Dependable Systems

Software Dependability

Dr. Peter Tröger

Sources:

Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial.

Brian Randell and Jie Xu. The Evolution of the Recovery Block Concept.

Lui Sha. Using Simplicity to Control Complexity.

Several publications by Avizienis et al.

Software Dependability

- Four inherent properties that make software hard [Brooks 87]
  - **Complexity** - Huge number of states with non-linear interactions
  - **Conformity** - Software must fulfill outside / inside system expectations
  - **Changeability** - New and revised system functionality is appealing
  - **Invisibility** - Real activity reasoned by the code is mostly not obvious

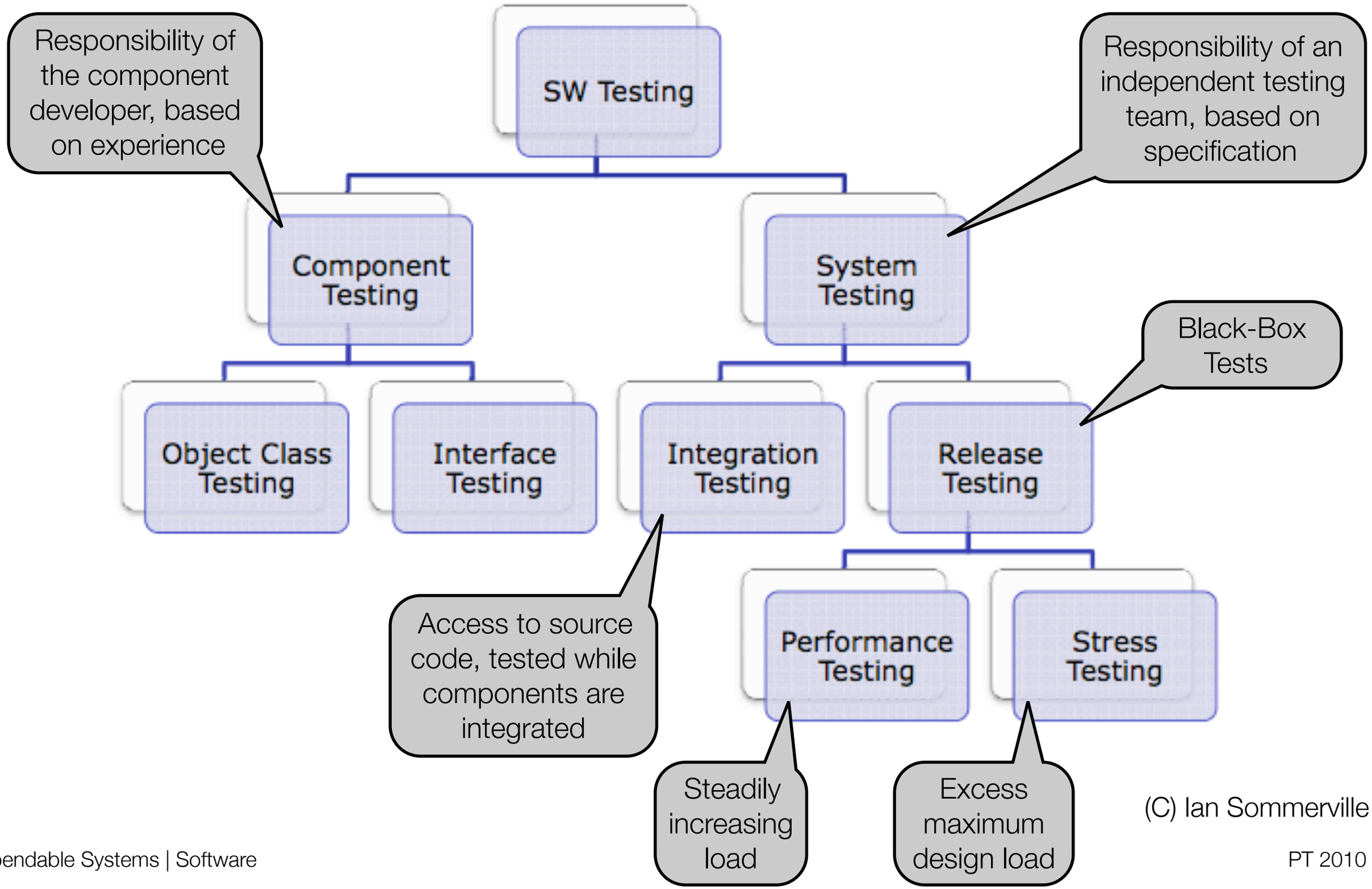
Forget Hardware ...

- In 1990 there were an estimated **120 billion lines** of source code being maintained (Ulrich, 1990).
- In 2000 there are already about **250 billion lines** of source code being maintained, and that number is increasing (Sommerville, 2000).
- As a result, the amount of code maintained **doubles in size every 7 years** (Müller et al., 1994).
- Older languages are not dead. E.g. **70%** or more of the still active business applications are written in COBOL (Giga Information Group).
- There are at least **200 billion lines of COBOL-code** still existing in mainframe computers alone (Gartner Group).
- Source: http://www.cs.jyu.fi/~koskinen/smcosts.htm
- Some collected reference data and fault models for software available, e.g.
  - http://hissa.nist.gov/effProject/ , http://www.amber-project.eu/

Software Dependability

- **Software testing**
  - Reduce number of dormant faults at development time
- Fault-tolerant software
  - Techniques to achieve fault tolerance for software faults
  - Application of redundancy idea to software modules
- Software fault tolerance
  - Techniques to achieve fault tolerance by software mechanisms
  - Typically for hardware failures on lower levels in the system stack
  - Redundancy managed by operating system, cluster framework, application code

Software Testing



Testing Approaches

- **Integration testing**
  - Top-down integration: Start with system skeleton
    - Better in discovering design errors in the system architecture
    - Allows limited prototype at an early stage
  - Bottom-up integration: Incremental integration of dependent components
    - Makes test development easier
- **Release resp. Acceptance testing**
  - Only based on system specification
  - Increase vendor‘s confidence into the product readiness

Testing Approaches

- **Component / unit testing**
  - Defect testing in an isolated code piece
  - Component might be function, class, or composite component with interface
- **Object class testing**
  - Complete coverage would mean to test all operations, all attribute combinations, all possible object states
  - Inheritance brings problem of localizing the test candidate

Test Case Design

- Aims at good validation and defect testing coverage
- Design approaches
  - **Requirements-based testing**
    - Often implemented by deriving test cases from described use cases
  - **Partition testing**
    - Often input parameters can be clustered - take one candidate from each set
    - Typical applications with mathematical properies (<, >, <<, >>, even, odd, ...)
- **Structural testing resp. white box testing**
  - Knowledge of the program is used to identify additional test cases
  - Excercise all program statements (not all code paths)

Testing Through Software Fault Injection

- Intentionally trigger erroneous behavior of the execution environment
  - Typical approach for communicating software entities
  - Orientation towards implementation details - program state, functional behavior
  - Several non-intrusive implementation techniques, such as AOP
    - Injection can be done as part of execution under real conditions
    - Huge variety of fault classes, including SWIFI fault classes
    - New approaches support remote fault injection (e.g. fuzzing)
- **Compile-time injection:** Leads to erroneous image being executed
- **Run-time injection:** Demands some altering of application state during runtime
- Typical triggers: Time-out, exception, debugging trap, code insertion

Software Dependability

- Software testing
  - Reduce number of dormant faults at development time
- **Fault-tolerant software**
  - Techniques to achieve fault tolerance for software faults
    - Application of redundancy idea to software modules
- Software fault tolerance
  - Techniques to achieve fault tolerance by software mechanisms
  - Typically for hardware failures on lower levels in the system stack
  - Redundancy managed by operating system, cluster framework, application code

Fault-Tolerant Software

- Fault-tolerant software unit: Continues to deliver service in an error state
  - Non-fault-tolerant software unit is called **simplex unit**
- Examples
  - Algorithmic calculation problems
    - Need to investigate units, operators, intervals, limits, ranges, FP handling, ...
  - Problems with input data
    - Units, ranges, change quantity or frequency, ...
    - Prevention: Assertions for invalid values, checks for ranges that imply incorrect data. Implementation of input data validation for code
  - Problems with initialization, interfaces, control logic, omission of system parts, timing /synchronization, exceptional conditions

Fault-Tolerant Software

- **Forward-error recovery**
  - Feasible in control loop / real-time systems, since propagation is predictable
  - Timing faults with the real-time criteria would be more severe
  - Typical example: N-version programming
- **Backward-error recovery**
  - In case of unpredictable error propagation effects
  - Typically expensive in terms of time
  - Ensure best-possible correctness of ultimate computational outcome
  - Large variety in examples: Retry, restart, reboot, checkpointing, audit logs, transactions, recovery blocks, exceptions, wrappers **...**

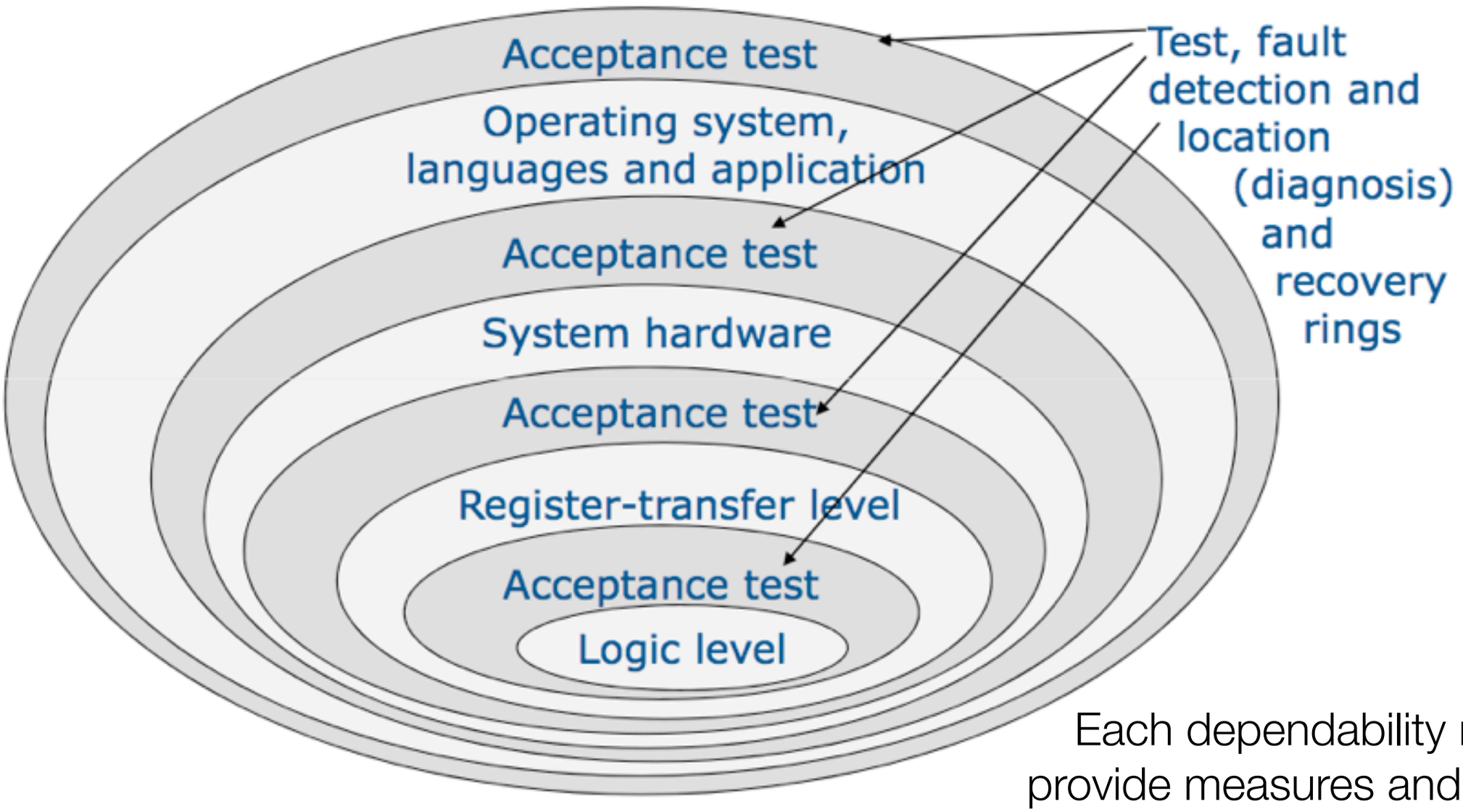
Fault-Tolerant Software - Another categorization [Lyu 95]

- **Single version techniques**
  - Add mechanisms for detection, containment, and handling of errors to the software component
  - Examples: Software structure and actions approaches, error detection, exception handling, checkpointing and restart, process pairs, data diversity
- **Multi version techniques**
  - Rely on structured utilization of variants of the same software
  - Examples: Recovery blocks, N-version programming
- Principles can be applied to any software layer
  - Identify source of most design faults
  - Typically no problem with parallel application, beside cost factor

Single-Version Approaches - Wrapper

- Piece of software that encapsulates a given program when it is being executed
- Typical approach for operating systems and middleware stacks
- Structure: Wrapper Software and Wrapped Entity
- Inputs and outputs are checked by the wrapper
- Examples include:
  - Dealing with buffer overflow, checking scheduler correctness (e.g., EDF), bypassing known bugs, checking output correctness
- When pre- or postconditions are violated, usually an exception is being raised
- Wrappers form acceptance tests in the „rings of dependability“
- Good approach for patching efforts after product delivery

Dependability Rings for Fault Tolerance



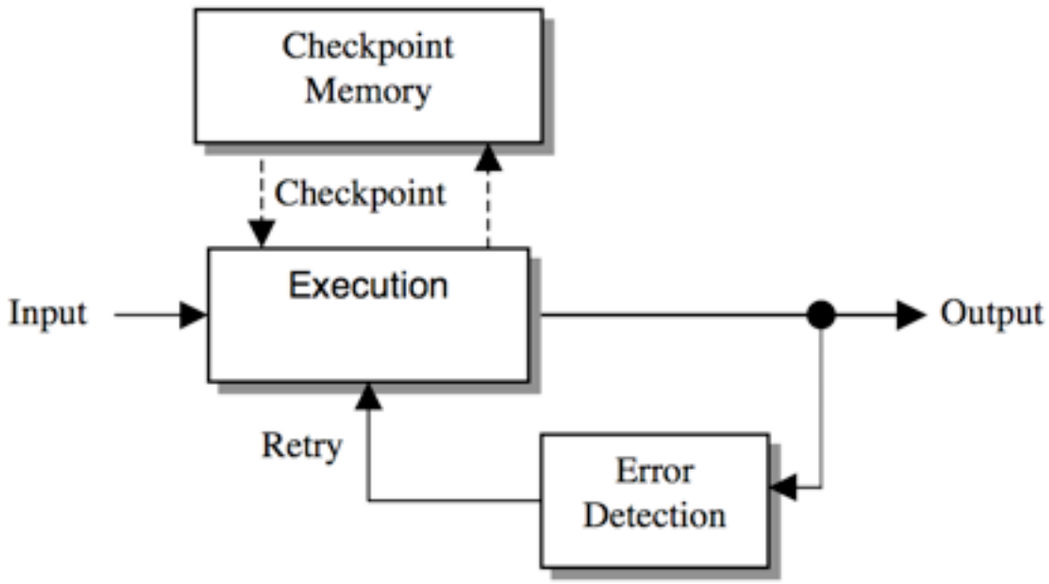
Each dependability ring should provide measures and mechanisms for fault tolerance (detection, location, testability and recovery)

Single-Version Approaches - Software Structures and Actions

- **Partitioning**
  - Isolate functionally independent modules
  - Horizontal (n-tier architectures) vs. vertical partitioning (factoring)
- **System closure**
  - No action is permissible unless explicitly authorized
  - Any capability damaged only disables a valid action
- **Temporal structuring**
  - Enable atomic interactions between components without disturbance
  - From outside: Either terminates correctly, or is aborted upon error detection

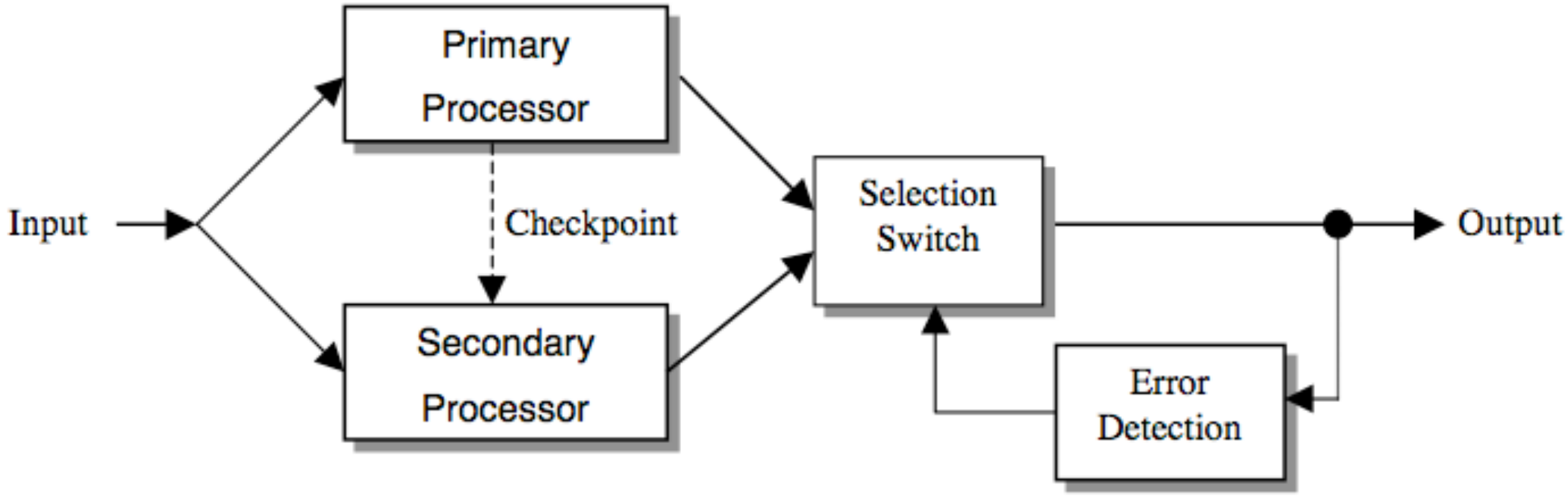
Single-Version Approaches - Checkpointing

- Save application state data at recovery points
  - Can be reloaded on crash or any other kind of data loss
  - Possible on different levels: local per process, partial, complete, distributed
- Optimum checkpointing interval
  - Checkpointing too frequent: Majority of time spent for data saving
  - Checkpointing too rare: May take long time to recover
- Several specialized solutions for C / C++ language, easier with reflection support
- Popular approach in clusters / high-performance computing
  - Latest trend: In-memory checkpointing



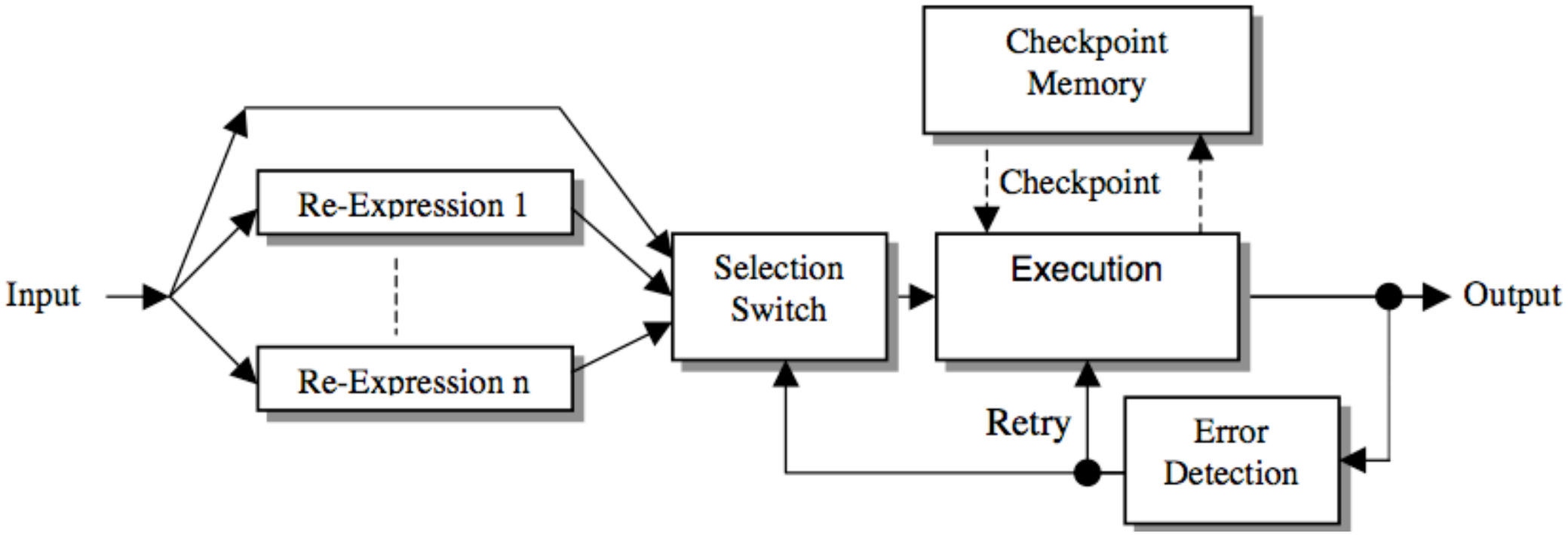
taken from
Software Fault Tolerance: A Tutorial

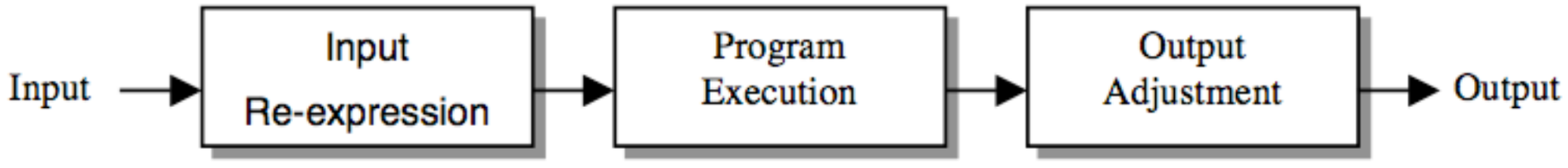
Single-Version Approaches
Process Pairs [Pradhan 96]

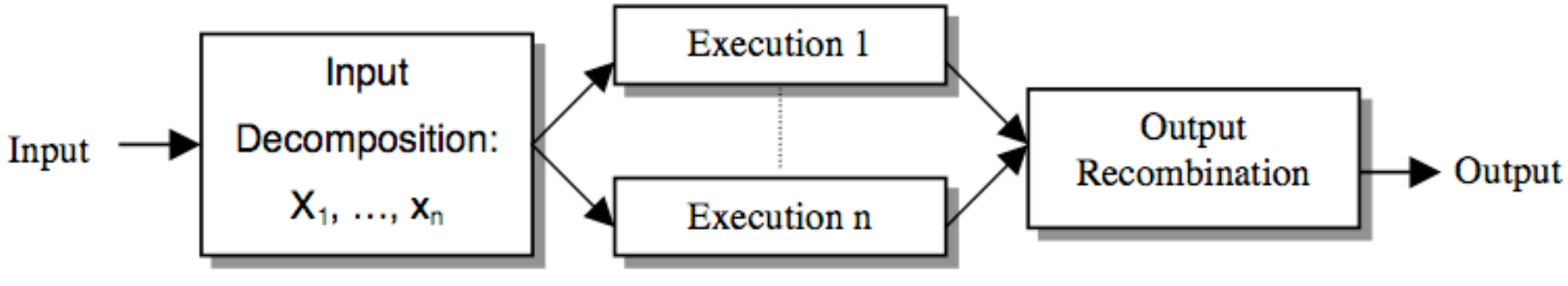


taken from
Software Fault Tolerance: A Tutorial

Single Version Approaches
Data Diversity [Ammann 88]







taken from
Software Fault Tolerance: A Tutorial

Multi-Version Approaches
Recovery Blocks

- Introduced in 1974 by Horning et. al.
  - Dynamic fault tolerance approach, related to stand-by sparing in hardware
- *Redundant* system implementations are typically used simultaneously, best answer is picked i.e. by voting
- Alternative way: Sequential execution of *recovery blocks*

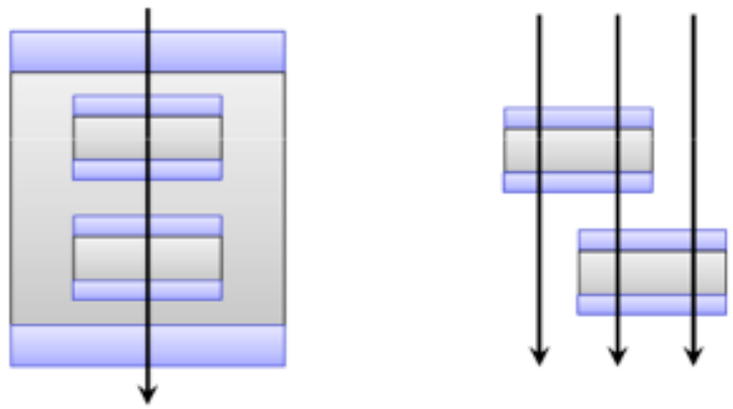
```
establish Checkpoint
ensure    Acceptance Test
by        Primary Module
else by   Alternative Module 1
else by   Alternative Module 2
else by   ...
else      Failure Exception
```

Recovery Blocks

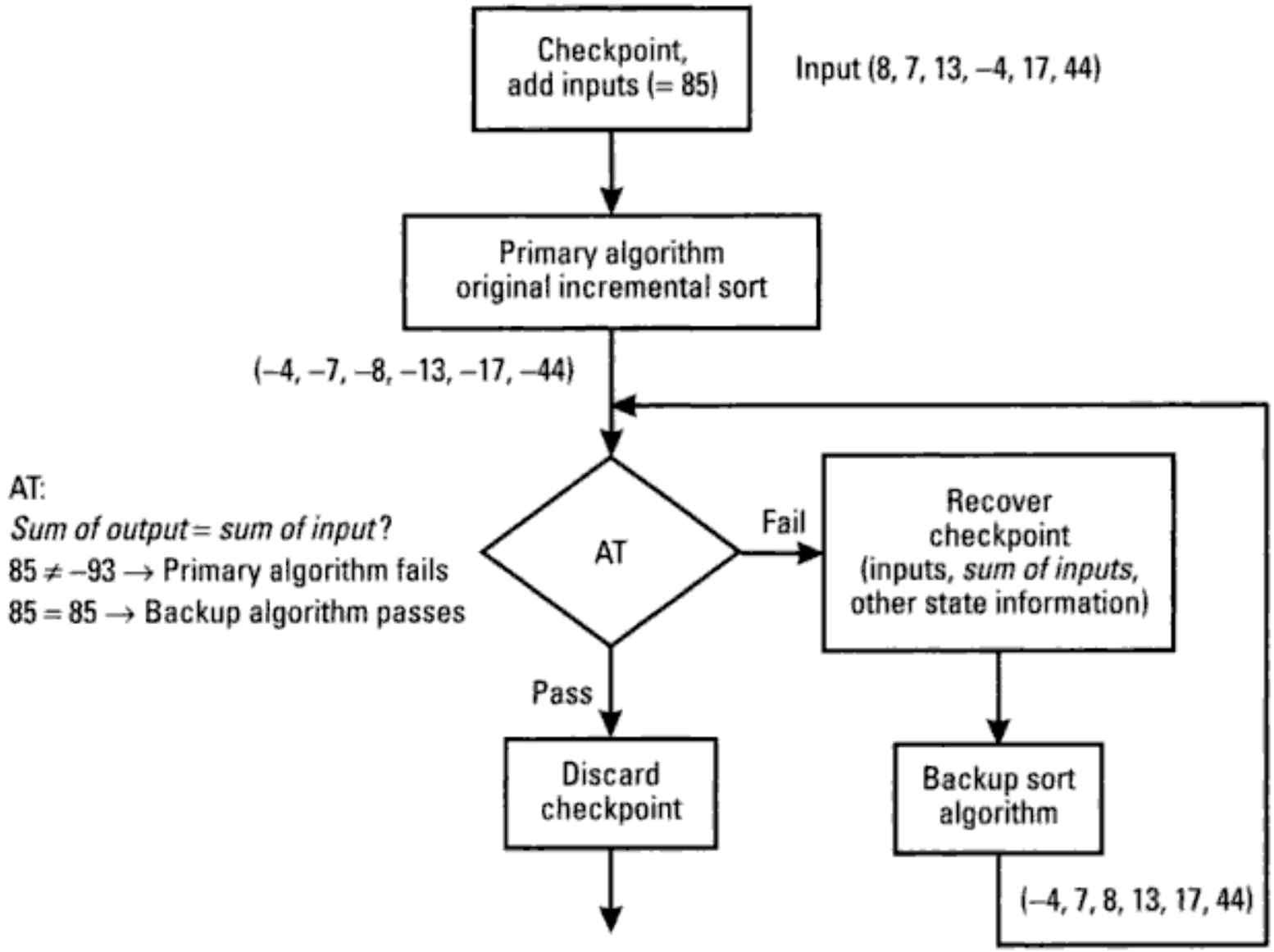
- **Primary module**
  - Debugged and tested non-redundant software, which hopefully meets specifications
  - At the beginning, checkpointing resp. **recovery cache** is filled
- **Acceptance test**
  - Reasonableness check of the calculated results
  - Acceptance tests per block, might lead to final *error handler*
  - Must be simple to not contain design faults on its own
- **Alternate module** is a standby software to be executed if primary module fails
- Possible primary / alternate module failure conditions
  - Failure of acceptance test, detected failure to terminate, implicit error detection (e.g. division by zero), failure exception of an inner recovery block

Recovery Blocks

- Limited overhead (execution only in error case), redundancy in time
- **Diversity of redundancy** implementation is relevant
  - Even results might be different, as long as they are acceptable
- Checkpoint before first block needed to ensure same preconditions
- Make successive block more simple, maybe loose parts of the result
- Might be accompanied by a Watchdog timer for some deadline support
- Problems: Shared global data, lack of alternative algorithms, added complexity
- Can be nested, can span multiple processes
- Major impact from acceptance test
  - Reasonableness checks, timing checks, reversal tests, replication tests



Example: Recovery Blocks



(C) Laura L. Pullum

Recovery Blocks - Example

- Application of concept in Naval Command and Control system software, 1985
  - 8000 lines of additional codes, utilized PDP-11 hardware extension for checkpointing
  - Failure coverage of over 70%
  - 60% addition in software implementation costs for fault-tolerant version
  - 33% extra code memory during runtime
  - 35% extra data memory
  - 40% additional run time

Extensions to Recovery Block Concept

- **Distributed Recovery Blocks** by Kane Kim
- Forward recovery scheme wit emphasis on real-time applications
- Pair of self-checking processing nodes
  - **Primary node** and **shadow node** both run a recovery block scheme
  - Two-phase structured cycle for less synchronization overhead - Input acquisition and output phase
  - Nodes use **different modules as the primary one**
  - Approach for uniform treatment of hardware and software faults
- Acceptance test first checks primary, and then shadow node

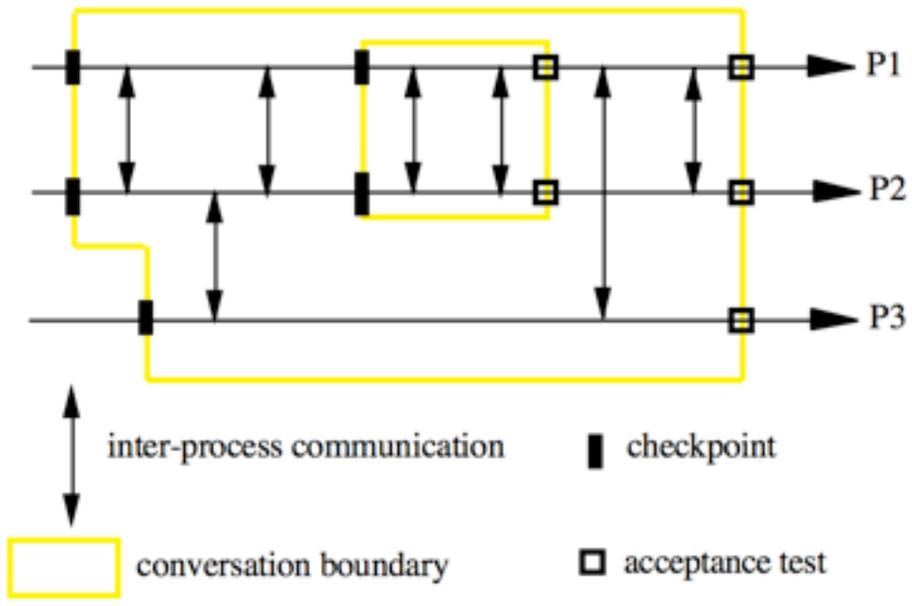
Primary Node	Backup Node
Begin the computing cycle (`Cycle`).	Begin the computing cycle (`Cycle`).
Receive input data from predecessor computing station (`Input`).	Receive input data from predecessor computing station (`Input`).
Start the recovery block (`Ensure`).	Start the recovery block (`Ensure`).
Inform the backup node of pickup of new input (`Status-1` message).	Inform the primary node of pickup of new input (`Status-1` message).
Run the primary try block (`Try`).	Run the alternate try block (`Try`).
Test the primary try block's results (`AT`). The results fail the AT.	Test the alternate try block's results (`AT`). The results pass the AT.
Inform backup node of AT failure (`Status-2` message).	Inform primary node of AT success (`Status-2` message).
Attempt to become the backup— rollback and retry using alternate try block (on primary node) using same data on which primary try block failed (to keep the state consistent or local database up-to-date). Assume the role of backup node.	Check AT result of primary node (`Check-1` message). The primary node failed. Assume the role of primary node.
	Deliver result to successor computing station (`SEND`) and update local database with result.
Test the alternate try block's results (`AT`). The results pass the AT.	Tell primary node that result was delivered (`Status-3` message).
Inform backup node of AT success (`Status-2` message).	—
Check AT result of backup node (`Check-1` message). It passed and was placed in the buffer.	—
Check to make sure the backup node successfully delivered result (`Check-2` message).	—
Backup was successful in delivering result (No `Timeout`).	—
End this processing cycle.	End this processing cycle.

Extensions to Recovery Block Concept

- **Retry blocks with data diversity [Ammann and Knight 1987]**
  - Not the algorithm is varied, but the input data
  - Easy and inexpensive to implement, since only data re-expression must be added
- Recovery for concurrent systems
  - Problem of cascaded rollbacks - **domino effect**
    - Recovery and process communication are not synchronized
    - Rollback affects connection partners and spreads out

Conversation Scheme

- **Conversation scheme [Randell et al. 1975]**
  - Processes enter a conversation asynchronously, only communication inside
  - Each process entering a conversation is check-pointed
  - Global acceptance test (conversation test line)
    - If any process detects an error, all participants must perform a rollback and user their next alternative module then
  - All processes leave the conversation together
  - Only communication inside the conversation
  - Need to prevent *information smuggling* - the altering of application state outside of the conversation



Multi-Version Approaches - N-Version Programming

- Common mode errors are only catchable by **design diversity**
- Design diversity is a complex issue
  - Design philosophies, software tools, programming languages, test philosophies
- Typical approaches try to utilize randomness - separate teams on different locations
- **N-Version Programming**
  - Suggested by Elmendorf in 1972, developed by Avizienis & Chen in 1977
  - Static approach, combination of decision mechanism and forward recovery
  - At least two independently designed and functionally equivalent variants
  - Variants are executed in parallel, decision mechanism selects the „best“ result
  - Can support reliability, but also system security against **malicious logic**

N-Version Programming

*„N-version programming is defined as the **independent generation** of N>=2 functionally equivalent programs from the **same initial specification**. The N programs possess all the necessary attributes for concurrent execution, during which **comparison vectors ("c-vectors")** are generated by the programs at certain points. The program state variables that are to be included in each c-vector and the **cross-check points ("cc-points")** at which the c-vectors are to be generated are specified along with the initial specification.*

*"Independent generation of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, **different algorithms and programming languages (or translators)** are used in each effort. The initial specification is a formal specification in a specification language. The goal of the initial specification is to state the functional requirements completely and unambiguously, while leaving the **widest possible choice of implementations to the N programming efforts**. The actions to be taken at the cc-points after the exchange of c-vectors are also specified along with the initial specification. "* (Avizienis 1977)

Comparison [Avizienis]

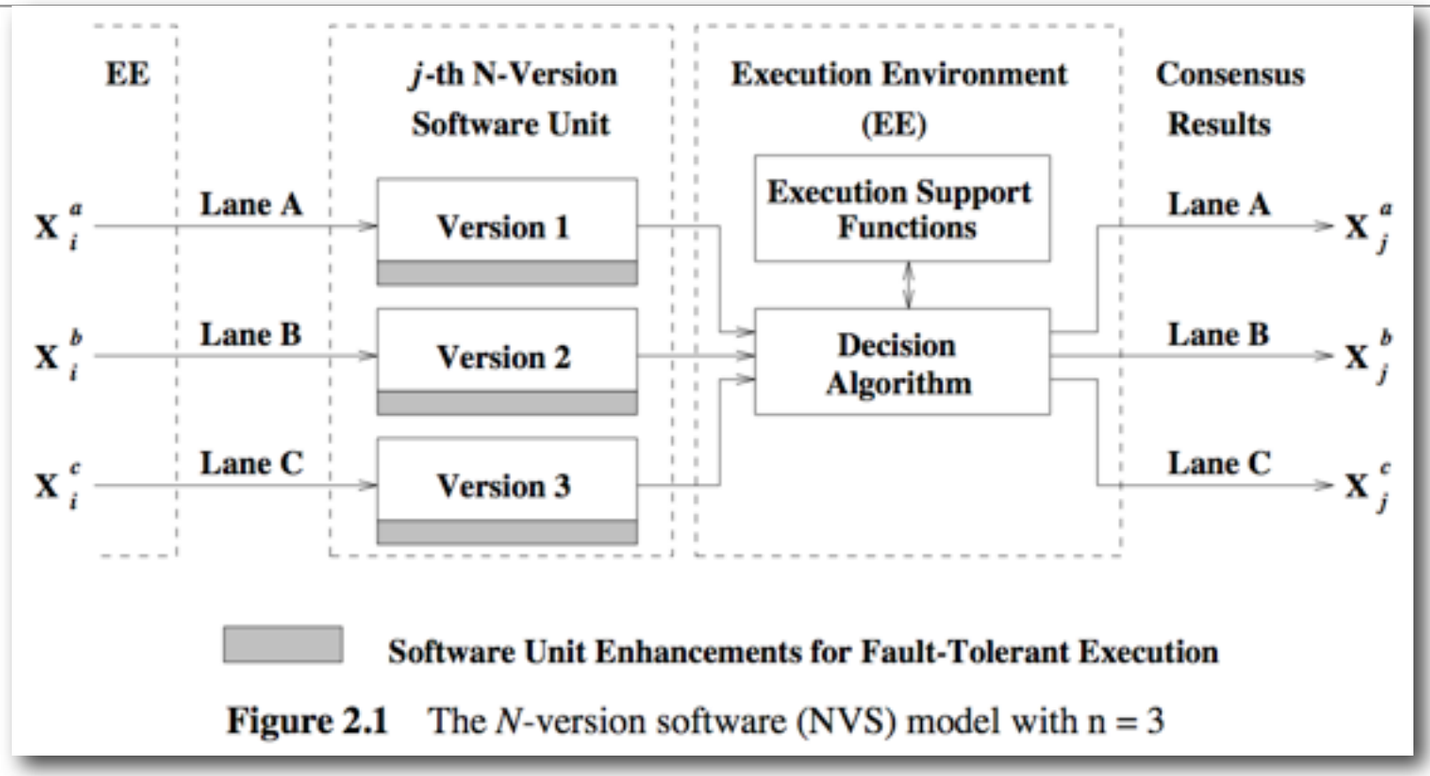


Figure 2.1 The *N*-version software (NVS) model with n = 3

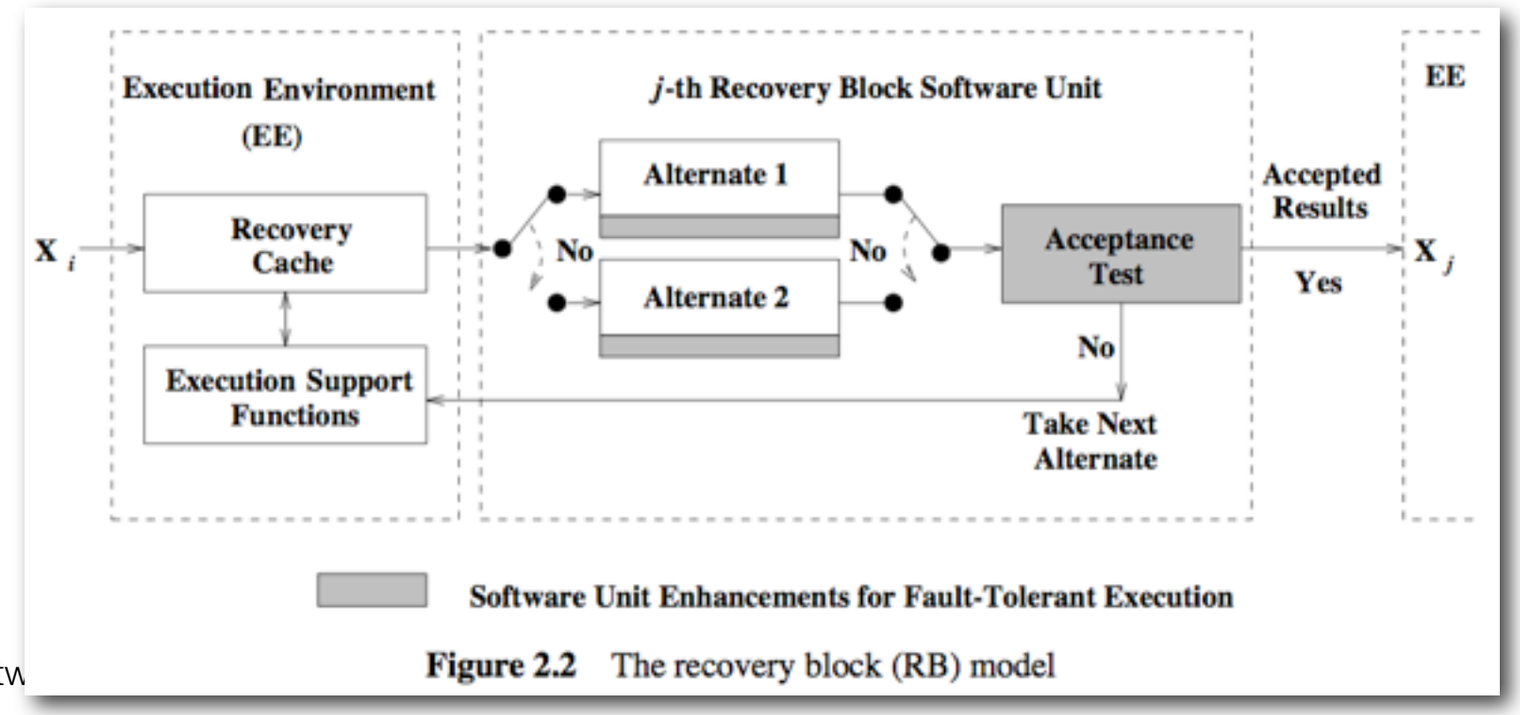
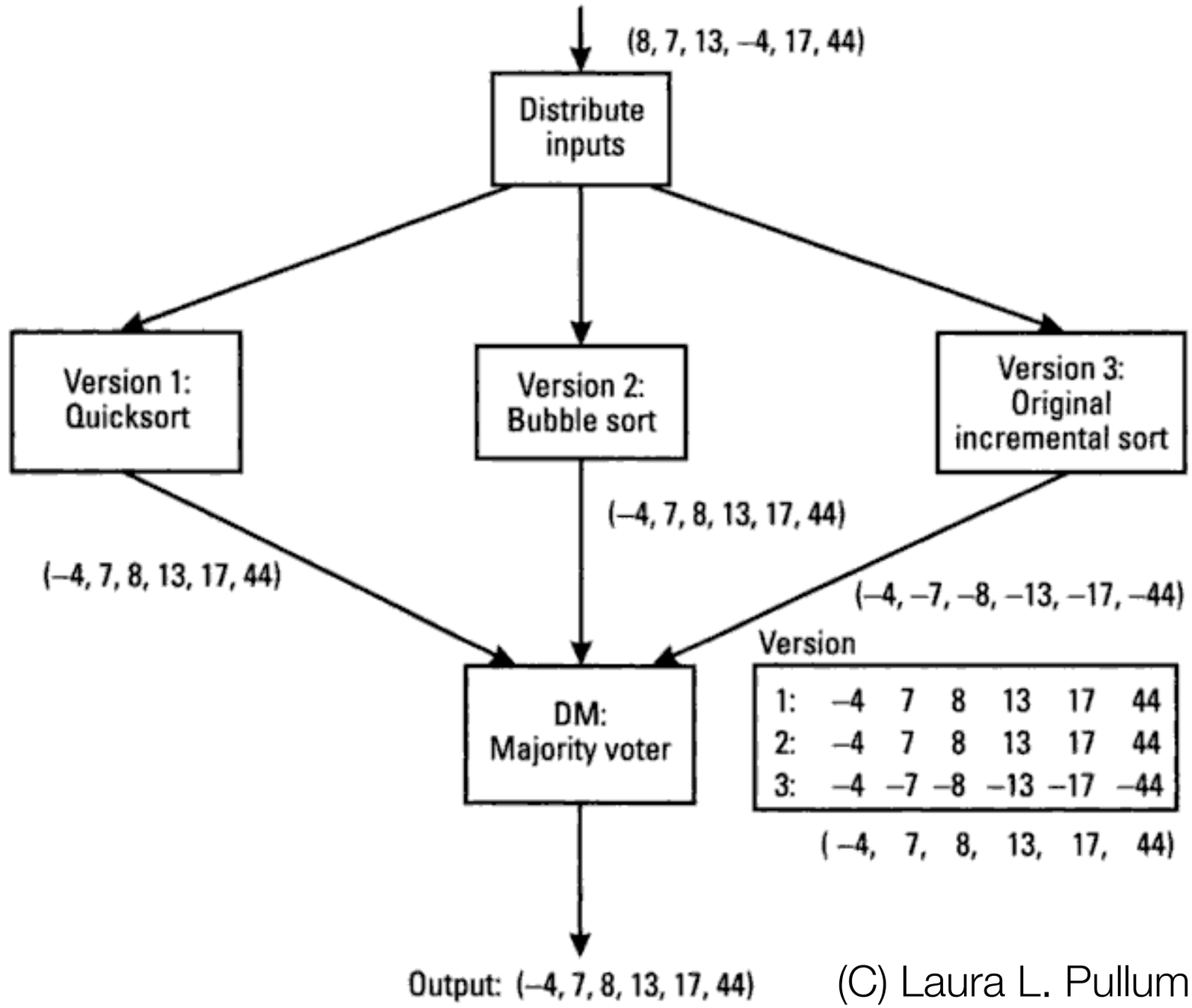


Figure 2.2 The recovery block (RB) model

Example: N-Version Programming



(C) Laura L. Pullum

N-Version Programming

- Driver resp. **control program**
  - Invokes each of the versions
  - Waits for the versions to complete their execution
  - Might also include the **decision module**
- Requirements for successful NVP, also hold for NMR
  - Consistency of all inputs and initial conditions over the modules
  - Reliable decision algorithm
- Resource and development costs / effectiveness tradeoff
  - Can be solved by software - hardware combination
  - Examples: Boeing 737-300, Airbus A-310, Airbus A-320 flight computers

N-Version Programming

- Large variety of options for the **decision module**, e.g.
  - Basic majority voter that waits for all results first
  - Perform vote whenever a new result arrives and the old vote was unsuccessful
- Method is intended for multiprocessor environments
- Attractive solution when continuity of service is a critical issue
- Relevant elements to the approach
  - Approach for initial specification and parallel development efforts (**process**)
  - Product of that process (**software**)
  - Runtime support and decision approach (**executive**)

N-Version Programming

„The second major observation concerning N-version programming is that its success as a method for on-line tolerance of software faults depends on whether the residual software faults in each version of the program are distinguishable. ***Distinguishable software faults*** *are faults that will cause a disagreement between c-vectors at the specified cc-points during the execution of the N-version set of programs that have been generated from the initial specification. Distinguishability is* ***affected by the choice of c-vectors and cc-points****, as well as by the nature of the faults themselves.*

It is a ***fundamental conjecture*** *of the N-version approach that the i****ndependence of programming efforts will greatly reduce the probability of identical software faults*** *occurring in two or more versions of the program. In turn, the distinctness of faults and a reasonable choice of c-vectors and cc-points is expected to turn N-version programming into an effective method to achieve tolerance of software faults. The effectiveness of the entire N-version approach depends on the validity of this conjecture, therefore it is of critical importance that the* ***initial specification should be free of any flaws that would bias the independent programmers*** *toward introducing the same software faults."* (Avizienis 1977)

NVP - Design Process

- Inconsistencies and omissions in the V-spec can bias the independent design efforts
- Simplex software specification tends to formulate „what“ and „how“
  - Latter part can influence diversity aspect
- Diversity can take place in
  - Training, experience and location of the developers
  - Application algorithms and data structures, testing methods and tools
  - Programming languages, software development methods and tools
  - **Random diversity** (of individuals) vs. **required diversity** (in implementation)
- **Matching features** - Needed for execution as fault-tolerant module cluster
- Distinct specifications (from one requirement document) vs. formal specification

NVP - Programming Process

- Choice of a suitable software development process for an individual version
  - Aims at maximum isolation and independence
  - Encourage greatest diversity of the implementations
  - Avoid **fault leak links** that introduce **related software faults**
- **Rules of isolation** - Ongoing process to avoid fault leak links between teams
  - **Communication and documentation protocol**
    - Supervise team information flow
    - Ensures archiving of messages for later process root cause analysis
  - **Coordinating team**

NVP - Coordinating Team

- Prepares and distributes specification and test data sets
- Sets up the communication and documentation protocol
- Supervises NVP process and the rules of isolation
- Collects and answers inquiries from teams
- Conduct formal reviews, coordinate synchronization points in the development
- Gather and evaluate all documentation, conduct acceptance tests
- All communication preferably in written format, to have fault leaks documented for post-mortem analysis

NVP - Executive

- Set of generic functions to run multiple software variants in a coordinated fashion
- Proper **matching features** are defined by the specification
- High dependability, fast operation - can be hardware, software, or combination
- Basic functions
  - Decision algorithm(s) and assurance of input consistency
  - Inter-version communication facility, enforcement of timing constraints
  - Local supervision per version
  - Global decision function for version error recovery

N-Version Programming

- Success depends on the permanent faults in each variant being distinguishable
- Variants could also be used for improved testing
  - Has danger of letting the variants progressively become closer
- Initial specification must be free of flaws, otherwise no real divergence
  - Common residual design faults
- Global data structures still need to be unified
- Choosing small modules implies [Stringini and Avizienis]
  - Frequent invocation -> low error latency, but high overhead
  - Less computation with less data on rollback, but more data to checkpoint / vote

NVP Case Study

- *A. Avizienis, M. R. Lyu, and W. Schuetz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In Digest of 18th FTCS, pages 15–22, Tokyo, Japan, June 1988.*
- University experiment with industry support for the development of an automatic aircraft landing system
  - Real specification for a flight control computer, algorithms, control laws - pitch control problem was taken out
- Three-member coordinating team, only minimal information to developers
- Independent programming teams, combination of required and random diversity
  - Different programming languages, but not different algorithms (timing, matching)
  - Procedural languages (C / Pascal), OO languages (Ada / Modula 2), logic language (Prolog), functional language (Lisp variation)

NVP Case Study

- 12 week-phase of version generation, six teams of two persons
  - Training phase - Meetings with all developers, explanation of isolation requirement
  - Design phase - Each teams discusses with domain expert and coordination team
  - Coding phase - Independent development, no communication between teams
  - Unit testing phase - Same test data for all teams
  - Integration testing phase - Same test data for all teams
  - Acceptance testing phase - Full flight simulation, iterative debugging with teams
- Communication protocol
  - Questions from programmers only to coordination team by eMail, response in 24h
  - Expert consulted by coordination team

NVP Case Study

- 120 questions from six teams, 30 answers broadcasted
- First results after acceptance test
  - Number of lines with / without comments, number of executable or arithmeitc statements, number of modules, mean number of statements per module

Metrics	ADA	C	MODULA-2	PASCAL	PROLOG	T
LINES	2253	1378	1521	2234	1733	1575
LN-CM	1517	861	953	1288	1374	1263
STMTS	1031	746	546	491	1257	1089
MODS	36	26	37	48	77	44
STM/M	29	25	15	10	16	25

NVP Case Study

Test Phase	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Coding/Unit Testing	2	4	4	10	15	7	42
Integration Testing	2	5	0	2	7	4	20
Acceptance Testing	2	4	0	0	4	10	20
Total	6	13	4	12	26	21	82

Table 2: Fault Classification by Phases

Fault Class	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Typographical	0	1	0	0	9	0	10
Omission	1	3	0	0	8	5	17
Unnecessary Code	1	0	0	2	0	2	5
Incorrect Algorithm	3	5	2	6	9	13	38
Spec. Misinterpretation	1	3	1	4	0	1	10
Spec. Ambiguity	0	1	0	0	0	0	1
Other	0	0	1	0	0	0	1
Total	6	13	4	12	26	21	82

Compiler bug

Table 3: Fault Classification by Fault Types

NVP Case Study

- Only one identical fault, based on mis-read specification text
- All cross-checks and recovery point routines written in C
  - Additional interoperability problem with diverse languages
- Evaluation performed by requirements-based stress testing and structural analysis
- One problem caused by unconsidered late specification update
- Severe structural faults by **underground variables**
  - Introduction of new, unspecified state variables that are not considered in the cross-check
- Some implementations did not consider corrected results from the decision module

NVP Case Study

- Conclusion
  - Original specification contain too much implementation hints for diversity
  - Order of computations has strong impact on diversity possibilities
  - Use of different programming languages supports team isolation
  - Failure to follow NVP design rules lead to some structural faults
  - Similar and time-coincident errors were rare

NVP Independence Evaluation

- *John C. Knight and Nancy G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming.*
- NVP relies on assumption that independent software version fail independently
  - Faults occur at random and are unrelated
  - Probability of common mode failures (identical incorrect output) assumed small
  - Additional costs for multiple versions would be offset by reduced validation costs
  - But: Even in mechanical systems, common design faults are a serious threat
- NVP used in crucial systems
  - Slat and flap control system of the Airbus A310
  - Point switching, signal control, and traffic control in Gothenburg railways

NVP Independence Evaluation

- Experiment
  - Graduate students from two universities, 27 programs
  - Requirement specification and golden unit from previous NVP experiment
  - No development methodology imposed, predefined language and system
  - No restrictions on reference and documentation sources
  - Fifteen input data sets with given output, for debugging
  - 200 randomly generated test cases for acceptance test, different sets per version to avoid ‚filtering‘ of common mode failures by the acceptance test
  - Final examination with one million random tests, comparison of version result with golden unit result

NVP Independence Evaluation

- Approx. one half of total software faults involved two or more versions
- All common faults involved students from different schools
- Examples for correlated faults across versions
  - Misunderstandings in numerical analysis (angle comparison)
  - Misunderstandings in geometry (not considering special cases)
- Some conclusion
  - Common mode faults are application-specific, so NVP might be still very valuable
  - Certain parts of any problem are just more difficult
    - Semantic aspects, human misconception, missing details in the specification
  - Unique faults tend to be more likely detected by compilers / testing

Comparison of Approaches

- *Timothy J. Shimeall and Nancy G. Leveson. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. February 1991*
- Comparison of five software fault detection approaches
  - Run-time assertions
  - N-version programming resp. back-to-back testing
    - Vote is used as test oracle
    - Varying specification language, development practices, languages
  - Functional and structural testing
  - Code reading by stepwise abstraction (without comments to avoid biasing)
  - Static data-flow analysis with pre-defined algorithms

Comparison of Approaches

- Experiment setup
  - Set of programs written from a single specification for combat simulation
  - Tree-step data transformation, 2600-4500 input data values
  - Senior-level students for software engineering, teams of two persons
    - One set performed architectural designs, coding, and debugging to pass pre-defined acceptance test
    - Disjoint set of students to detect faults in the programs
- Report generation per fault, administrator acts as final arbiter for false alarms
- Acceptance test designed to execute each of the major code portions at least once

Comparison of Approaches

- Comparison between fault elimination and fault tolerance
  - Voting tolerates faults, assertions have the potential
  - Claim by Avizienis et al. - Multiversion might reduce the need for testing
  - Are the same faults detected by fault elimination and voting ?
  - Is a particular testing type irrelevant when voting is used ?
  - Difficult comparison: One tolerated error condition does not mean that all error conditions from this fault are handled
  - Results
    - 67 faults handled only by voting, 28 only by assertions, 119 only by fault elimination; 27 detected by both voting and fault elimination
    - On average, the voting triplets only tolerated 38% of the failures

Comparison of Approaches

- Comparison of testing techniques with respect to fault detection
  - Data shows that most of the faults detected by each technique were not found by no other technique
  - Code reading
    - Found incorrect formulas, missing checks, bad conditions on branches, missing condition checks for special cases
    - Did not find any globally missing code or missing application logic
    - False alarms: From code that was difficult to abstract, from inconsistent implementation strategies (e.g. variable naming), from syntactical focus
  - Static data flow analysis
    - Found only initialization faults, but is cheap

Comparison of Approaches

- Voting
  - Found missing paths in the program, parameter ordering flaws, wrong variable usage, wrong ordering of operations
  - Missing check of specific conditions not found by voting approaches, more success with boundary test cases
- Run-time assertions
  - Found parameter ordering flaws, wrong variable usage
  - No detection of missing code flaws
  - Simple range checks detected 23 flaws not detected by any other approach

Comparison of Approaches

- Functional and structural testing
  - Found wrong ordering of operations, missing check of specific conditions, missing functionality or missing single program paths
  - Incompleteness through module-by-module testing approach
    - Best effectiveness with atypical data sets
- Variation in effectiveness mostly reasoned by
  - Ability to examine internal state (problem for voting)
  - Scope of evaluation (problem for assertions / code reading)

Comparison of Approaches

Class	Comments	Detecting Technique
Overrestriction	e.g., forcing all weather to move northeast, rejecting legal input	Assert, Read, Test, Vote
Loop Condition	e.g., infinite loop	Vote, Assert, Test
Calculation	Incorrect formula	Read
Initialization	Variable not initialized	Stat. Analysis, Test
Substitution	Wrong variable used	Vote, Assert
Missing Check	Exceptional case not handled e.g., divide by zero	Read
Branch Condition	Bad condition on a branch	Vote, Read, Test
Missing Branch	Localized missing code to detect and handle specific conditions in normal execution	Read, Test
Missing Thread	Missing path throughout program	Vote, Test
Unimplemented Requirement	Missing functionality on all paths	Test
Ordering	Operations in wrong order (e.g., updating value before use)	Vote, Test
Parameter Reversal	Actual parameter order permuted with respect to formal parameter	Vote, Assert
Data Structure	e.g., linked list becomes circular	Vote, Test, Read, Assert

Damage Confinement

- Easier in hardware, due to physical isolation; Software has shared resources
- Basic principle: Interacting components have to be mutually suspicious
- Constraints on interactions are typically very implementation-specific
- Some generic models, based on security interaction work
  - Bell-LaPadula model - Information flows to higher levels, but not vice versa
    - Reformulation based on layer integrity demands
    - Back-flow with heavy checking typically allowed
- Damage flows that bypass intended interactions are another issue
  - Strongly typed languages, operating system address spaces, sandboxing, reference monitor that mediates all data accesses

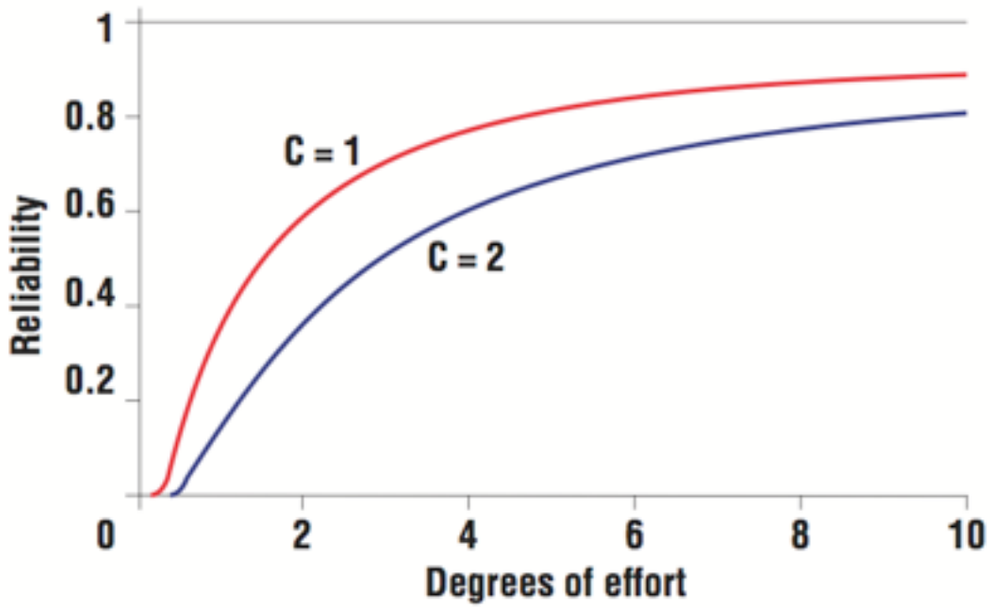
Damage Confinement Strategies

- Checking error conditions at interface
- Law-Governed Systems [Minsky91]
  - Constraints on interactions are externalized, set of rules enforced at run-time
  - Specialized approaches for message exchange between objects
    - Static vs. dynamic enforcement mechanisms, power depends on language
    - Rule-enforcement facilities might have their own state:
      - „Method X of object Y is only invokable if this process holds a token.“
- Voting
  - High costs, problems with stateful functionality

An Alternative: Simplex Approach

- Approach by Lui Sha, University of Illinois at Urbana-Champaign
- Where to spend the money: Multiple diverse versions, or one ultra-robust version ?
  - Relationship between reliability, development effort, and code complexity
  - Proposed model: Failure rate is proportional to software complexity *C* and inversely proportional to development effort *E*
- Example: Assuming t=1
  - Decreasing rate of reliability improvement for increasing effort
  - With more complexity comes higher effort for same reliability
  - Effort *E* == costs, typically constant

$$R(t) = e^{-\lambda t} = e^{-Ct/E}$$



Simplex

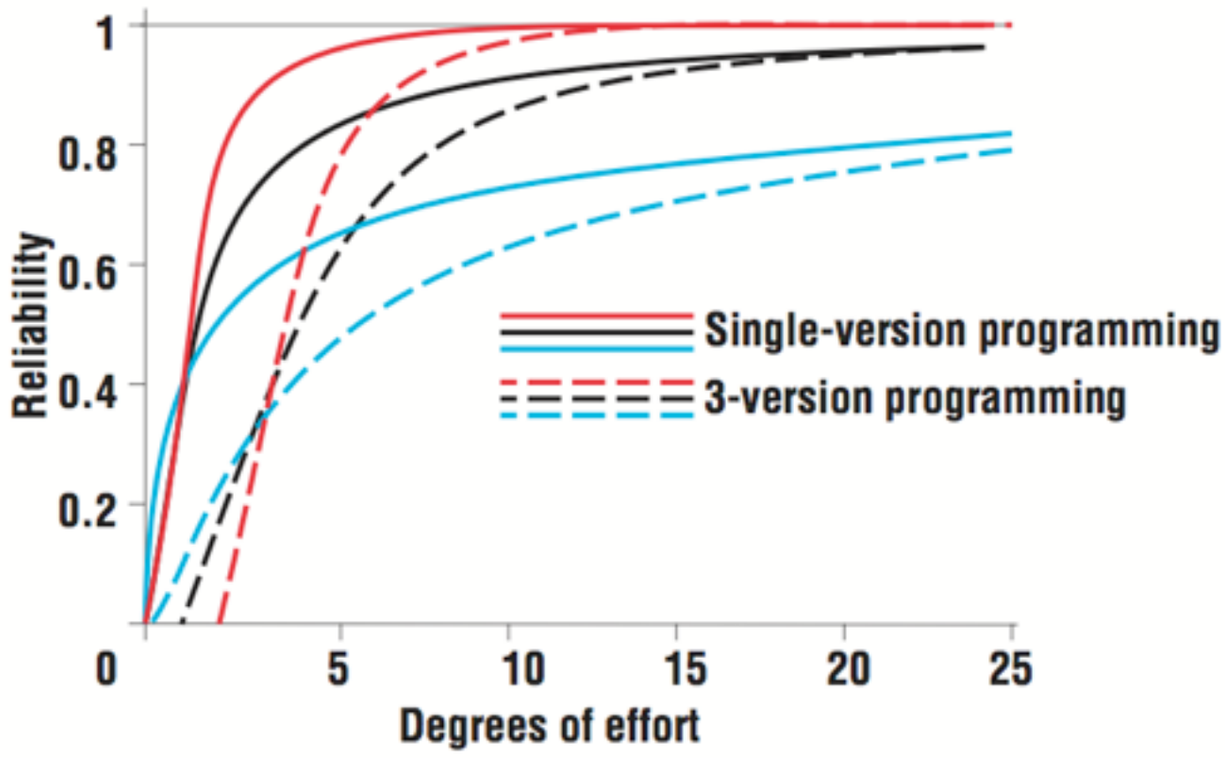
- N-Version Programming
  - Total effort *E*, divided by three teams; assuming C=1

$$R_{NVP} = R_M^3 + 3R_M^2(1 - R_M) \qquad R_M = e^{-3/E}$$

  - Optimistic assumption: Failure rate is inversely proportional to square of software development effort (red)

- Pessimistic assumption: Failure rate is inversely proportional to square root of software development effort (blue)

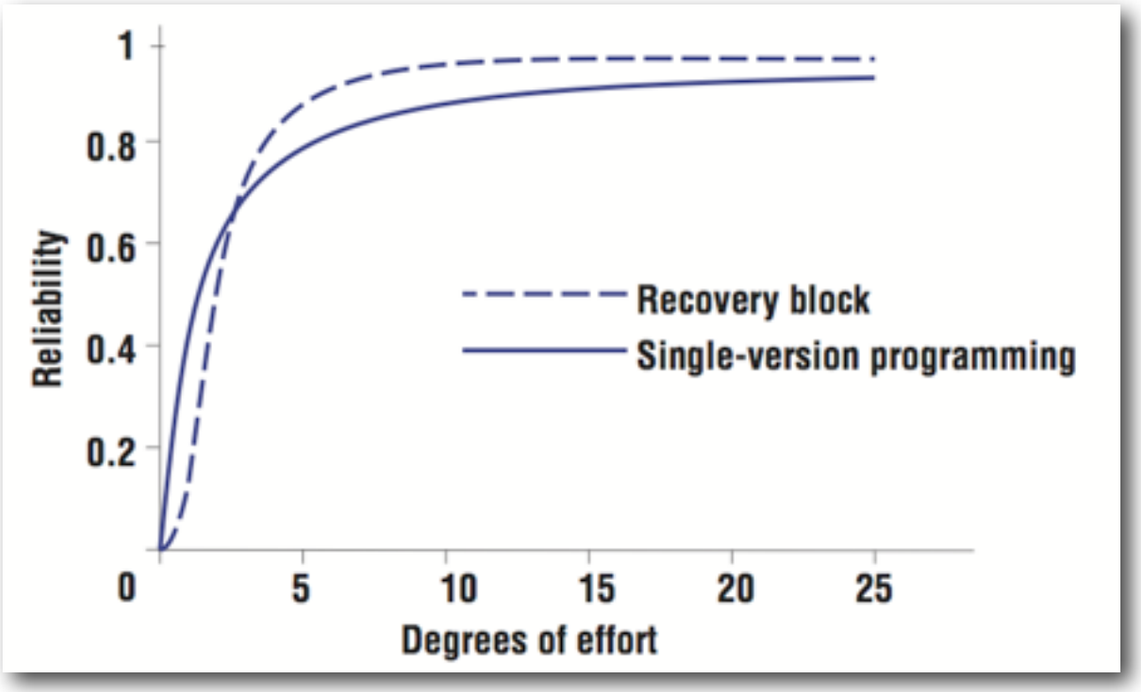
- Single-version approach always outperforms NVP
  - But multiple versions might be obtainable much cheaper



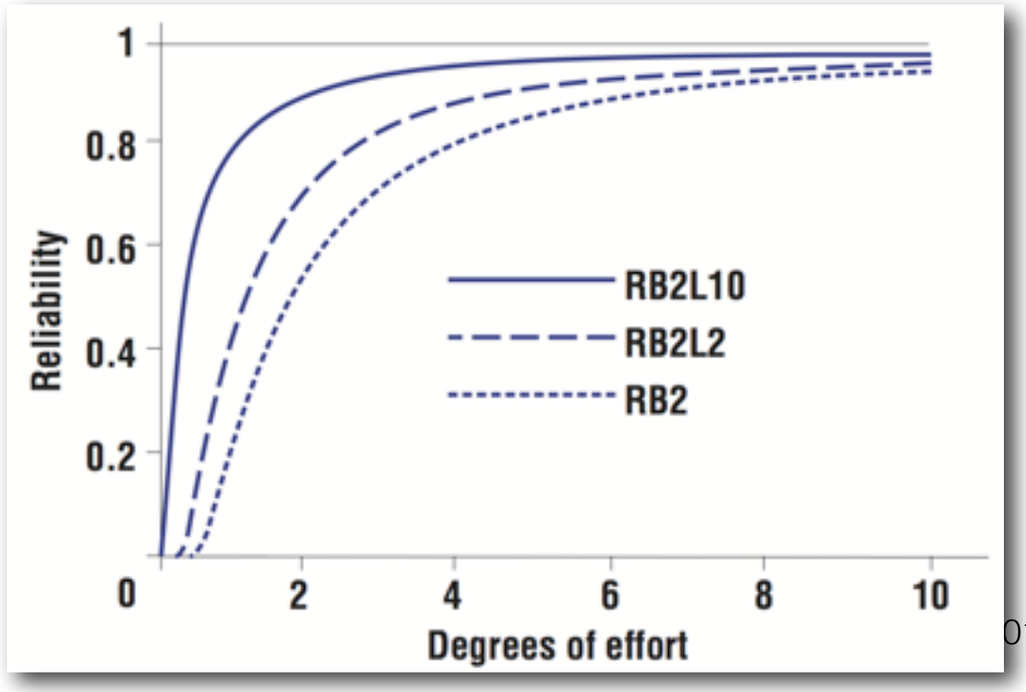
Simplex

- Recovery blocks
  - System works as long as any alternative works (perfect acceptance test)
  - For three alternatives: $R_{RVB} = 1 - (1 - R_M)^3, R_M = e^{-3/E}$

With three-way divided effort, some minimum effort is needed to get better reliability by recovery blocks

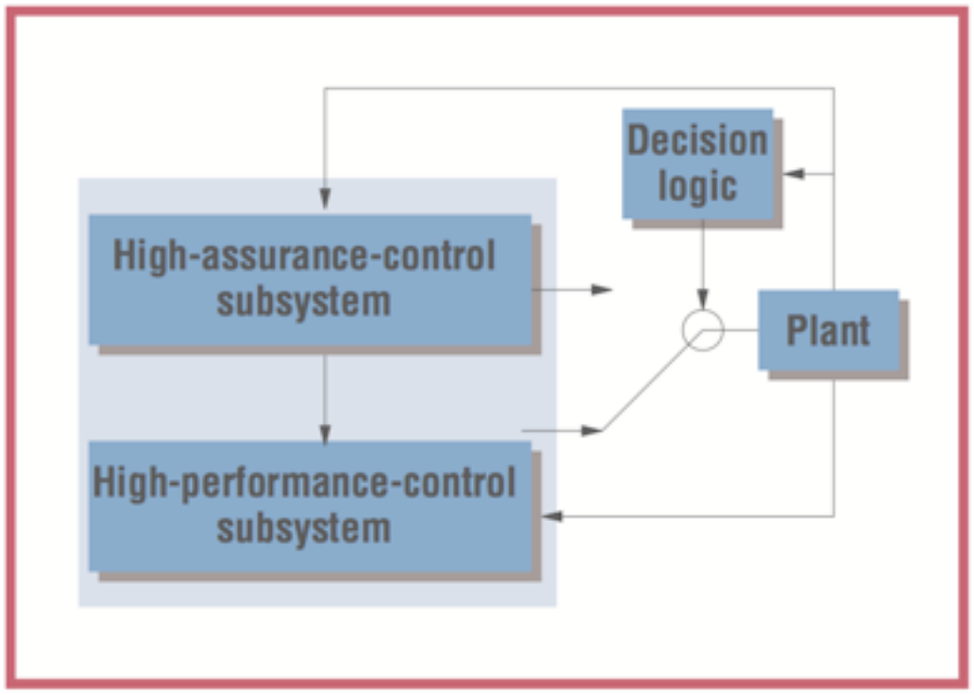


With two alternatives, complexity reduction of alternative (by factor 2 resp. 10) brings huge improvement for the same effort



Simplex

- Idea: Using simplicity to control complexity
  - Forward recovery approach, based on feedback loop
  - **HAC subsystem** - Simple construction, formal methods, reliable hardware
  - **HPC subsystem** - Complex technology, advanced features
- HPC can use HAC output, but not vice versa
- Decision logic based on control loop output
- Typically performance degredation with HAC
- Example: Boeing 777 primary and secondary flight controller



Software Dependability

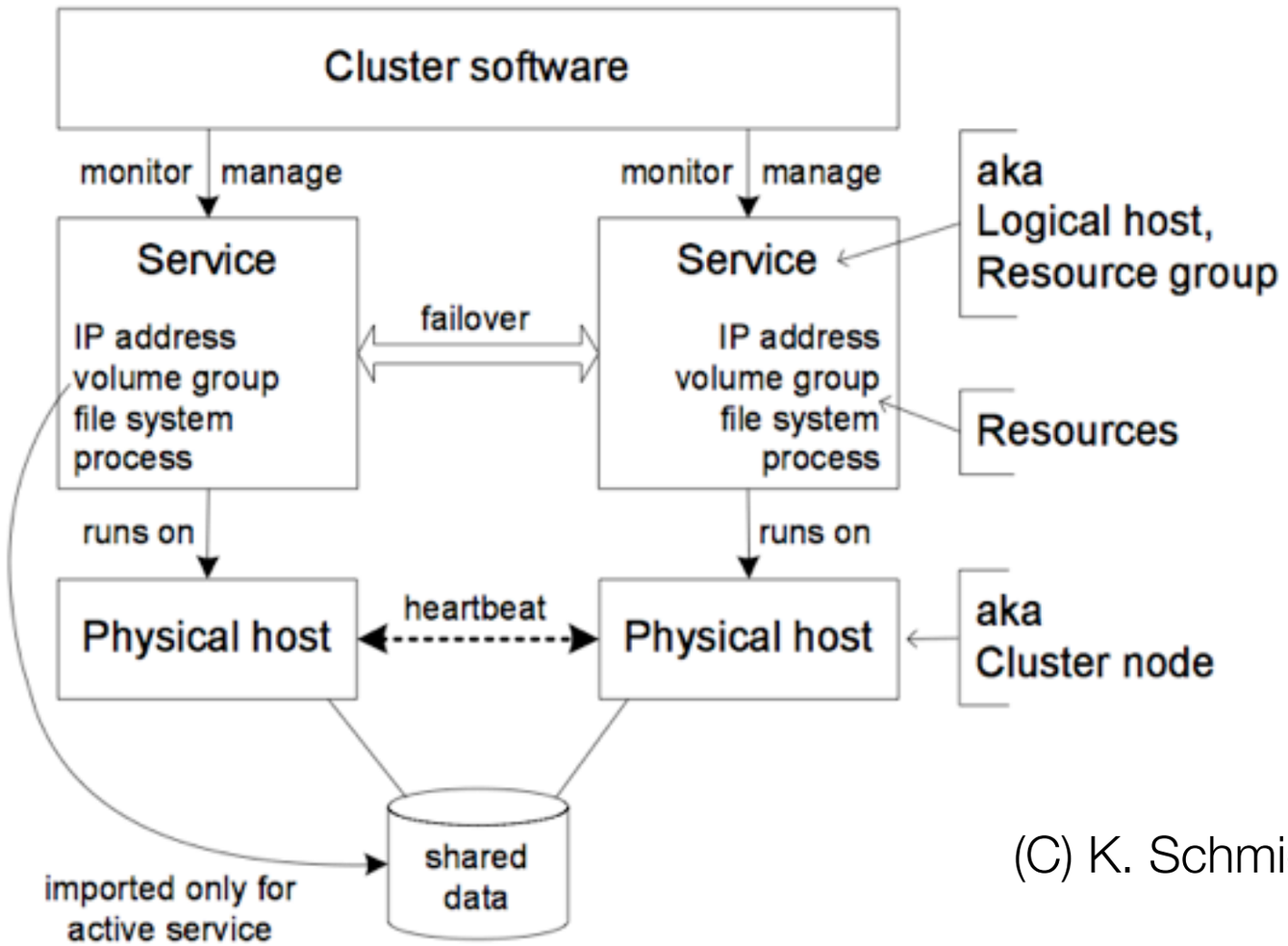
- Software testing
  - Reduce number of dormant faults at development time
- Fault-tolerant software
  - Techniques to achieve fault tolerance for software faults
  - Application of redundancy idea to software modules
- **Software fault tolerance**
  - Techniques to achieve fault tolerance by software mechanisms
  - Typically for hardware failures on lower levels in the system stack
  - Redundancy managed by operating system, cluster framework, application code

Software Fault Tolerance

- Can be realized on different levels
  - Operating System - Failover and / or load-balancing cluster framework
  - Database, middleware stack
  - Application itself
  - Combination of the above methods
- Maintenance is an issue
  - Failover success rate above 90% is understood as good value
  - Understanding of a „major outage“ is driven by SLA
  - Crashes can leave garbage behind
- New wave with fault tolerance through virtualization technologies

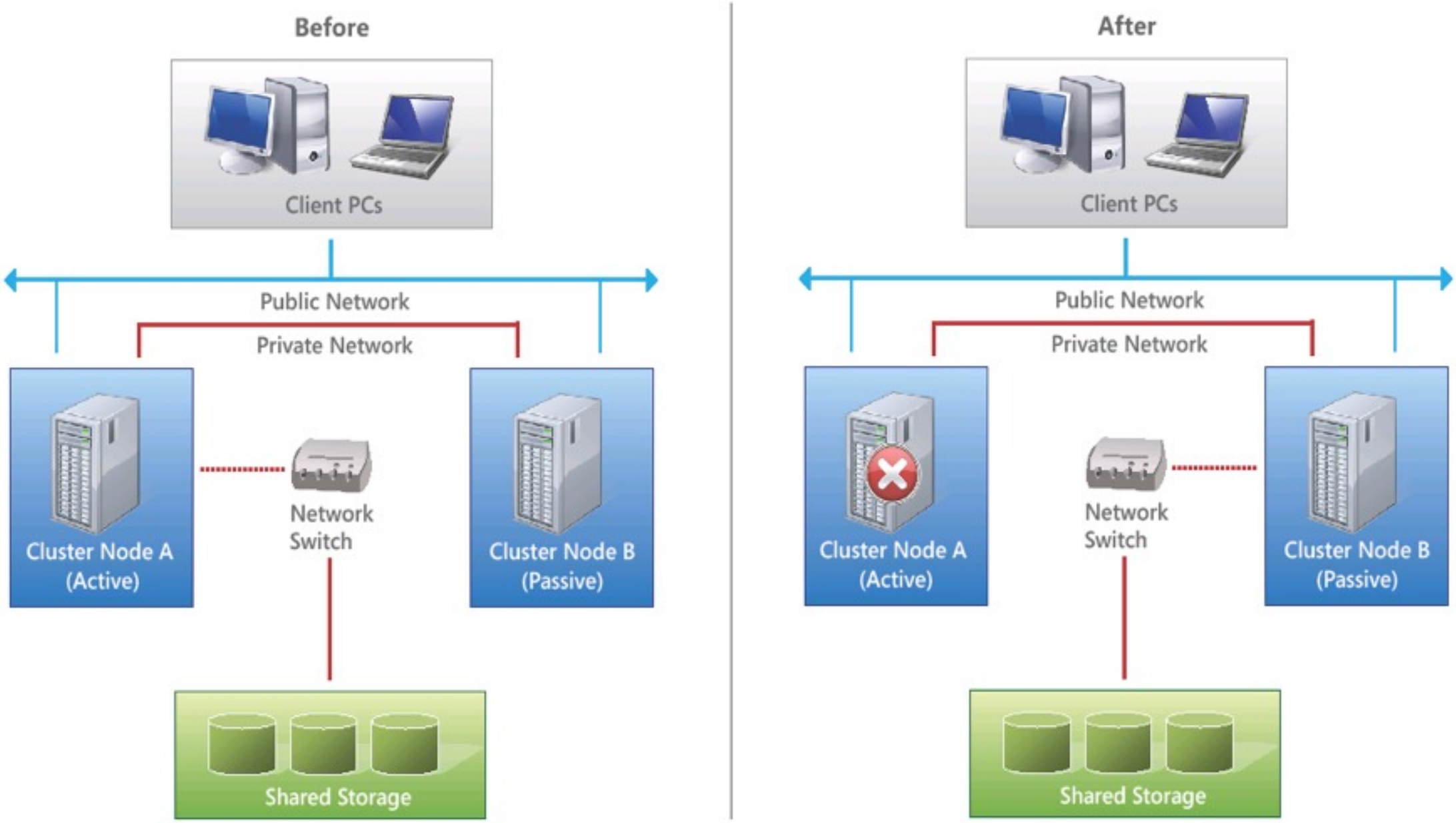
Operating System

- Basic approach: Host clustering
  - Does not know about session state or transactional behavior
  - Only concerned with overall availability of services, not their individual state
  - Typically issues with proper service deactivation (e.g. mount)
  - Logging, packaging
  - Major improvements with latest virtualization technologies



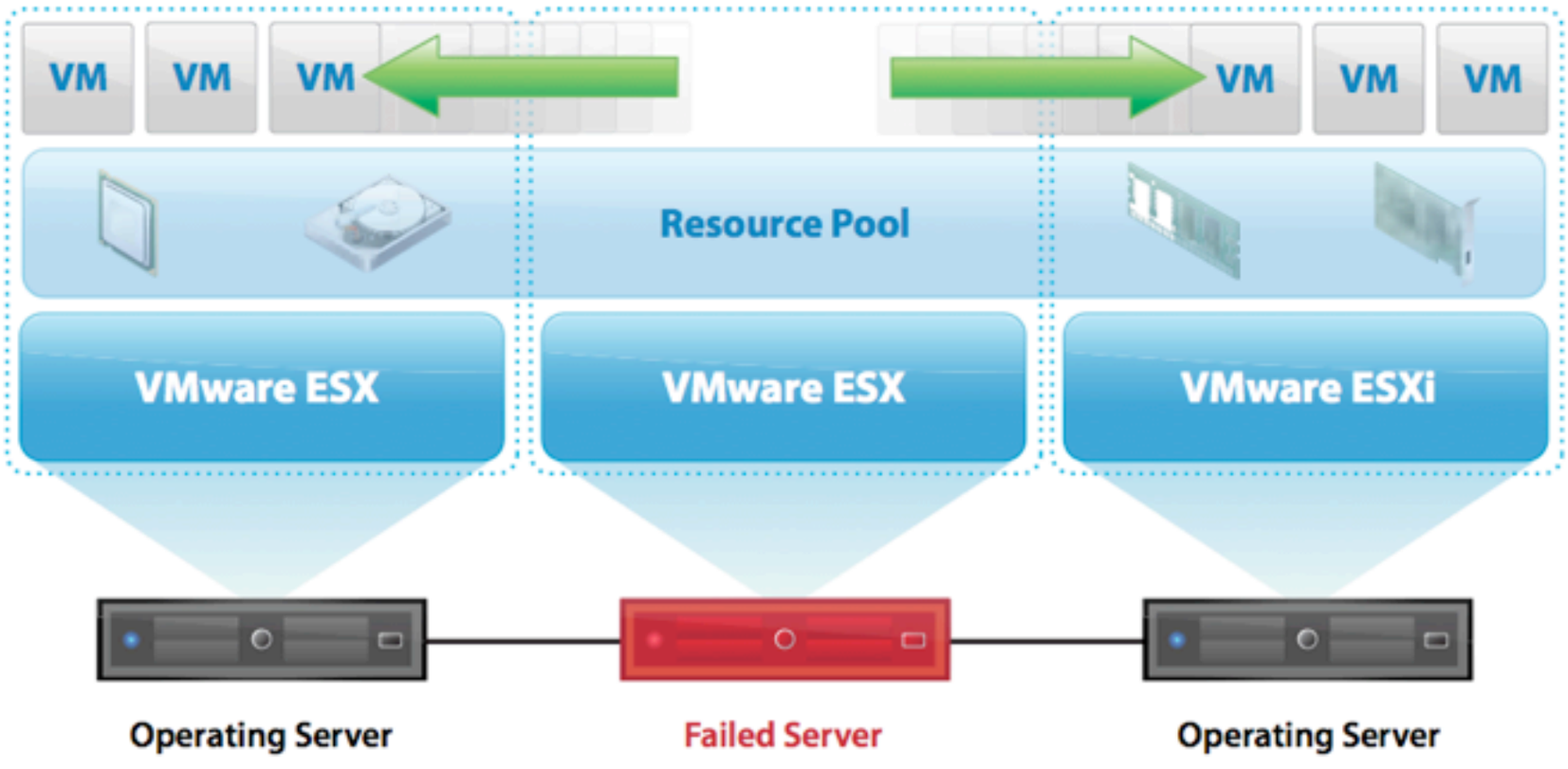
(C) K. Schmidt

Example: Operating System Clustering



VMWare HA

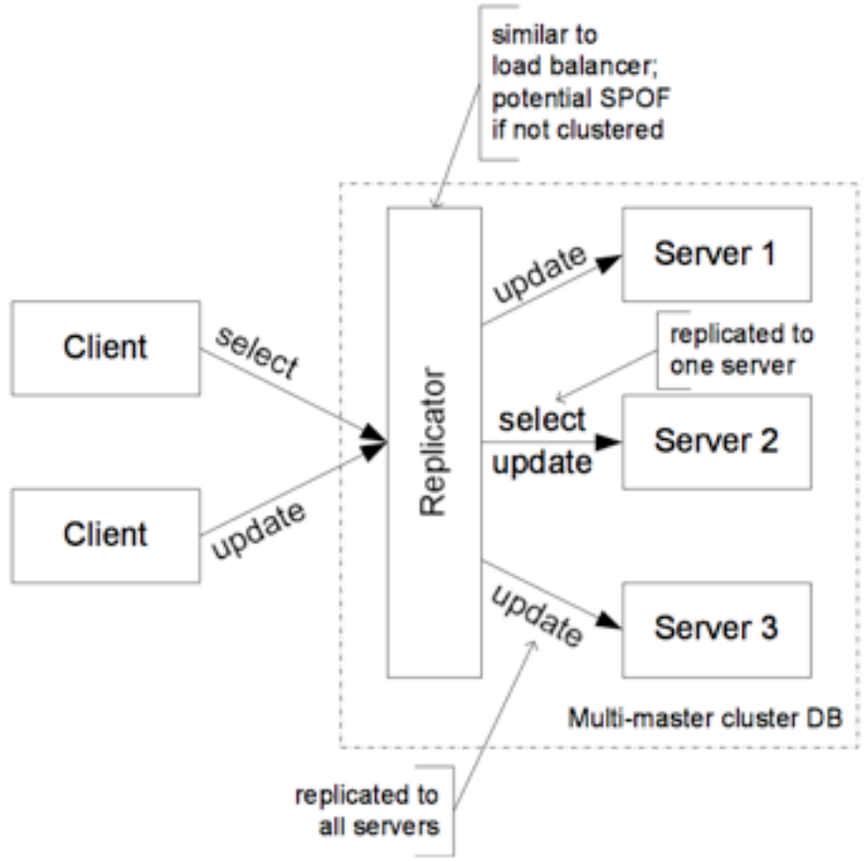
- On failure of physical host, virtual machine is automatically restarted on another host
- Distributed scheduling ensures always available resources for failover case
- Based on mutual hardware and shared storage (e.g. iSCSI)



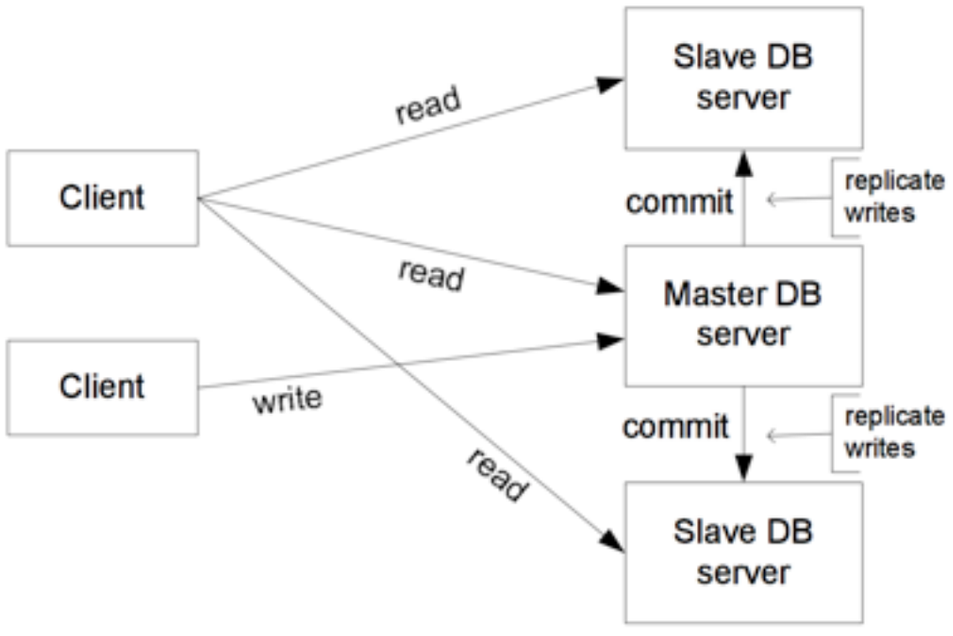
from www.vmware.com

Databases

- Databases - ACID principle
  - Atomicity and durability by write-ahead logs
    - Contain data value before (for rollback) and after (for recovery) the modification
    - Written synchronously, so that actual data operations can be heavily buffered
    - Can also be used for database synchronization
- Classical shared-disk vs. shared-nothing discussion



Multi-Master Database Cluster



Master-Slave Setup