: Given a transition (s_0, s_1), the variables that need to be written to for the transition to occur can be determined. Write restrictions amount to ensuring that transitions of a process p_i only modify variables p_i can write to. Specifically, if a process p_i can only modify the values of variables contained in a set W_{p_i}, and the value of a variable $v \notin W_{p_i}$ is modified by a transition (s_0, s_1) of process p_i, then p_i cannot use this transition. In effect, the transitions that a process p_i cannot use are those that update any variable $v \notin W_{p_i}$, i.e., all transitions
$\tau \in \{(s_0, s_1) : (\exists v : v \notin W_{p_i} : v(s_0) \neq v(s_1)\}$[3].

Read Restrictions: Given a transition (s_0, s_1), it appears that all variables need to be read for that transition to be executed. In such a case, read restrictions result in grouping transitions together. Specifically, the transitions are grouped in such a way that "reading" the variables that cannot be read is irrelevant. To see this, consider the following example: There are two possible transitions t_1 and t_2, of process p_i, with the following read restrictions - p_i can read variable x, but not y (both having domains $\{0,1\}$). Let $t_1 = (x = 0, y = 0) \rightarrow (x = 1, y = 0)$, and $t_1 = (x = 0, y = 1) \rightarrow (x = 1, y = 1)$. If only one transition is to be included, then p_i needs to be able to read variable y also to be able to make this decision. On the other hand, if both transitions can be grouped and included together, reading variable y is irrelevant, which then becomes consistent with the read restriction imposed on that variable. Thus, given a process p_i, together with its set of readable variables R_{p_i}, the group defined by a given transition $()s_0, s_1$ is defined as:
$\text{group}(p_i, R_{p_i})(s_0, s_1) = \{(s_0', s_1') : (\forall v \in R_{p_i} : v(s_0) = v(s_0') \wedge v(s_1) = v(s_1')) \wedge (\forall v \notin R_{p_i} : v(s_0) = v(s_1) \wedge v(s_0') = v(s_1'))\}$

Lack of Implementation Knowledge: The lack of implementation details for the design of detectors is in a sense equivalent to defining constraints on variables reads/writes. In those cases, detectors are usually only defined to monitor actions that update interface variables, specifically those that define the observable behavior of the system. Such detectors are usually referred to as *specification-based* detectors.

Subsequently, we do not distinguish between read/write restrictions and lack of implementation knowledge since all of them can be similarly modeled.

3 Addition of Fail-Safe Fault Tolerance

In this section, we explain the addition of fail-safe fault tolerance to a fault-intolerant program. We first review the role of detectors in ensuring fail-safe fault tolerance.

3.1 Role of Detectors in Fail-Safe Fault Tolerance

Informally, a detector[4] is a program component that detects whether a given predicate is true in a given state. Arora and Kulkarni showed in [3] that, for every action ac of a program p with safety specification SS, there exists a predicate such that execution of ac in a state where this predicate is true satisfies SS. If a transition (s, s') induced by ac violates SS, then we call such a transition a *bad transition*. Thus, any computation that violates SS contains a bad transition.

Proposition 2 *Let SS be a safety specification, p an F-intolerant program for SS. If p violates SS then there*

[3] $v(s)$ denotes the value of variable v in state s

[4] For a more formal introduction, we refer the reader to [3].

exists a transition $t \in \delta_p$ such that for all computations σ of p holds: If t occurs in σ then $\sigma \notin SS$.

Given a program p with safety specification SS and specification S expressed as a temporal logic formula, the set of bad transitions can be computed in polynomial time by considering all transitions (s, s') where $s, s' \in S_p$. For simplicity, we assume that the safety specification is concisely expressed as a set of bad transitions. The authors of [3] also show that fail-safe fault-tolerant programs contain detectors. However, [3] *does not* provide information pertaining to the properties the detectors need to possess, viz. completeness and accuracy, so as to ensure fail-safe fault tolerance, which is one goal of this paper.

Before presenting the theory of detectors, we first explain the transformation problem for addition of fail-safe fault tolerance.

3.2 Transformation Problem for Addition of Fail-Safe Fault Tolerance

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

C.1 If there exists a transition (s, t) in p' that is not used by p to satisfy SS, then (s, t) cannot be used by p', since this means that there are other ways p' can satisfy SS in the absence of faults. Thus, the set of transitions of p' should be a subset of the set of transitions of p.

C.2 If there exists a state s reachable by p' in the absence of faults that is not reached by p in the absence of faults, then this means that p' can satisfy SS differently from p in the absence of faults, and such a state s should not be reached by p' in the absence of faults. Thus, the set of states reachable by p' should be a subset of the set of states reachable by p.

In general, these conditions C.1 and C.2 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) *Let SS be a safety specification, F a fault model, and p an F-intolerant program for SS. The fail-safe transformation problem is defined as follows: Identify a program p' such that the following three conditions hold:*

1. *p' satisfies SS 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'.*

A program p' that satisfies the above conditions is said to solve the fail-safe transformation problem for p.

In the next section, we present a theory of detectors, based upon which, we provide an algorithm that synthesizes a program p' from a fault-intolerant program p, such that p' solves the fail-safe transformation problem.

3.3 A Theory of Perfect Detectors

The theory is based on the concept of SS-inconsistency, where SS is the safety specification of a program p. The intuition behind the definition of inconsistency is that if a given computation of p in the presence of faults violates the safety specification, then some "erroneous" transition has occurred in the computation.

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

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

Fig. 1 illustrates 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 SS 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 SS. In this computation, the three transitions following the fault transition match Definition 9 and hence are SS-inconsistent w.r.t. α_1 in the presence of F. Note that an SS-inconsistent transition is only reachable in the presence of faults.

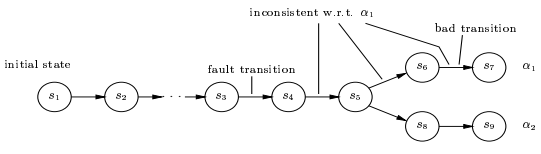

Figure 1: Graphical explanation of SS-consistency.

Intuitively, an SS-inconsistent transition for a given program computation is a program transition where the

subsequent execution of a sequence of program transitions can cause the computation to violate the safety specification, i.e., *SS*-inconsistent transitions lead the program computation on the "wrong path".

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

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

In general, due to non-determinism in program execution, a transition can be *SS*-inconsistent w.r.t. a computation α_1, and not be *SS*-inconsistent w.r.t. α_2. In Fig. 1, (s_4, s_5) is not SS-inconsistent w.r.t. α_2. If we cannot find a computation in the presence of faults for which a particular transition is *SS*-inconsistent then we say that this transition is *SS*-consistent.

Definition 11 (*SS*-consistent transition for p) *Given a program p with safety specification SS. A* transition (s, s') is *SS*-consistent for *p iff (s, s') is not SS-inconsistent for p.*

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

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

Proof sketch: Since the computation violates the safety specification, it contains a bad transition, which is itself *SS*-inconsistent.

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

3.4 Perfect Detectors

From Sec. 3.3, we observed that *SS*-inconsistent transitions are those transitions that can lead a program to violate its safety specification in the presence of faults if no precautions are taken. Perfect detectors are a means to implement these precautions. The definition of perfect detectors follows two guidelines: A detector d monitoring a given action ac of program p needs to (1) "reject" the starting states of all transitions induced by ac that are *SS*-inconsistent for p, and (2) "keep" the starting states of all induced transitions that are *SS*-consistent for p. These two properties are captured in the definition of *completeness* and *accuracy* of detectors (the notions are defined in analogy to Chandra and Toueg [4]).

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

Definition 13 (Detector accuracy) *Given a program p with safety specification SS, fault class F, and a program action ac of p. A* detector d monitoring ac is *SS*-accurate for ac in p in presence of F *iff for all transitions (s, s') induced by ac holds: if (s, s') is SS-consistent for p, then $s \in d$.*

Definition 14 (Perfect detector) *Given a program p with safety specification SS, fault class F, and a program action ac of p. A* detector d monitoring ac is *SS*-perfect for ac in p in presence of F *iff d is both SS-complete and SS-accurate for ac in p.*

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

Intuitively, the completeness property of a detector is related to the safety property of the program p in the sense that the detector should filter out all *SS*-inconsistent transitions for p induced by the action it is monitoring, whereas the accuracy property relates to the liveness specification of p in the sense that the detector should not rule out *SS*-consistent transitions. This intuition is captured by the following lemmas. Lemma 1 uses the accuracy property to show that the fault free behavior of a program is not affected by adding perfect detectors. 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 Sketch: Since the detectors are perfect, they reject all *SS*-inconsistent transitions, which are only reachable in presence of faults. In the absence of faults, they are not reachable. Hence, p and p' have the same computations in absence of faults.

Before we characterize the role of perfect detectors in presence of faults, we formally define *critical actions* of a program. A critical action is one which causes violation of safety when executed in an erroneous state.

Definition 15 (Critical and non-critical actions) *Given a program p with safety specification SS, and fault class F. An action ac of p is said to be* critical *iff there exists a transition (s, s') induced by ac such that (s, s') is a bad transition (Proposition 2) that is reachable (Definition 7) in presence of faults F. An action is non-critical iff it is not critical.*

Lemma 2 (Behavior in the presence of faults) *Given a fault-intolerant program p with safety specification SS, and fault class F. Given also a program p' by composing each critical action ac of p with a perfect detector for ac in presence of F. Then, p' satisfies SS in presence of faults F.*

Proof sketch: Since all critical actions are composed with perfect detectors, all bad transitions are "rejected", hence safety cannot be violated. The resulting program is thus fail-safe.

Hence, from Lemmas 1 and 2, we observe that a program p' obtained by composing each critical action ac of a fault-intolerant program p with a perfect detector for ac in p in presence of faults F solves the transformation problem.

Thus, we have shown that the composition of critical actions of a fault-intolerant program with perfect detectors, which are guaranteed to exist, is crucial to solve the fail-safe transformation problem.

3.5 Algorithm for Adding Perfect Fail-Safe Fault Tolerance:

Having established the role of perfect detectors in fail-safe fault tolerance, in Fig. 2, we provide the algorithm developed in [7] that solves the fail-safe transformation problem, using perfect detectors. It takes as arguments the program p, the fault class F, and the set of ss of bad program transitions encoding the safety specification.

add-perfect-fail-safe(δ_p, δ_F, ss: set of transitions):
{ ss_r := get-ssr(δ_p, δ_F, ss)
return ($p' = \delta_p \setminus ss_r$)}

get-ssr(δ_p, δ_F, ss: set of transitions):
{ ss_r := $\{(s,t) | (s,t)$ is induced by a critical action of p and is SS-inconsistent for p and (s,t) is reachable using transitions in $\delta_p^F\}$
return (ss_r)}

Figure 2: Algorithm that solves the fail-safe transformation problem

Theorem 1 *Algorithm* add-perfect-fail-safe *solves the fail-safe transformation problem.*

The algorithm assumes that implementation knowledge is available, i.e., all variables are readable/writeable. However, this assumption may not be readily met in the case of component-based designs. Hence, the above algorithm may not be readily used to generate perfect detectors for component-based systems. This is because, as explained in Sect. 2.4, some transitions may have to be included (resp. excluded) even though they are SS-inconsistent (resp. SS-consistent).

4 Completeness and Inaccuracy Metrics

The algorithm presented in Fig. 2 may not be easily used to generate perfect detectors for component-based systems, or any other programs where implementation details are not available. One goal of the paper is to develop an approach on how to develop such detectors for such systems.

Perfect detectors have two main properties: (i) completeness, and (ii) accuracy. We propose two metrics, termed as Completeness (C) and Inaccuracy (I), and use both as a composite metric (C,I). The completeness metric (C), as the name suggests, encapsulates the completeness property of a detector, while the inaccuracy metric (I) encapsulates the accuracy property.

Intuitively, the C metric of a detector monitoring a program ac in program p with safety specification SS tries to capture the ratio of SS-inconsistent transitions induced by ac that are rejected by the detector. On the other hand, the I metric of a detector monitoring a program ac in program p with safety specification SS tries to capture the ratio of SS-consistent transitions induced by ac that are rejected by the detector. Thus, the metrics are defined as follows:

1. C - Completeness of detector d_i = (number of concurrent detection by d_i and safety violation)/(number of safety violations)
2. I - Inaccuracy of detector d_i = (number of detections by d_i and no safety violation)/(number of computations of P in presence of faults that do not violate SS)

Note: The denominator of each metric can never be equal to 0, leading to the metric to have value ∞. For the C metric, the denominator cannot have value 0, since the program is assumed to violate SS in presence of faults (program is fault-intolerant). For the I metric, if the denominator is 0, then it means the program is fault-tolerant in itself, which violates our assumption of having an initially fault-intolerant program.

Also, a perfect detector d_i monitoring an action ac in program p with safety specification SS will have value (C = 1, I = 0). A detector that is not perfect will have value $(c, i) : 0 \leq c < 1, 0 < i \leq 1$. Overall, our composite metric (C,I) achieves two main goals:

1. It helps in the *design of perfect detectors* in component-based systems. For example, if a detector has value (0.7, 0), then the system designer knows that the detector is not complete. The system designer knows that the detector is not restrictive enough, in the sense that it does not reject all SS-inconsistent transitions induced by the action it is monitoring. Thus, the designer knows that he needs to "tighten" the detector, so that the completeness reaches 1. This would not have been possible without these metrics, as the fine-tuning process of detectors would have been ad-hoc.

2. It helps in the *allocation of resources* in the design of perfect detectors. For example, assume that there are two detectors d_i and d_j in program p monitoring actions a_i and a_j respectively. Assume that d_i has value (0.7, 0), and d_j has value (0.85, 0.15). Depending on project policies, and the nature of the application, the system designer may decide to invest more time to either refine d_i or d_j. For example, if the application is safety-critical in nature, it may make more sense to refine d_j since its completeness factor is closer to 1.

5 Experimental Validation of Detectors

In this section, we describe an experimental approach used to evaluate the composite metric introduced. In this experiment, we used specification-based detectors 2.4, and assess their completeness/accuracy, i.e., assess their perfectness. Recall that a specification-based detector is usually designed over interface variables only, i.e., there are read restrictions over the program's variables. In our experiment, we found that these specification-based detectors not to be perfect, which *corroborates* findings detailed in [10].

5.1 Example Target System: Aircraft Arresting System

To illustrate our approach and the problems with specification based detectors, we make use of an example system, specifically an embedded control system for arresting aircraft (similar to the cable-and-hook systems found on, e.g., aircraft carriers).

The target system is developed according to the specifications in [1]. It consists of two rotating drums, one on each side of the runway, and a cable is strapped across the runway. Incoming aircraft use a hook to grab hold of the cable. The system detects movement on the rotating drums and will try to slow them down by applying a braking pressure. This will eventually pull the aircraft to a complete stop. The structure of the software is illustrated in Fig. 3.

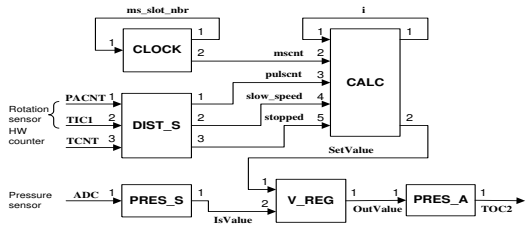

Figure 3: Software structure of the example system.

The modules are as follows:

CLOCK provides a millisecond-clock, *mscnt*. The system operates in seven 1-ms-slots. In each slot, one or more modules (except for CALC) are invoked. The signal *ms_slot_nbr* tells the module scheduler the current execution slot. Period = 1 ms.

DIST_S receives *PACNT* and *TIC1* from the rotation sensor and *TCNT* from the hardware counter modules. The rotation sensor reads the number of pulses generated by a tooth wheel on the drum. The module provides a total count of the pulses, *pulscnt*, generated during the arrestment. It also provides two boolean values, *slow_speed* and *stopped*, i.e., if the velocity is below a certain threshold or if the aircraft has stopped. Period = 1 ms.

CALC uses *mscnt*, *pulscnt*, *slow_speed* and *stopped* to calculate a set point value for the pressure valves, *SetValue*, at six predefined checkpoints along the runway. The checkpoints are detected by comparing the current *pulscnt* with predefined *pulscnt*-values corresponding to the various checkpoints. The current checkpoint is stored in *i*. Period = n/a (background task, runs when other modules are dormant).

PRES_S reads the the pressure that is actually being applied by the pressure valves, using *ADC* from the internal A/D-converter. This value is provided in *IsValue*. Period = 7 ms.

V_REG uses *SetValue* and *IsValue* to control *OutValue*, the output value to the pressure valve. *OutValue* is based on *SetValue* and then modified to compensate for the difference between *SetValue* and *IsValue*. This module contains a PID-regulator implemented in sofwtare. Period = 7 ms.

PRES_A uses *OutValue* to set the pressure valve via the hardware register *TOC2*. Period = 7 ms.

In the next section we will show experimental measurements performed on the example system, illustrating the problems associated with achieving perfect detectors with specification-based detectors.

6 Experimentally Ascertaining Detector Perfectness

Using the specification of the system [1] and data sheets for the sensors and actuators and the representations of internal data, a set of detectors were inserted into the software of the system. The detectors used here are so called executable assertions (EA) and each mechanism monitors one individual signal. In this example we will concentrate on the V_REG module. Here we added detectors to monitor the signals *SetValue* and *IsValue* (EA1 and EA2, respectively). We also have a detector at the output signal *OutValue* (EA7) acting as the safety specification of the system. Thus, if EA7 detects an error, we consider this a violation of the system's safety specification.

To experimentally assess the consistency of the detectors, we performed fault injection experiments using PROPANE [5] to perturb the data with bit-flips in *SetValue* and *IsValue*. Multiple injection runs were conducted, and in each run, one bit in one signal was flipped. When analysing the results, the detectors in an injection run were said to be consistent if the detector at *SetValue* or *IsValue* (depending on where the error was inserted) and the detector at *OutValue* exhibited the same profile and detection coverage. That is, the detectors were considered to exhibit consistency if they detected the same set of errors.

The signals *SetValue* and *IsValue* are both 16 bits wide. We injected single bit-flips into each bit position at 10 different instances in time and for different 25 test cases (i.e., landing aircraft). In each injection run, only one bit-flip was performed, i.e., we had no multiple errors, neither in location nor in time. Thus, for each signal we injected a total of 16·10·25=4,000 errors. However, some of these errors were injected into the system after an aircraft was completely arrested (stopped). These *inactive* errors were diregarded during the analysis of the results.

Generating Value for Completeness and Inaccuracy: For the signal *SetValue*, we had 3,840 active errors (n_{inj}). The safety specification was violated (i.e., EA7 detected an error) for 2,051 of these (n_{vss}). The mechanism monitoring *SetValue*, EA1, detected a total of 1,932 errors (n_{ea1}). However, only 1,561 of these were coincident with the violation of the safety specification ($n_{ea1,vss}$). Using the expressions for the completeness and inaccuracy (see Sect. 4), we can calculate the metrics as follows: i) completeness of EA1 = $\frac{n_{ea1,vss}}{n_{vss}}$ = 0.7611, ii) inaccuracy of EA1 = $\frac{n_{ea1}-n_{ea1,vss}}{n_{inj}-n_{vss}}$ = 0.2074. Thus, the metric for EA1 is (0.7611, 0.2074).

For the signal *IsValue*, we had 3,840 active errors (n_{inj}). The safety specification was violated (i.e., EA7 detected an error) for 2,083 of these (n_{vss}). The mechanism monitoring *IsValue*, EA2, detected a total of 2,146 errors (n_{ea2}). However, only 1,890 of these were coincident with the violation of the safety specification ($n_{ea2,vss}$). Using the expressions for the completeness and inaccuracy (see Sect. 4), we can calculate the metrics as follows: i) completeness of EA2 = $\frac{n_{ea2,vss}}{n_{vss}}$ = 0.9074, ii) inaccuracy of EA2 = $\frac{n_{ea2}-n_{ea2,vss}}{n_{inj}-n_{vss}}$ = 0.1457. Thus, the metric for EA2 is (0.9074, 0.1457).

As can be seen from the results, the specification based detectors in V_REG are not perfect for the actions they monitor. This means that, in the absence of faults, liveness can be compromised, decreasing the efficiency of the system (because of the inaccuracy of the detectors). Also, there are cases where safety can be seriously compromised, whenever the completeness metric is less than 1. In such cases, much effort should be spent on trying to increase the completeness metric to 1.

Clearly, having perfect detectors will ensure that the error handling mechanisms of a system will only be invoked if they are really necessary, i.e., when an error exists which can lead to violation of the safety specification of the system, while also guaranteeing that the program will satisfy its liveness in absence of faults.

However, to obtain perfect detectors, the algorithm presented in Fig. 2 requires access to all variables of the program (implementation knowledge). This means that the resulting detectors require information on the entire state of the system. Specification-based detectors, on the other hand, are local predicates requiring only local information. Thus, the idea is to use local predicates, such as specification-based detectors, and using the composite metric as guide to fine tune the local predicates to "emulate" perfect detectors.

6.1 Interpreting (C,I) Values

For the experiment, we obtained the following (C,I) values for each of these specification-based detectors:

1. EA1 = (0.7611, 0.2074)
2. EA2 = (0.9074, 0.1457)

From the metrics above, we conclude that none of the specification-based detectors (EA1 and EA2) are perfect. What this means is that, in the absence of faults, liveness of the system can be compromised [7]. This can be deduced from the inaccuracy metric as it captures the accuracy property of detectors. For example, for EA1, it means that 20% of the cases when safety will not be violated, EA1 detects an error, i.e., in 20% of the cases where there is no harmful error in the system, EA1 flags something as harmful. Hence, the rate

of false alarms is around 20%. Formally, it means that EA1 wrongly rejects 20% of SS-consistent transitions of the program action it is monitoring.

The completeness metric, on the other hand, conveys information pertaining to the completeness property of the detector. For example. EA1 has completeness metric 0.7611. This means that in almost 25% of cases where safety can potentially be violated, EA1 does not detect anything erroneous. Formally, this means that EA1 fails to reject 25% of the SS-inconsistent transitions induced by the program action it is monitoring. This severely compromises safety, the more so in safety-critical systems.

Hence, if a program being designed is for safety critical systems, then more efforts need to be focused to increase the completeness factor to 1. Once this is achieved, if sufficient resources (time, human etc) are available, the inaccuracy metric can be dealt with, decreasing it to 0. On the other hand, if the program being designed is not safety-critical in nature, and requires good performance in the absence of faults (such as multimedia applications), then perhaps more resources should be invested to decrease the inaccuracy value to 0. In such cases, it would mean liveness is preserved when there is no harmful fault in the system. Hence, this composite metric allows performance and safety to be traded off against each other, depending on the nature of the system to be designed.

Also, the results obtained are consistent with an observation mad by Leveson *et al.* [10], where they observed that specification-based detectors are not very effective at detecting errors, or that they detect errors when there is none in the system. Though the authors of [10] did not specify what properties the detectors are lacking, the research presented in this paper shows that specification-based detectors are not usually perfect, i.e., not complete and/or accurate.

7 Discussion and Summary

In this paper, we have presented a theory of perfect detectors, and identified two main properties of detectors, namely completeness and accuracy, that underpin their effectiveness. We have presented an algorithm that generates perfect detectors, based on the assumption of implementation details. However, there are several instances where such assumptions are not met, for example, in component-based designs, OO programs etc. To circumvent this problem, we have proposed a composite metric (C,I) (C - Completeness, and I - Inaccuracy) that captures the degree of perfectness of a detector. We have also presented an evaluation approach, based on fault-injection, for evaluating these metrics. Our approach works well for a class of programs called *bounded programs*, of which embedded applications are instances. The property of bounded programs is that the set of reachable states in finite (bounded). One possible limitation of the approach is that the values obtained are only experimental and not analytical, meaning that those values are not the real-world accurate values of the detectors.

Kulkarni and Ebnenasir [8], and Jhumka *et al.* [6] showed that design of efficient fail-safe fault tolerance is NP-hard where read/write constraints are imposed. Kulkarni and Ebnenasir considered a subclass of programs where addition of fail-safe fault tolerance can be achieved in polynomial time. As way of contrast, the approach presented in this paper has no such restriction, but however is a heuristic approach in the design of perfect detectors for component-based systems.

As future work, we are looking into including the evaluation of such metrics for detectors as an options in the fault-injection tool Propane [5], such that when FI experiments are conducted, such composite metric is automatically generated.

References

[1] US AirForce 99. *"MIL - SPEC: Aircraft Arresting System BAK-12A/E32A; Rotary Friction."* MIL-A-38202C- Notice 1, US Dept. of Defence, Sept 86.

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

[3] Anish Arora and Sandeep S. Kulkarni. *"Detectors and correctors: A theory of fault-tolerance components."* In *Proc. ICDCS'98*, May 1998.

[4] T.D. Chandra, S. Toueg *"Unreliable failure detectors for reliable distributed systems" "Journal of the ACM, 43(2), pp. 225–267, 1996"*

[5] M. Hiller, A. Jhumka, and N. Suri. *"PROPANE: an environment for examining the propagation of errors in software"* ISSTA 2002: 81-85

[6] A. Jhumka, M. Hiller, V. Claesson and N. Suri. *"On Systematic Design of Globally Consistent Executable Assertions in Embedded Software"* Proc. LCTES/SCOPES, pp. 74-83, 2002

[7] A. Jhumka, F. Gärtner, C. Fetzer, and N. Suri. *"On systematic design of fast, and perfect detectors."* EPFL-TR 200263, September 2002.

[8] S. Kulkarni, A. Ebnenasir. *"The Complexity of Adding Failsafe Fault-Tolerance"* Proc. ICDCS 2002, 337–344, 2002.

[9] L. Lamport. *"Proving the Correctness of Multiprocess Programs"* IEEE Trans. on Soft. Eng., 2, 125-143, March 1977

[10] N. Leveson *et al.* *"The Use of Self Checks and Voting in Software Error Detection: An Empirical Study"* IEEE Trans. on Software Engineering, 16(4), April 1990