Fault-Tolerant Computing

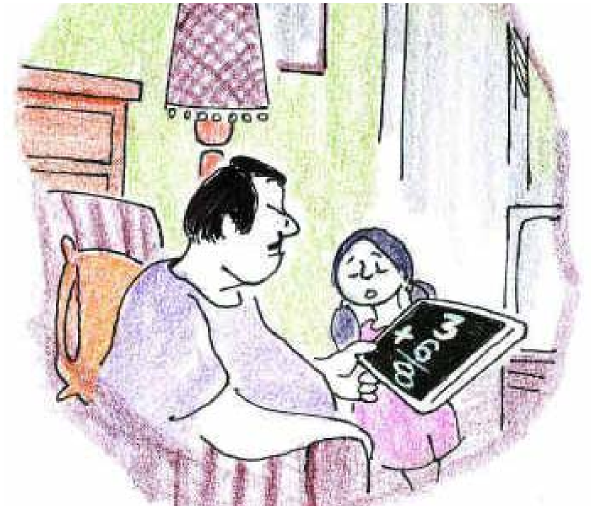
Software
Design
Methods

About This Presentation

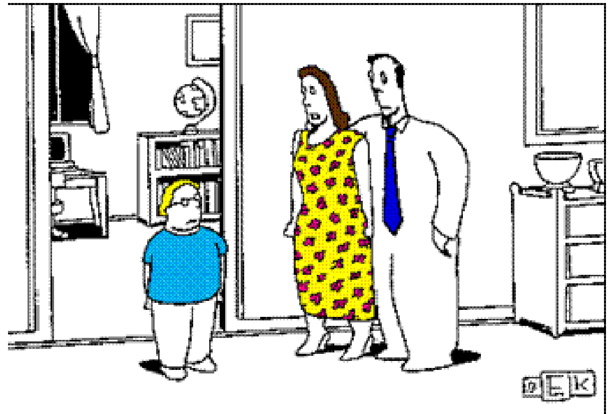
This presentation has been prepared for the graduate course ECE 257A (Fault-Tolerant Computing) by Behrooz Parhami, Professor of Electrical and Computer Engineering at University of California, Santa Barbara. The material contained herein can be used freely in classroom teaching or any other educational setting.

Edition	Released	Revised	Revised
First	Nov. 2006		

Software Reliability and Redundancy

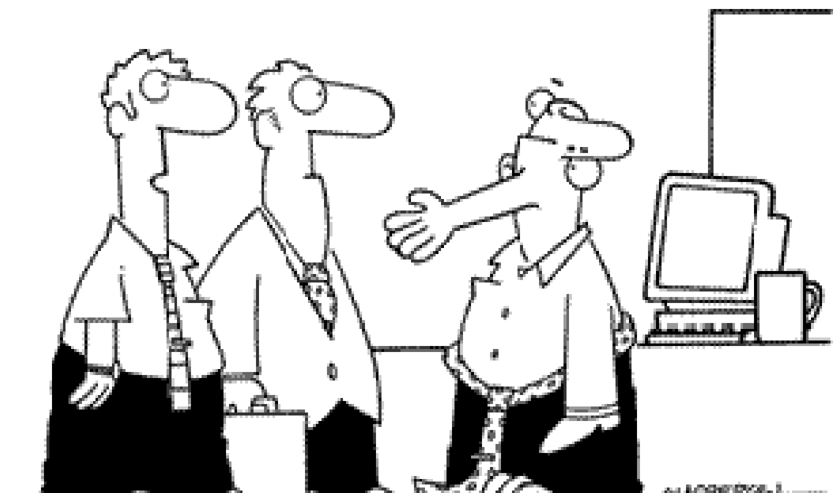


“Well, what’s a piece of software without a bug or two?”



“We are neither hardware nor software; we are your parents.”

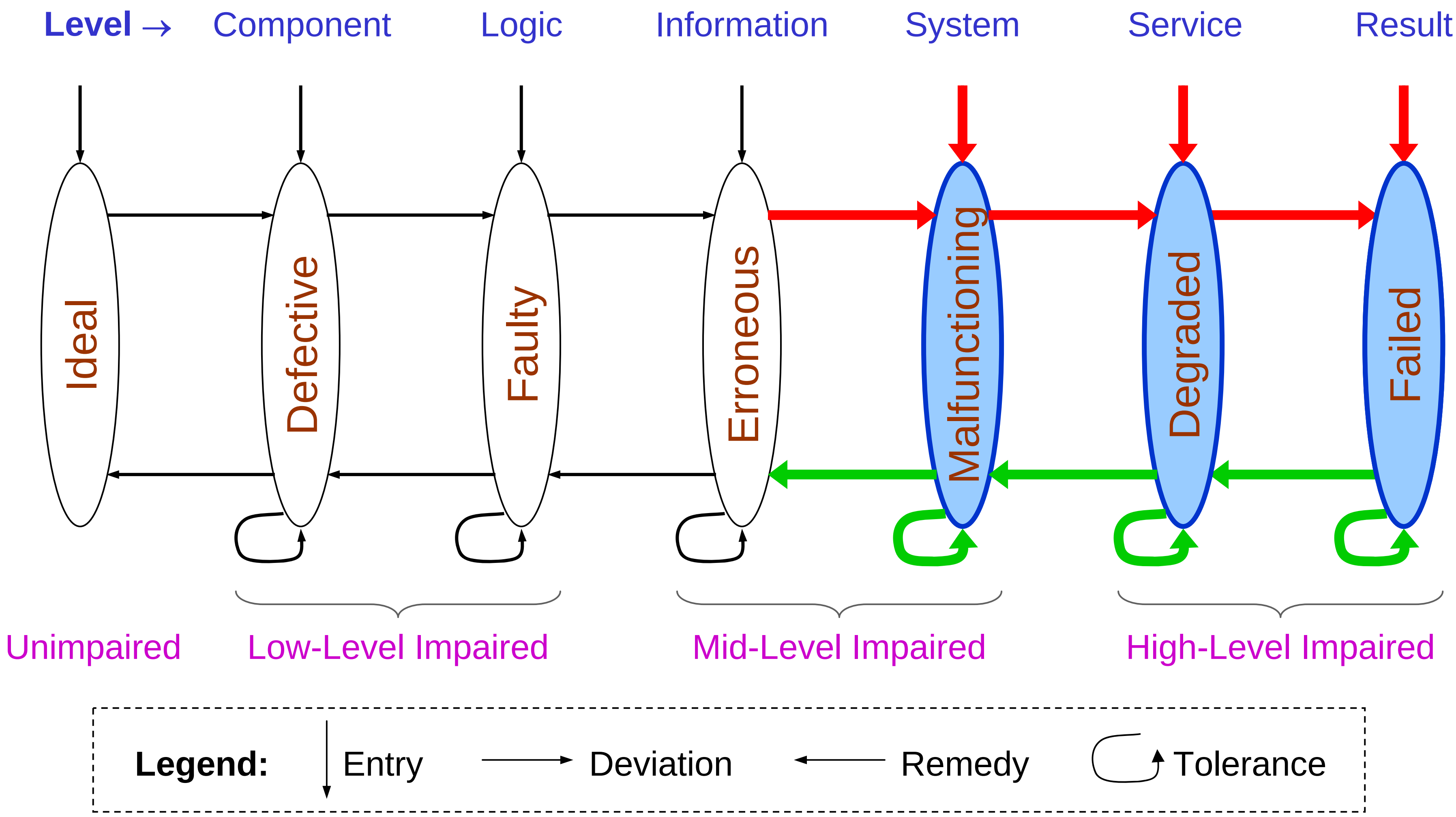
“There’s nothing wrong with your personal finance software. You just don’t have any money.”



“That’s our CIO. He’s encrypted for security purposes.”

“I haven’t the slightest idea who he is. He came bundled with the software.”

Multilevel Model of Dependable Computing



How Is Software Different from Hardware?

Software unreliability is caused predominantly by design slips, not by operational deviations – we use *flaw* or *bug*, rather than *fault* or *error*

Not much sense in replicating the same software and doing comparison or voting, as we did for hardware

At the current levels of hardware complexity, latent design slips also exist in hardware, thus the two aren't totally dissimilar

The curse of complexity

The 7-Eleven convenience store chain spent nearly $9M to make its point-of-sale software Y2K-compliant for its 5200 stores
The modified software was subjected to 10,000 tests (all successful)
The system worked with no problems throughout the year 2000
On January 1, 2001, however, the system began rejecting credit cards, because it "thought" the year was 1901 (bug was fixed within a day)

Software Development Life Cycle

> Software flaws may arise at several points within these life-cycle phases

Project initiation	
Needs	
Requirements	
Specifications	
Prototype design	
Prototype test	Evaluation by both the developer and customer
Revision of specs	
Final design	
Coding	Implementation or programming
Unit test	Separate testing of each major unit (module)
Integration test	Test modules within pretested control structure
System test	
Acceptance test	Customer or third-party conformance-to-specs test
Field deployment	
Field maintenance	
System redesign	New contract for changes and additional features
Software discard	Obsolete software is discarded (perhaps replaced)

What Does Software Reliability Mean?

Major structural and logical problems are removed very early in the process of software testing

What remains after extensive verification and validation is a collection of tiny flaws which surface under rare conditions or particular combinations of circumstances, thus giving software failure a statistical nature

Software usually contains one or more flaws per thousand lines of code, with < 1 flaw considered good (linux has been estimated to have 0.1)

If there are f flaws in a software component, the hazard rate, that is, rate of failure occurrence per hour, is kf, with k being the constant of proportionality which is determined experimentally (e.g., $k = 0.0001$)

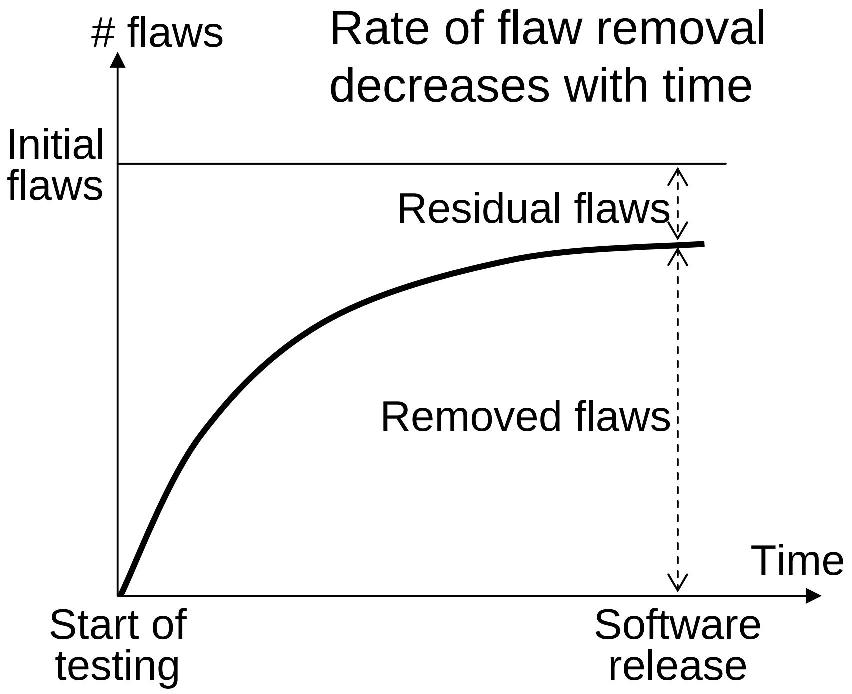
Software reliability: $R(t) = e^{-kft}$

The only way to improve software reliability is to reduce the number of residual flaws through more rigorous verification and/or testing

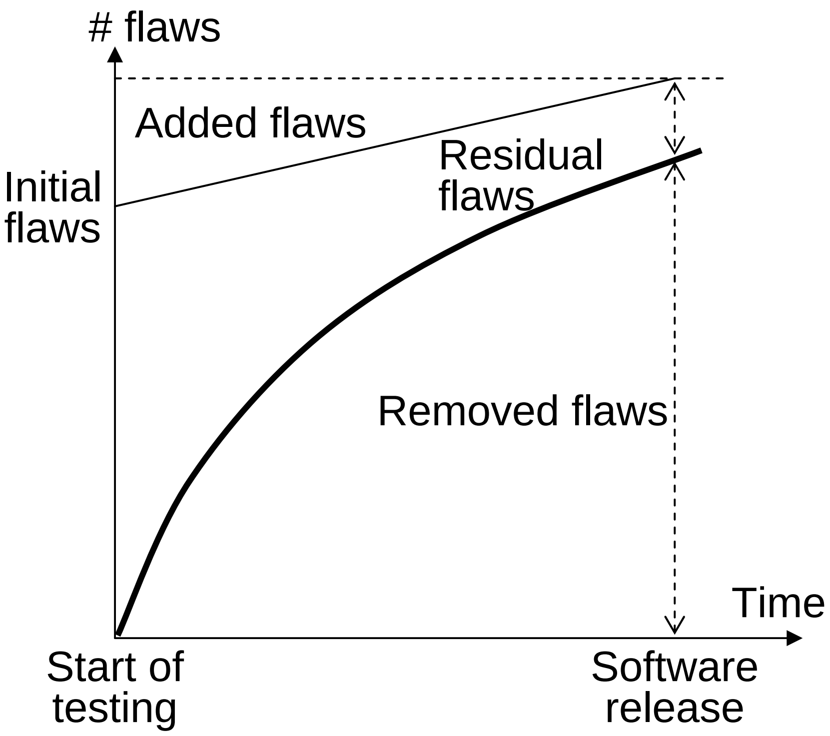
Software Error Models

Software flaw/bug $\Rightarrow$ Operational error $\Rightarrow$ Software-induced failure

"Software failure" used informally to denote any software-related problem



Removing flaws, without generating new ones



New flaws introduced are proportional to removal rate

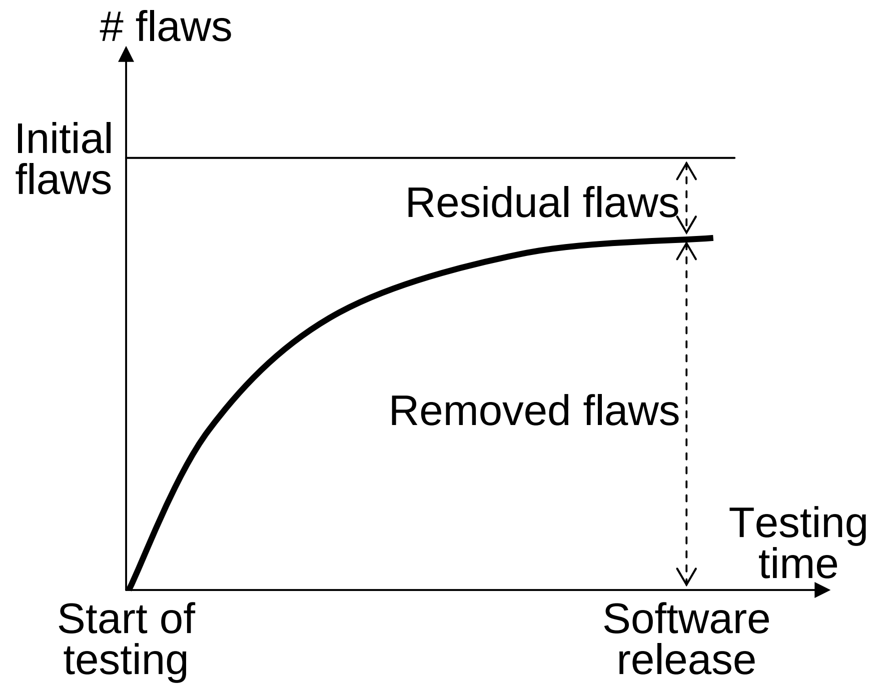
Software Reliability Models and Parameters

For simplicity, we focus on the case of no new flaw generation

Assume linearly decreasing flaw removal rate (F = residual flaws, τ = testing time, in months)

$dF(\tau)/d\tau = -(a - b\tau)$
$F(\tau) = F_0 - a\tau(1 - b\tau/(2a))$

Example: $F(\tau) = 130 - 30\tau(1 - \tau/16)$



Hazard function

$z(\tau) = k(F_0 - a\tau(1 - b\tau/(2a)))$

In our example, let $k = 0.000132$
$R(t) = \exp(-0.000132(130 - 30\tau(1 - \tau/16))t)$

Assume testing for $\tau = 8$ months:

$R(t) = e^{-0.00231t}$

τ	MTTF (hr)
0	58
2	98
4	189
6	433
8	758

More on Software Reliability Models

Linearly decreasing flaw removal rate isn't the only option in modeling

Constant flaw removal rate has also been considered, but it does not lead to a very realistic model

Exponentially decreasing flaw removal rate is more realistic than linearly decreasing, since flaw removal rate never really becomes 0

How does one go about estimating the model constants?

- Use handbook: public ones, or compiled from in-house data
- Match moments (mean, 2^{nd} moment, . . .) to flaw removal data
- Least-squares estimation, particularly with multiple data sets
- Maximum-likelihood estimation (a statistical method)

Software Verification and Validation

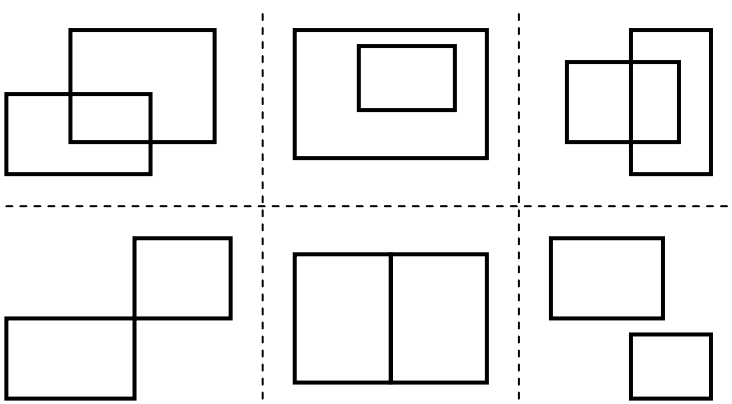
Verification: "Are we building the system right?" (meets specifications)

Validation: "Are we building the right system?" (meets requirements)

Both verification and validation use testing as well as formal methods

Software testing
Exhaustive testing impossible
Test with many typical inputs
Identify and test fringe cases

Example: overlap of rectangles



Formal methods
Program correctness proof
Formal specification
Model checking

Examples: safety/security-critical

Railway interlocking system
[Hlavaty 2001]

Cryptography device
[Kirby 1999]

Smart cards
[Requet 2000]

Automated lab analysis test equipment
[Bicarregui 1997]

Formal Proofs for Software Verification

Program to find the greatest common divisor of integers $m > 0$ and $n > 0$

input m and n
$x := m$
$y := n$
while $x \neq y$
   if $x < y$
   then $y := y - x$
   else $x := x - y$
   endif
endwhile
output x

m and n are positive integers

x and y are positive integers, $x = m$, $y = n$

Loop invariant: $x > 0$, $y > 0$, $\gcd(x, y) = \gcd(m, n)$

$x = \gcd(m , n)$

Steps 1-3: "partial correction"
Step 4: ensures "total correctness"

The four steps of a correctness proof relating to a program loop:

1. Loop invariant implied by the assertion before the loop (precondition)
2. If satisfied before an iteration begins, then also satisfied at the end
3. Loop invariant and exit condition imply the assertion after the loop
4. Loop executes a finite number of times (termination condition)

Software Flaw Tolerance

Flaw avoidance strategies include (structured) design methodologies, software reuse, and formal methods

Given that a complex piece of software will contain bugs, can we use redundancy to reduce the probability of software-induced failures?

The ideas of masking redundancy, standby redundancy, and self-checking design have been shown to be applicable to software, leading to various types of fault-tolerant software

“Flaw tolerance” is a better term; “fault tolerance” has been overused

Masking redundancy: N-version programming

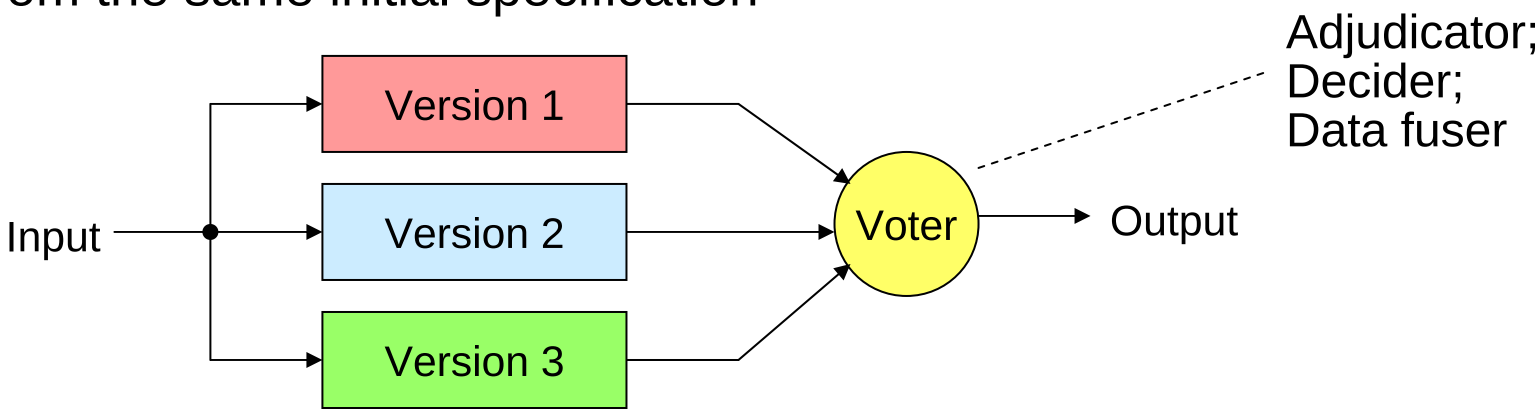
Standby redundancy: the recovery-block scheme

Self-checking design: N-self-checking programming

Sources: *Software Fault Tolerance*, ed. by M.R. Lyu, Wiley, 2005 (on-line book at http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html)
Also, “Software Fault Tolerance: A Tutorial,” 2000 (NASA report, available on-line)

N-Version Programming: The Idea

Independently develop *N* different programs (known as “versions”) from the same initial specification



The greater the diversity in the *N* versions, the less likely that they will have flaws that produce correlated errors

Diversity in:
Programming teams (personnel and structure)
Software architecture
Algorithms used
Programming languages
Verification tools and methods
Data (input re-expression and output adjustment)

N-Version Programming: Some Objections

Developing programs is already a very expensive and slow process; why multiply the difficulties by *N*?	Cannot produce flawless software, regardless of cost
Diversity does not ensure independent flaws (It has been amply documented that multiple programming teams tend to overlook the same details and to fall into identical traps, thereby committing very similar errors)	This is a criticism of reliability modeling with independence assumption, not of the method itself
Imperfect specification can be the source of common flaws	Multiple diverse specifications?
With truly diverse implementations, the output selection mechanism (adjudicator) is complicated and may contain its own flaws	Will discuss the adjudication problem in a future lecture

N-Version Programming: Reliability Modeling

Fault-tree model: the version shown here is fairly simple, but the power of the method comes in handy when combined hardware/software modeling is attempted

Probabilities of coincident flaws are estimated from experimental failure data

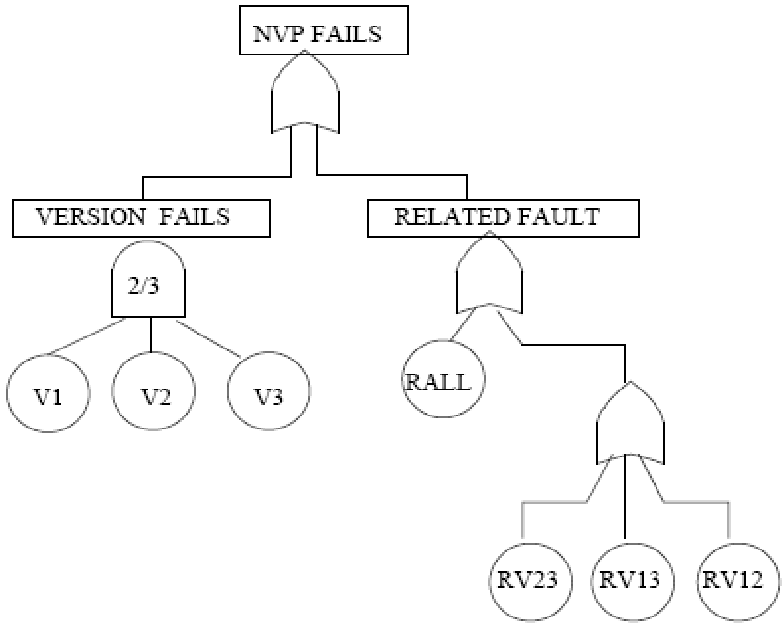


Table 5.6 Error characteristics for four-version configurations

Category	BY-CASE		BY-FRAME	
	Number of cases	Frequency	Number of cases	Frequency
F_0 - no errors	322010	0.65052	2613781410	0.9998951
F_1 - single error	152900	0.30889	2719200	0.001040
F_2 - two coincident	16350	0.03303	2070	0.00000079
F_3 - three coincident	3700	0.00747	0	0.0
F_4 - four coincident	40	0.00008	0	0.0
Total	495000	1.0000	2614055400	1.000000

Source: Dugan & Lyu, 1994 and 1995

N-Version Programming: Applications

Back-to-back testing: multiple versions can help in the testing process

B777 flight computer: 3 diverse processors running diverse software

Airbus A320/330/340 flight control: 4 dissimilar hardware/software modules drive two independent sets of actuators

Some experiments in N-version programming

Experiment	Specs	Languages	Versions	Reference
Halden, Reactor Trip	1	2	2	[Dah79]
NASA, First Generation	3	1	18	[Kel83]
KFK, Reactor Trip	1	3	3	[Gme80]
NASA/RTI, Launch Interceptor	1	3	3	[Dun86]
UCI/UVA , Launch Interceptor	1	1	27	[Kni86a]
Halden (PODS), Reactor Trip	2	2	3	[Bis86]
UCLA, Flight Control	1	6	6	[Avi88]
NASA (2nd Gen.) Inertial Guidance	1	1	20	[Eck91]
UI/Rockwell, Flight Control	1	1	15	[Lyu93]

Source: P. Bishop, 1995

Recovery Block Scheme: The Idea

The software counterpart to standby sparing for hardware

Suppose we can verify the result of a software module by subjecting it to an acceptance test

ensure	*acceptance test*	e.g., sorted list
by	*primary module*	e.g., quicksort
else by	*first alternate*	e.g., bubblesort
⋮		⋮
else by	*last alternate*	e.g., insertion sort
else fail		

The acceptance test can range from a simple reasonableness check to a sophisticated and thorough test

Design diversity helps ensure that an alternate can succeed when the primary module fails

Recovery Blocks: The Acceptance-Test Problem

Design of acceptance tests (ATs) that are both simple and thorough is very difficult; for example, to check the result of sorting, it is not enough to verify that the output sequence is monotonic

Simplicity is desirable because acceptance test is executed after the primary computation, thus lengthening the critical path

Thoroughness ensures that an incorrect result does not pass the test (of course, a correct result always passes a properly designed test)

Some computations do have simple tests (inverse computation)
Examples: square-rooting can be checked through squaring, and roots of a polynomial can be verified via polynomial evaluation

At worst, the acceptance test might be as complex as the primary computation itself

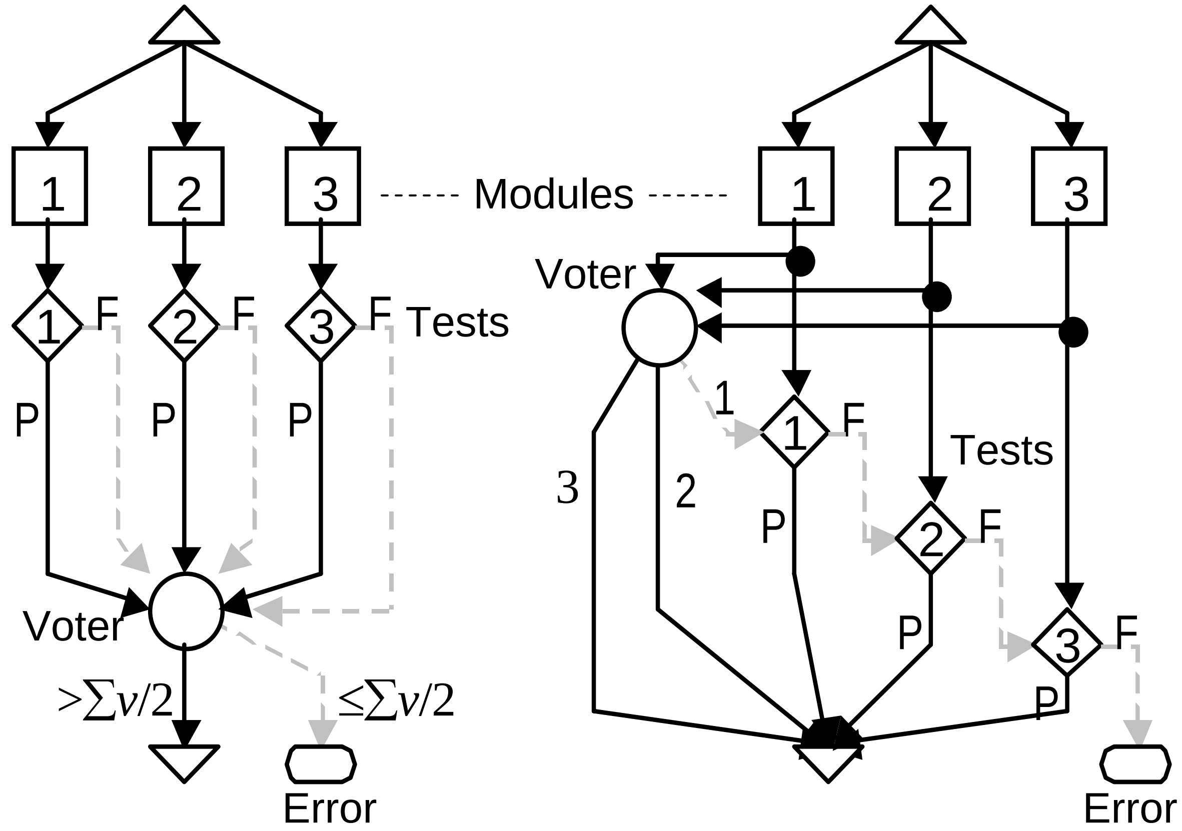
Combined NVP and Acceptance Testing

Recoverable N-version block scheme = N-self-checking program

Voter acts only on module outputs that have passed an acceptance test

Consensus recovery block scheme

Only when there is no majority agreement, acceptance test applied (in a prespecified order) to module outputs until one passes its test

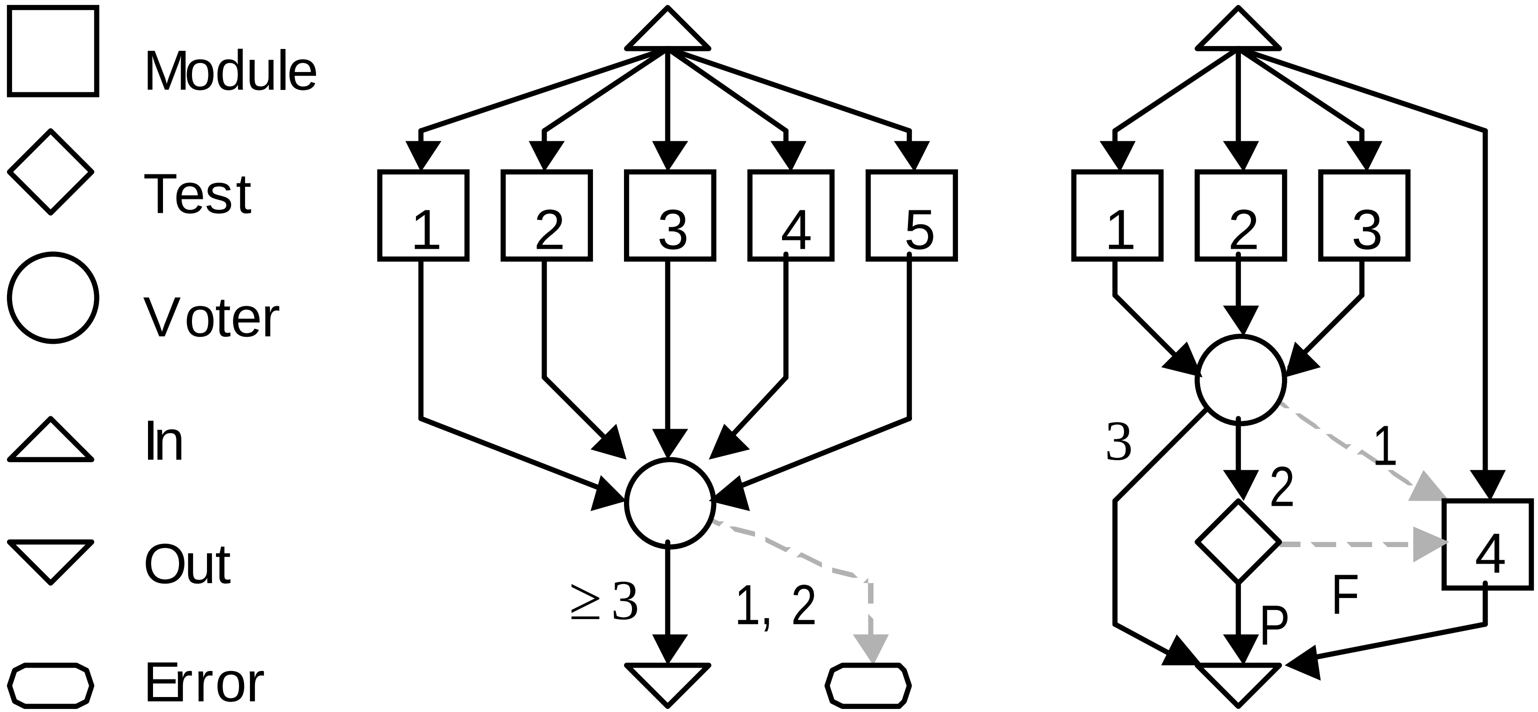


(a) RNVB / NSCP

(b) CRB

Source: Parhami, B., "An Approach to Component-Based Synthesis of Fault-Tolerant Software," *Informatica*, Vol. 25, pp. 533-543, Nov. 2001.

More General Hybrid NVP-AT Schemes



(a) Legend (b) 5VP (c) ALT1

Source: Parhami, B., “An Approach to Component-Based Synthesis of Fault-Tolerant Software,” *Informatica*, Vol. 25, pp. 533-543, Nov. 2001.