Safety-Critical Computer Systems

Designing faults away

Simin Nadjm-Tehrani

www.ida.liu.se/~snt

Recall from earlier...

- Faults may lead to failures
- Failures may cause hazards
- Hazards may jeopardise safety

Thus:

- Removing/containing certain faults enhances safety

Redundancy

- Can be used to tolerate faults
- In space: HW/SW/Data
  - Transient, intermittent or permanent faults
- In time: Repeat the same computation
  - Transient faults

Designing faults away

- Tolerating faults
  - How can it be seen as a conceptual part of program (system) **design**?
- Avoiding faults
  - How can the potential for permanent faults in programs (systems) be reduced?

Adding tolerance

- How to represent a fault-intolerant system?
- What it means to add tolerance, for which type of fault, which type of method?

[Arora & Kulkarni 98, Gärtner 99]

Simple model

- Distributed reactive programs: a set of processes each with a set of variables representing local state
- Each process: a set of actions, specified as guarded commands

  Guard → Command
- Program **P**: $P_1 \,||\, P_2 \,||\ldots||\, P_n$

Guarded commands

- If the Boolean condition (the guard) for an action is true, then the action is enabled: it *may* take place

$$\neg ready \wedge y < 10 \rightarrow x := 0; z := 1$$

- Fairness: if a guard is true infinitely often the action will be eventually taken

Computations

- Each computation in the distributed system: a potentially infinite sequence of the (distributed) states
- An interleaving of computations of the individual processes

$$\sigma : s_1 s_2 \ldots s_n \ldots$$

Desired behaviours

- Behaviours: sets of computations
- Desired properties defined as sets of computations:
  - Safety (what should not happen)
  - Liveness (what should happen)
- Specification **S**: a combination of safety and liveness properties

Correctness

- To show that **P** is correct wrt **S**

show that
set of computations for **P** $\subseteq$ **S**

To add tolerance

- Must decide what fault classes to tolerate
- How to detect them
- What action to take on detection
- Later: ensure that addition of tolerance does not sacrifice correctness

Fault models

- Example: those leading to crash failures
- Extend the program with fault actions, and fault effects based on the chosen fault model

Considering faults

```
Begin
var wait: boolean init true
var up: boolean init true {* to detect error *}
{* normal actions *}
up ∧ ¬ wait → send(m); wait := true
||
up ∧ wait ∧ rec(a) → wait:= false
||
{* fault action *}
up → up := false   {* crash *}
end
```



```
Begin
var wait: boolean init true
var up: boolean init true {* to detect error *}
{* normal actions *}
up ∧ ¬ wait → send(m); wait := true
||
up ∧ wait ∧ rec(a) → wait:= false
||
{* fault action *}
up → up := false   {* crash *}
||
{* protection mechanism *}
¬ up → up := true
end
```


How does FT affect computations?

- We formalise the effects of fault-tolerance on program behaviour
- Let predicates over state variables denote the set of states in which the predicate holds

$\varphi_1 : x < 10 \wedge y < 1 \qquad \varphi_2 : x < 100 \wedge y < 10$

Formalising fault-tolerance

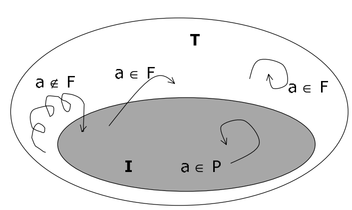
A distributed program **P** *tolerates faults* from a fault class **F** for an invariant **I** iff there exists a predicate **T** such that 3 conditions apply:

- **I** ⇒ **T**
- **T** is closed in **P** and **F**
- **P** actions in **T** eventually lead to **I**

What does it mean?

- at any state where **I** holds, **T** holds too
- starting from any state in **T**, if any **P** or **F** actions are performed, the resulting state is in **T**
- starting from any **T** state, every sequence of **P** actions alone, eventually reaches a state in **I**

Reachable system states



Fault-tolerance methods

	Live	**Not live**
Safe	Masking	Fail-safe
Not safe	Non-masking	None

Current results

- Detectors essential for safety properties
  - in distributed settings not easy
- Correctors essential for liveness properties
- Achieving both safety and liveness (masking) difficult, even in non-distributed setting

Designing faults away

- Tolerating faults
  - How can it be seen as a conceptual part of program (system) **design**?
- Avoiding faults
  - How can the potential for permanent faults in programs (systems) be reduced?

Removing permanent faults

- 40% of medical systems which are called in by FDA are due to program errors
- In a typical application 35% of the code is tested
- Is it possible to perform full testing for critical subsystems?

State space

- Consider a model M with n Boolean variables
- To decide whether M is correct wrt specification S, we must check that none of reachable states in M contradicts S
- Potential state space size: 2^n.

State space search

- With 55 variables, at a test speed of 1 MHz, it would (in worst case) take over 1 billion years to visit every state!

Other problems

- Testing heterogeneous systems costly (hardware in the loop simulations)
- Some systems can not be tested (nuclear reactors, etc.)
- System must be tested again after maintenance and adaptation to new demands (air traffic control)
- Microsoft *evolutionary model*

Verification techniques

- Inspection
- Testing
- Simulation/animation
- Static analysis
- Formal verification

Combination of techniques

- Faults in the requirements specification phase are 70 times more costly to fix, if detected during acceptance tests
- Can one find more errors at early phases of development?

⇒ Formal verification!

Inspections

- Find the fault directly instead of finding the symptom (as in testing)
- Require special training and planning
- Inspections in groups eliminate ”false alarms”

A *success story*

- C130J Hercules safety-critical program modules, 500k loc
- Upon sale - after all certification :-(
- Combination of inspections, static analysis (formal verification)
- 70 man-years, 11590 anomalies
- 3% av anomalies safety-critical