Arshad Jhumka
Martin Hiller, Neeraj Suri

TU – Darmstadt, Germany

Context

- Safety-critical software
- Design of fault-tolerant software known to be difficult, and very often ad-hoc.
- Validation is expensive – running of a lot of experiments.
- May still end up with ``inefficient´´ software, e.g., false alarms, late error detection.

- Title: A Framework for the Design and Validation of Efficient Fail-Safe Fault-Tolerant Software

- Presentation outline:
  - Background
  - Problems and Objectives
  - Design of efficient fail-safe fault-tolerant SW
  - Test case generation for validation of efficient fail-safe fault-tolerant SW
  - Summary

Background (1)

- Fault: An unexpected event, e.g., node crashes, variable corruptions. Each one is a fault class.

- Fault-tolerant program: Satisfies some form of correctness in presence of faults.

- Different levels of fault tolerance
  - Masking fault tolerance (ideal)
  - Fail-safe fault tolerance

Background (2)

- Correctness: Specification
- Safety: ``always...´´
  - mutual exclusion, always (output > 100)
- Liveness: ``eventually...´´ -- Termination
- A fail-safe fault-tolerant program always satisfies its safety specification in presence of faults. Ok to just stop.

Fail-Safe Fault Tolerance

- Detection is important in fault tolerance

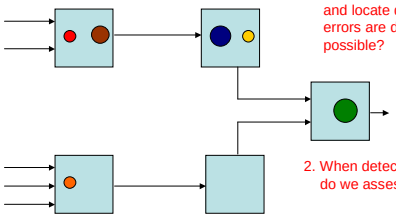
- Detector – A program component that checks the validity of a predicate, e.g, assertion checks, comparator.

- Arora & Kulkarni, 1998 – detectors are both necessary and sufficient to ensure fail-safeness

Assumptions

- Bounded programs – finite number of states, e.g., embedded programs. Can be achieved via proper subtyping.
- Logically (and physically) distributed software.
- Source code available.

General Problems



1. How do we systematically design and locate detectors such that the errors are detected as early as possible?
2. When detectors are distributed, how do we assess their consistency?
3. How can we systematically generate test cases to validate our design?

Goals

- Fail-safe fault-tolerant program able to

v Detect all harmful errors
v No false alarms
v Detect errors early

- Test cases for validation of fail-safe fault-tolerant program

Transformational Approach

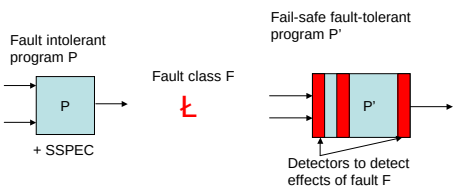
- Fault-intolerant program P (viewed as a state machine)
- Safety specification SSPEC
- Fault class F

Ł Obtain fail-safe F-tolerant program P'

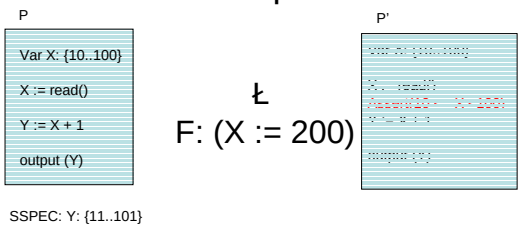
P' always satisfy SSPEC in presence of F +
P' has minimal detection latency for F

Graphical Illustration

- Transform fault-intolerant program P into a fail-safe fault-tolerant program P', with minimal detection latency.



Example



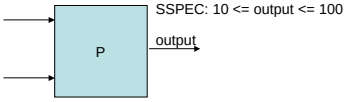
Some Advantages

- Separation of concern between design for functionality (P) and for fault tolerance (P')
- Modular – different fault classes can be considered

Detector Role

- Harmful event: e.g., output > 100.
- Safety specification: defines a set of harmful events.
- Prevent the occurrence of harmful events.

Safety Specification



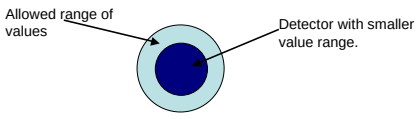
Example of a bad event: Any program transition that allows output to violate SSPEC, (<output = 90>, <output = 110>)

Formal Design Approach

- Given:
  Program P, safety specification SSPEC, and fault class F.
- Goal: Compose P with a set of detectors D such that P' = P[]D (i) is fail-safe F-tolerant, and (ii) has minimal detection latency for F.

Detector Design (1)

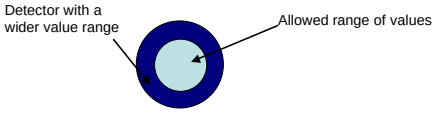
- A detector can be too strong – it filters out harmless events.



- Leads to false alarms!

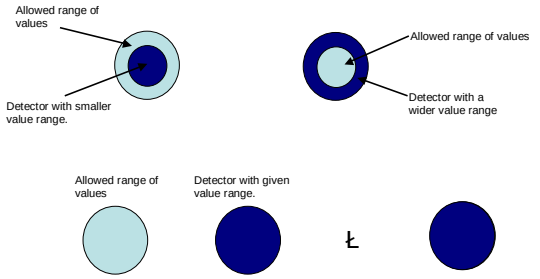
Detector Design (2)

- A detector can be too weak – it does not filter out all harmful events.



- Can have catastrophic consequences!

Perfect Detectors



Detector Design (3)

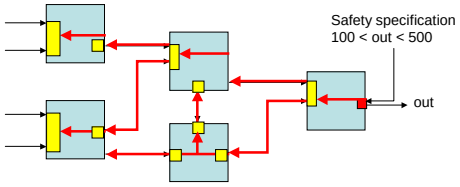
- We want a detector to (i) detect all harmful events, (ii) have no false positives.
- Such a detector is perfect.
- Thus, we need a set of perfect detectors D.

Ł We compose P with a set D of perfect detectors, yielding P'.

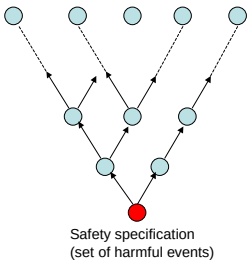
Detector Design (4)

- Given program P, its safety spec. (set of harmful events) SSPEC, and fault class F.
- Perform a backward propagation operation along information flow to yield potentially harmful events, i.e., events that can lead to occurrence of harmful events.
- A set of perfect detectors is obtained.

Approach - Graphically

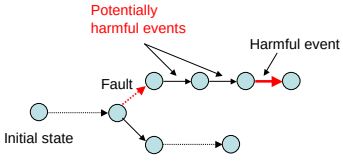


Backward Propagation

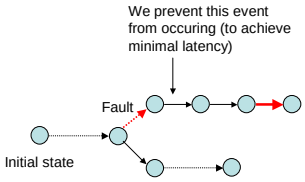


Backward Propagation (1)

Consider the state transition view of a program



Backward Propagation (2)



Fail-safe fault tolerance + minimal latency achieved

Properties of P'

- Perfect coverage
  - Ł No false alarm, rejects all harmful events.
- Minimal detection latency.

Where are we?

- Objective: Design of fail-safe fault-tolerant program with minimal detection latency.
- Perfect detectors are important.
- Use of a backward propagation operation to generate a set of perfect detectors.

What next?

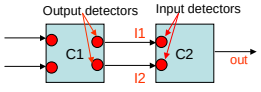
- Have to ascertain that what we have is right (validation), i.e., check if program is indeed fail-safe fault-tolerant, with minimal detection latency.
- Different methods:
  (i) Testing
  (ii) Fault injection
- Need test cases for this.

Test Case Generation

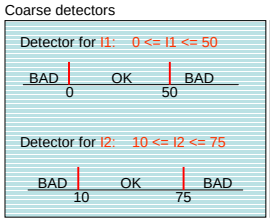
- Early: ad-hoc approach, random sampling.
- Our approach: We use detector design decisions to generate test cases.

Test Case Generation

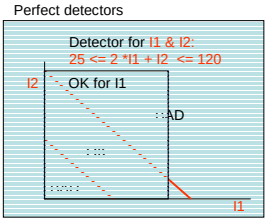
Assume that detectors are used to monitor the input and output actions of different modules, i.e, faults can occur at either input or output.



Coarse detectors



Perfect detectors



Summary

- Program-transformation based approach to design fail-safe fault-tolerant program + minimal latency.
- Addition of perfect detectors.
- Use of perfect detector design for test case generation.