EDA122/DIT061 Fault-Tolerant Computer Systems
DAT270 Dependable Computer Systems

Welcome to Lecture 13

Hardware reliability prediction
More on software diversity

Outline

- Hardware reliability prediction (from lecture 10)
- More on design diversity in software

Outline
(from lecture 10)

- Risk analysis
  - Risk classification
  - Acceptability of risk - ALARP
  - Assignment of Safety Integrity Levels
- ISO 26262
- Hazard analysis
  - Hazard and operability studies (HAZOP)
- Safety case
- Hardware reliability prediction

Hardware failure rates

- Ways of improving reliability of hardware
  - Decrease temperature
  - Decrease electrical stress (derating)
  - Reduce number of components or increase integration
  - Increase quality of components
  - Improve physical environment
    - Reduce exposure to moisture
    - Reduce exposure to vibrations

Examples of Failure Rate Prediction for Hardware

- MIL-HDBK-217, Military handbook, US Department of Defense, Parts Stress Model (Revision F Notice 2, released February 1995)
- Telcordia SR-332, Issue 2 (released Sept 2006)

Failure Rate Prediction Mil-Hdbk-217F

$\lambda_p = (C_1\Pi_T + C_2\Pi_E)\Pi_Q\Pi_L$ failures / 10^6 hours

λ_p is the part failure rate
C_1 is related to die complexity
Π_T is related to ambient temperature
C_2 is related to the package type
Π_E is determined by the operating environment
Π_Q is determined by the part quality
Π_L represents the learning factor and is determined by the experience of the manufacturer.

Telcordia SR-332 (Bellcore)

$\lambda_{SS} = \lambda_G \Pi_Q\Pi_S\Pi_T$ failures / 10^6 hours

λ_{SS} is the steady state failure rate
λ_G is the generic steady state failure rate (table look up based on field data)
Π_Q is determined by the part quality
Π_S is determined by the electrical stress
Π_T is related to operating temperature

Standards for hardware reliability prediction

- **MIL-HDBK-217 Part Stress & Part Count**
  MIL-HDBK-217 F Notice 2.
- **217Plus - Based on Handbook of 217PlusTM**
  Reliability Prediction Models, 26 May 2006 by Reliability Information Analysis Center (RIAC).
- **Telcordia Issue 2 -** Reliability Prediction Procedure for Electronic Equipment, SR-332, Issue 2, September 2006
- **IEC 62380 (RDF 2003)**
  Updated version of RDF 2000 UTEC 80810 method – French Telecom reliability prediction Standard. It includes most of the same components as MIL-HDBK-217.

Standards for hardware reliability prediction

- **FIDES Guide 2009**
  The FIDES methodology is applicable to all domains using electronics: aeronautical, naval, military, production and distribution of electricity, automobile, railway, space, industry, telecommunications, data processing, home automation, household appliances.
- **BRT - British Telecom -** British Telecom Module for reliability prediction based on British Telecom document HRD-4 or HRD-5.
- **GJB299 -** Chinese reliability standard.
- **Siemens SN29500.1 -** Siemens reliability standard.

Design Diversity

Design diversity is used to tolerate development faults in hardware and software

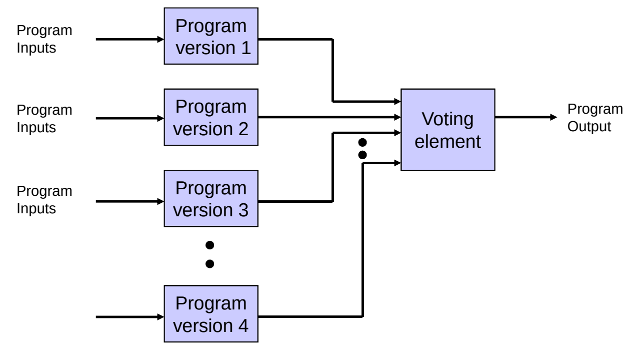
Two techniques for tolerating software design faults:

- N-version programming
- Recovery blocks

N-version programming

- Uses majority voting on results produced by N program versions
- Program versions are developed by different teams of programmers
- Assumes that programs fail independently
- Resembles hardware voting redundancy

N-version programming



Ensuring independence in N-version programming

- Use different design teams for each version
- Use diverse specifications
- Prevent cooperation among design teams
- Use diverse programming languages, compilers, CASE tools, etc.
- …

Evaluation of N-version programming

Objective
- To investigate if independently developed programs fail independently

Overview
- Missile interceptor program
- 27 versions produced by students at University of Virginia and University of California, Irvine.
- All students was given the same specification
- 200 test cases to validate each program
- 1 million test cases to test independence (simulation of production environment)
- Published 1985

Knight, J.C., N.G. Leveson, and L.D. St. Jean, ”A Large Experiment in N-version Programming”, Digest of Papers, Int. Symposium on Fault-tolerant Computing (FTCS-15), Ann Arbor, Michigan, June, 1985, pp. 135-139.

Experimental set-up (1)

- 27 versions produced by senior-level students
  - 9 versions from University of Virginia
  - 18 versions from University of California, Irvine
  - Written in Pascal
- Program for anti-missile system
  - Determines if radar reflections represents a incoming hostile missile.
  - Well-known problem – previously used in software engineering experiments.

Experimental set-up (1)

- Input to students
  - Requirements specification
  - Instructed not to cooperate or discuss the problem amongst themselves
  - No restrictions on the use of references
  - 12 input data sets for debugging
- Acceptance test for programs
  - 200 randomly generated tests
  - Different set of tests for each program
  - Resembles testing in real systems
  - Only programs that passed the acceptance test was used in the experimental data

Table 1 – Version Failure Data

Version	Failures	Reliability	Version	Failures	Reliability
1	2	0.999998	15	0	1.000000
2	0	1.000000	16	62	0.999938
3	2297	0.997703	17	269	0.999731
4	0	1.000000	18	115	0.999885
5	0	1.000000	19	264	0.999736
6	1149	0.998851	20	936	0.999064
7	71	0.999929	21	92	0.999908
8	323	0.999677	22	9656	0.990344
9	53	0.999947	23	80	0.999920
10	0	1.000000	24	260	0.999740
11	554	0.999446	25	97	0.999903
12	427	0.999573	26	883	0.999117
13	4	0.999996	27	0	1.000000
14	1368	0.998632			

Evaluation of N-version programming

Occurrence of Multiple Program Failures

# Failed Programs	# Test Cases
2	551
3	343
4	243
5	73
6	32
7	12
8	2

Conclusion: The programs in this experiment do not fail independently*!

(1256 multiple failures, 21257 single failures)

*The hypothesis of independence is rejected at the 99% confidence level.

Table 3 – Correlated Failures Between UVA And UCI

UCI Versions	UVA Versions 1	2	3	4	5	6	7	8	9
10	0	0	0	0	0	0	0	0	0
11	0	0	58	0	0	2	1	58	0
12	0	0	1	0	0	0	71	1	0
13	0	0	0	0	0	0	0	0	0
14	0	0	28	0	0	3	71	26	0
15	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	1	0	0	0
17	2	0	95	0	0	0	1	29	0
18	0	0	2	0	0	1	0	0	0
19	0	0	1	0	0	0	0	1	0
20	0	0	325	0	0	3	2	323	0
21	0	0	0	0	0	0	0	0	0
22	0	0	52	0	0	15	0	36	2
23	0	0	72	0	0	0	0	71	0
24	0	0	0	0	0	0	0	0	0
25	0	0	94	0	0	0	1	94	0
26	0	0	115	0	0	5	0	110	0
27	0	0	0	0	0	0	0	0	0

Discussion (1)

Is it realistic to use students in a software engineering experiment?

- Programming experiences of students outside their degree programs
  - 12 students had less than two years of programming experience
  - 10 students had between two and five years of programming experience
  - 5 students had more than five years of programming experience
- Students had diverse backgrounds

Discussion (2)

Is one million test cases enough?

- Test cases represent “unusal” events.
- “If the program is executed once per second and unusal events occur every ten minutes, then one million test cases correspond to 20 years of operational use”

Conclusions of NVP study (1)

- The assumption of independence of failures among versions ***does not hold***
- The above does not render NVP useless! - It merely shows that the impact of correlated failures must be taken into consideration when estimating the reliability of systems that use NVP.
- The result is only valid for the application used
- Similar results may, or may not, be observed for other applications.

Conclusions of NVP study (2)

- More than half of the software fault was present in two or more programs
- Possible explanations for the high percentage of correlated faults:
  - Programmers make similar mistakes
  - Certain parts of the problem is difficult and lead to mistakes by many programmers
  - Flaws causing uncorrelated failures are easy to catch by normal debugging

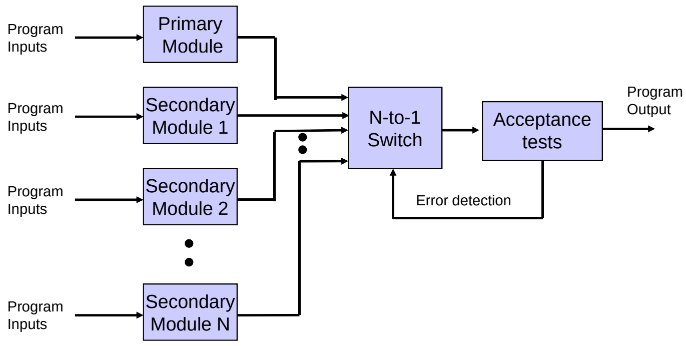
Conclusions of NVP study (3)

- Need for further research
  - More experiments needed to draw general conclusions
  - Possible explanations for the high percentage of correlated faults need to be investigated.
  - Relying on random chance to obtain diversity may not be an effective approach. Deliberate diversity may work better.

Recovery Blocks

- Uses one primary software module and one or several secondary (back-up) software modules
- Assumes that program failures can be detected by acceptance tests
- Executes only the primary module under error-free conditions
- Resembles dynamic hardware redundancy

Recovery blocks



Construction of acceptance tests

- An acceptance test is a software implemented check designed to detect errors in the results produced by a primary or a secondary module
- Acceptance tests often relies on application specific information
- An acceptance test is similar to a software assertion (a.k.a. executable assertion).

Software Assertions

- Executes ***consistency checks*** on application data or operating system data
- Implemented in software
- Automatic generation
  – E.g., run-time type checking generated by complier
- Manual generation
  – E.g., application programmer inserts checks on a valid temperature range, acceleration, etc.

Examples of how acceptance tests/ software assertions can be constructed

- Satisfaction of requirements
  - Inversion of mathematical functions; e.g. squaring the result of a square-root operation to see if it equals the original operand.
  - Checking sort functions; result should have elements in descending order

  …
- Reasonable checks
  - Checking physical constraints; e.g. speed, pressure, etc
  - Checking sequence of application states

  …

Evaluation of Recovery Blocks

- Goal: to evaluate recovery blocks for a medium-scale naval command and control system (concurrent real-time system)
- The system provides a simulated radar display overlaid with tracking information. Allows the operator to attack hostile submarines.
- 8000 lines of source code in CORAL, 14 concurrent activities
- Programmed by professional programmers
- Recovery supported by a special recovery cache

Conduct of Experiment

- The command and control system was run against an environment simulator by the operator
- Several typical scenarios were simulated
- Operator logged all abnormal behaviors of the system
- Monitoring routines within the system recorded recovery and failure events

Evaluation of recovery blocks

Naval command and control system (8000 statements in the Coral language)

117 abnormal events

Correct recovery	78 %
Incorrect recovery, program failure	3 %
Incorrect recovery, no program failure	15 %
Unnecessary recovery	3 %

Anderson, T., et al., "Software Fault Tolerance: An Evaluation," IEEE Trans. on Software Engineering, vol. SE-11, no. 12, Dec 1985, pp. 1502-1510.

Overhead for the Case Study

- 60% supplementary development cost
- 33% extra code memory
- 35% extra data memory
- 40% extra execution time

Comparison of N-version programming and Recovery blocks

N-version programming
- Applied at the program level
- Runs N programs at the same time
- Resembles static hardware redundancy
- Assumes that independence among program versions is achieved by random differences in programming style among programmers

Recovery blocks
- Applied at the module (subprogram) level
- Runs only the primary module under error-free conditions
- Resembles dynamic hardware redundancy
- Independence is achieved by deliberately designing the primary and secondary modules to be as different as possible

Overview of Lecture 14

- Byzantine failures
  Preparations:
  - Byzantine Agreement, Section 3.1
  - Lecture slides
- Error detection and time redundancy
  Preparations:
  - Section 6.3 and 6.4 in the course book
  - Lecture slides