Good software gone BAD...

the Impact of Diversity of Fault Tolerance on Software Reliability

Beth Latronico – 17-939 Fall ‘01
The overview slide

- Scope
  - Goal
  - Big picture
  - What I am NOT doing

- Terms and timeline
  - Kinds of diversity
  - Evolution of techniques
  - Evolution of field

- From faults to failures
  - Contributions of techniques
  - Where are they now?

- Question-provokers
The goal

Examine:

- The evolution of redundancy techniques
  - Redundancy: Extra information or resources beyond those needed during normal system operation used in fault tolerance
  - Fault tolerance: How to ensure a service up to fulfilling the system’s function in the presence of faults

- to improve software reliability.
  - Reliability: The ability of a system or component to perform its required functions under stated conditions for a specified period of time
How does reliability fit into the ‘big picture’?

- Reliability is one “Attribute” of dependability
  - Dependability also has means of success despite impairments
- ‘Dependability’ has evolved over time, too
  - Eliminated / Altered / Added
  - Increasing scope; knowledge of fault->error->failure process
  - I need to: restrict impairments, means

Dependability: Laprie ‘85

Figure 2 - The dependability tree

Dependability: Laprie ‘95

Beth Latronico – 17-939 Wed. Nov. 14th
Faults, failures

- How does a fault turn into a failure?
  - Reliability techniques need to consider the entire process
  - I’ll talk about which pieces I’m covering

- Restrict ‘impairments’ to design faults
  - Design faults represent a major dependability bottleneck

How a fault becomes a failure

Causes of failures in a large-scale system
(client/server with several thousand workstations)
What I am not doing (faults)

- There are lots of other faults...

- I am only covering permanent design faults
What I am not doing (scope)

- There are lots of ways to improve reliability…
  - Fault prevention (code as fault-free as possible)
  - Fault removal (testing)
  - Fault forecasting (predict future faults from past data)

- Restrict the ‘means’ to fault tolerance
Evolution of techniques

- Techniques have evolved over time, roughly by:
  - Amount of foresight required (more -> less)
  - Level at which redundancy is applied (low -> high)
  - Also, interest in software reliability has increased

- My main focus includes:
  - Production of diverse faults
  - Correction of errors (through some additional resource)
    - Error detection is an enabling technology
The techniques

- **Replication**
  - Install n identical extra components running concurrently
  - Fault tolerance by brute force
  - Mentioned for historical purposes (not looking at evolution)

- **Recovery blocks**
  - Create special code to handle exceptional situations
  - Fault tolerance by foresight

- **N-version programming**
  - Create N independent versions of code
  - Fault tolerance by statistics

- **Analytic redundancy (eg Simplex method)**
  - A complicated but high-performance program can defer control to a simpler but (presumably) high-reliability program
  - Fault tolerance by reduced complexity
  - Can be used for error detection as well
Technique evolution

1950

'56: Replication proposed
von Neumann; Moore, Shannon

1960

'60: 

1970

'74: Recovery blocks proposed
Horning, Lauer, Mellear-Smith, Randell

'78: N-version proposed Avizienis, Chen

1980

'84: Large N-version experiment -Avizienis, Kelly

'85: Recovery blocks proposed
Horning, Lauer, Mellear-Smith, Randell

'85: N-version independence refuted-
Brilliant, Knight, Leveson

'87: Retry block (different data)
Ammann, Knight

1990

'87: Consensus block: Recovery block + N-version
Scott et. al.

'96/'97: Simplex method proposed
Sha

2000

'01: Modeling N-version
reliability gains
Littlewood, Popov, Strigini

'97: (or earlier?) Java formalism for recovery blocks – try/catch

Beth Latronico – 17-939 Wed. Nov. 14th
Reliability field evolution

- ’70: Fault Tolerant Computing (FTCS)
- ’74: International Reliability and Maintainability Symposium (RAMS)
- ’81 and ’82: FTCS special session on terminology
- ’85: Dependability tree, Laprie
- ’89: Symposium on Software Reliability Engineering (ISSRE)
- ’92: Dependable Computing for Critical Applications (DCCA)
- ’95: Dependability tree refined, Laprie
- ’00: FTCS + DCCA -> Dependable Systems and Networks (DSN)
- ’90: FTCS + DCCA -> Dependable Systems and Networks (DSN)

Legend:
- Blue circles: Conference inception or name change
- Purple circles: Efforts to define terminology

Beth Latronico – 17-939 Wed. Nov. 14th
Faults

- Idea: Create redundant components with different failures
  - Replication
    - Contribution: Can handle completely independent
    - Foresight: Useful for hardware with independent physical failures (independence known)
  - Recovery blocks
    - Contribution: Handles known exceptions
    - Foresight: Only works for anticipated exceptions
  - N-version programming
    - Contribution: Attempts to introduce independence to software faults
    - Foresight: Complete independence generally not achievable; must know and minimize dependencies
  - Analytic redundancy
    - Contribution: Simple backup for complicated system
    - Foresight: If primary fails, backup must not (weaker independence)
Detected errors

- **Fault masking**
  - Multiple resources: some can fail
  - Fail silent/Fail ‘totally obvious’
    - For example: watchdog, exceptions
    - Program can compensate for an identified anomaly
    - Enabler for recovery blocks

- **Voting**
  - For example: Majority voting – the majority is right
  - Relies on statistical independence of failures (to some extent)
  - Enabler for N-version

- **Data checking**
  - Outside observer performs analysis on data
    - Executable assertions – does data follow expected pattern?
    - In general, data checking is the precursor of Simplex: use simple algorithm to check accuracy of a more complicated algorithm
Tolerated errors

- Replication
  - Contribution: Tolerates completely independent faults
  - Level: Physical (extra components)

- Recovery blocks
  - Contribution: Tolerates anticipated exceptions
  - Level: Code (write additional code)

- N-version programming
  - Contribution: Faults (and hence failures) should be independent
  - Level: Algorithm (different ways of accomplishing same goal)

- Analytic redundancy
  - Contribution: Simple backup continues despite failed primary
  - Level: Requirements (systems purposefully have different requirements)
Where are these techniques now?

- Replication
  - Still used (a lot!) for hardware
  - Example: Triple modular redundancy

- Recovery blocks
  - Made their way into Java formalism
  - Most helpful for applications where operations can be undone (better for databases than nuclear power)

- N-version
  - Complete version independence proved untrue…
  - …But, has been used in industry (eg. space shuttle)

- Analytic
  - Relatively new
Food for thought

- What do you think of...

  - Evolution argument:
    - Foresight: more -> less
    - Level: low -> high

  - Use of the Fault->Failure model for analyzing contributions of techniques

- Where are they now?
  - Is this accurate?