TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Arvutitehnika instituut
ati.ttu.ee

IAF0030

Arvutitehnika erikursus I

Loeng 3
Safety, Fault Tolerance, Software Testing

Gert Jervan

Tallinna Tehnikaülikool
Arvutitehnika instituut

Lecture 2

- ✓ Safety Requirements
- ✓ Hazards
- ✓ Hazard Analysis
- ✓ Risks
- ✓ Risk Analysis
- ✓ Risk Management
- ✓ Safety & SILs
- ✓ Risk Reduction & Design

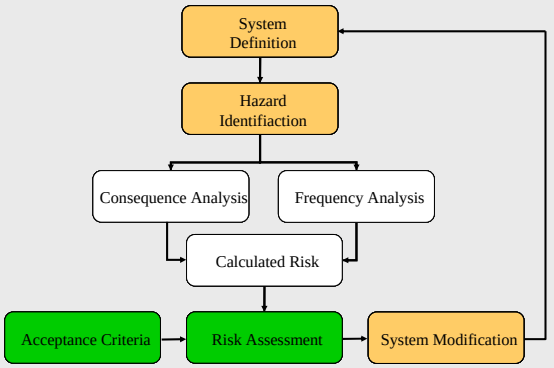
Lecture Outline

- ✓ Safety Requirements
- ✓ Hazards
- ✓ Hazard Analysis
- ✓ Risks
- ✓ Risk Analysis
- ✓ Risk Management

Lecture 2

- ✓ Safety & SILs
- ✓ Risk Reduction & Design
- ✓ Software Testing

Hazard and Risk Analysis Process



Safety Requirements

- ✓ Once hazards are identified and assessed, safety requirements are generated to mitigate the risk
- ✓ Requirements may be
  - primary: prevent initiation of hazard
    - eliminate hazard
    - reduce hazard
  - secondary: control initiation of hazard
    - detect and protect
    - warn
- ✓ Safety requirements form basis for subsequent development

Safety Integrity

- ✓ Safety integrity, defined by
  - Likelihood of a safety-related system satisfactorily performing the required safety functions under all stated conditions within a stated period of time
  - Hardware integrity, relating to random faults
  - Systematic integrity, relating to dangerous systematic faults
- ✓ Expressed
  - Quantitatively, or
  - As Safety Integrity Levels (SILs)
- ✓ Standards, IEC 1508, 61508
  - Define target failure rates for each level
  - Define processes to manage design & development
- ✓ Aims to deal with systemic failures

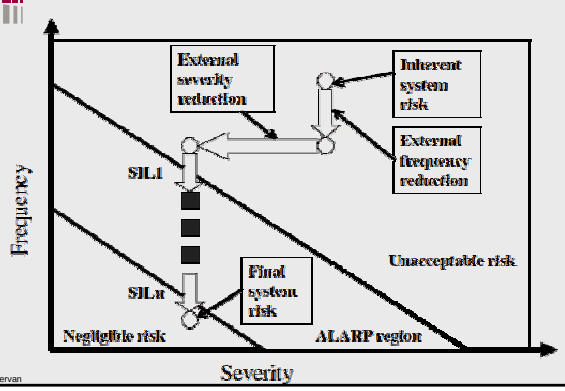
Safety Integrity Levels (SILs)

- Tolerable failure frequency are often characterised by Safety Integrity Levels rather than likelihoods
  - SILs are a qualitative measure of the required protection against failure
- SILs are assigned to the safety requirements in accordance with target risk reduction
- Once defined, SILs are used to determine what methods and techniques should be applied (or not applied) in order to achieve the required integrity level
- Point of translation from failure frequencies to SILs may vary

Automotive SIL

- Uncontrollable (SIL 4), critical failure
  - No driver expected to recover (e.g. both brakes fail), extremely severe outcomes (multiple crash)
- Difficult to control (SIL 3), critical failure
  - Good driver can recover (e.g. one brake works, severe outcomes (fatal crash)
- Debilitating (SIL 2)
  - Ordinary driver can recover most of the time, usually no severe outcome
- Distracting (SIL 1)
  - Operational limitations, but minor problem
- Nuisance (SIL 0)
  - Safety is not an issue, customer satisfaction is

Risk & SILs



IEC 61508 Standard

- New main standard for software safety
- Can be tailored to different domains (automotive, chemical, etc)
- Comprehensive
- Includes SILs, including failure rates
- Covers recommended techniques
- IEC = International Electrotechnical Commission
- E/E/PES = electrical/electronic/programmable electronic safety related systems

Safety-Integrity Table of IEC 61508

Safety Integrity Level	Low demand mode of operation (Average probability of failure to perform its design function on demand)	
4	$\geq 10^{-5}$ to $< 10^{-4}$	(> 99.99 % reliable)
3	$\geq 10^{-4}$ to $< 10^{-3}$	(> 99.9 % reliable)
2	$\geq 10^{-3}$ to $< 10^{-2}$	(> 99% reliable)
1	$\geq 10^{-2}$ to $< 10^{-1}$	(> 90% reliable)

Safety Integrity Level	High demand mode or continuous mode of operation (Probability of dangerous failure per hour)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

- The higher the SIL, the harder to meet the standard
- High demand for e.g. car brakes, critical boundary SIL 3
- Low demand for e.g. airbag, critical boundary is SIL 3, one failure in 1000 activations

SILs

- SILs 3 and 4 are critical
- SIL activities at lower levels may be needed
- SIL 1
  - Relatively easy to achieve, if ISO 9001 practices apply,
- SIL 2
  - Not dramatically harder than SIL 1, but involves more review and test, and hence cost
- SIL 3
  - Substantial increment of effort and cost
- SIL 4
  - Includes state of the art practices such as formal methods and verification, cost extremely high

Techniques and Measures

Clause 7.7 : Software Safety Validation					
TECHNIQUE/MEASURE	**Ref**	**SIL1**	**SIL2**	**SIL3**	**SIL4**
1. Probabilistic Testing	B.47	--	R	R	HR
2. Simulation/Modelling	D.6	R	R	HR	HR
3. Functional and Black-Box Testing	D.3	HR	HR	HR	HR
NOTE: One or more of these techniques shall be selected to satisfy the safety integrity level being used.					

- ✓ Implementing the recommended techniques and measures should result in software of the associated integrity level.
- ✓ For example, if the software was required to be validated to be of Integrity level 3, Simulation and Modelling are Highly Recommended Practices, as is Functional and Black-Box Testing.

Detailed Techniques and Measures

- ✓ Related to certain entries in these tables are additional, more detailed sets of recommendations structured in the same manner. These address techniques and measures for:
  - Design and Coding Standards
  - Dynamic analysis and testing
  - Approaches to functional or black-box testing
  - Hazard Analysis
  - Choice of programming language
  - Modelling
  - Performance testing
  - Semi-formal methods
  - Static analysis
  - Modular approaches

Modeling

D.6 : Modelling Referenced by Clauses 7.6					
TECHNIQUE/MEASURE	**Ref**	**SIL1**	**SIL2**	**SIL3**	**SIL4**
1. Data Flow Diagrams	B.12	R	R	R	R
2. Finite State Machines	B.29	--	HR	HR	HR
3. Formal Methods	B.30	--	R	R	HR
4. Performance Modelling	B.45	R	R	R	HR
5. Time Petri Nets	B.64	--	HR	HR	HR
6. Prototyping/Animation	B.49	R	R	R	R
7. Structure Diagrams	B.59	R	R	R	HR
NOTE: One or more of the above techniques should be used.					

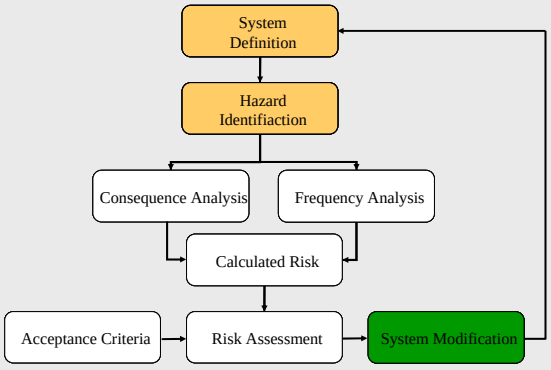
SILs

- ✓ What does it all mean?
  - SIL 4 system should have a duration of about 10^{-9} hours between critical failures
  - If established SIL 4 needed, used all the techniques…
  - But there is no measurement that the results actually achieves the target
  - Standard assumes that you are competent in all methods and apply everything possible
  - Except that these may be insufficient or not affordable

The Engineering Council's Code of Practice on Risk Issues

1	Professional responsibility	Exercise reasonable professional skill and care
2	Law	Know about and comply with the law
3	Conduct	Act in accordance with the codes of conduct
4	Approach	Take a systematic approach to risk issues
5	Judgement	Use professional judgement and experience
6	Communication	Communicate within your organization
7	Management	Contribute effectively to corporate risk management
8	Evaluation	Assess the risk implications of alternatives
9	Professional development	Keep up to date by seeking education and training
10	Public awareness	Encourage public understanding of risk issues

Hazard and Risk Analysis Process



Risk Reduction Procedures

- Four main categories of risk reduction strategies, given in the order that they should be applied:
  - Hazard Elimination
  - Hazard Reduction
  - Hazard Control
  - Damage Limitation
- Only an approximate categorisation, since many strategies belong in more than one category

Hazard Ellimination

- Before considering safety devices, attempt to eliminate hazards altogether
  - use of different materials, e.g., non-toxic
  - use of different process, e.g., endothermic reaction
  - use of simple design
  - reduction of inventory, e.g., stockpiles in Bhopal
  - segregation, e.g., no level crossings
  - eliminate human errors, e.g., for assembly of system use colour coded connections

Design Principles

- Familiar
  - use tried and trusted technologies, materials techniques
- Simple
  - testable (including controllable and observable)
  - portable (no use of sole manufacturer components compiler dependent features)
  - understandable (behaviour can easily be from implementation)
  - deterministic (use of resources is not random)
  - predictable (use of resources can be predicted)
  - minimal (extra features not provided)

Design Principles (cont.)

- Structured design techniques
  - defined notation for describing behaviour
  - identification of system boundary and environment
  - problem decomposition
  - ease of review
- Design standards
  - limit complexity
  - increase modularity
- Implementation standards
  - presentation and naming conventions
  - semantic and syntactic restrictions in software

Classes of System Failure

- Random (physical) failures
  - due to physical faults
  - e.g., wear-out, aging, corrosion
  - can be assigned quantitative failure probabilities
- Systematic (design) failures
  - due to faults in design and/or requirements
  - inevitably due to human error
  - usually measured by integrity levels
- Operator failures
  - due to human error
  - mix of random and systematic failures

Nature of Random Failures

- Arise from random events generated during operation or manufacture
- Governed by the laws of physics and cannot be eliminated
- Modes of failure are limited and can be anticipated
- Failures occur independently in different components
- Failure rates are often predictable by statistical methods
- Sometimes exhibit graceful degradation
- Treatment is well understood

Treating Random Failures

- Random failures cannot be eliminated and must be reduced or controlled
- Random failures can be mitigated by:
  - predicting failure modes and rates of components
  - applying redundancy to achieve overall reliability
  - performing preventative maintenance to replace components before faults arise
  - executing on-line or off-line diagnostic checks

Nature of Systematic Failures

- Ultimately caused by human error during development, installation or maintenance
- Appear transient and random since they are triggered under unusual, random circumstances
- Systematic and will occur again if the required circumstances arise
- Failures of different components are *not* independent
- Difficult to predict mode of failure since the possible deviations in behaviour are large
- Difficult to predict the likelihood of occurrence

Treating Systematic Failures

- In theory, design failures can be eliminated
- In practice, perfect design may be too costly
- Focus the effort on critical areas
  - identify safety requirements using hazard analysis
  - assess risk in system and operational context
- Eliminate or reduce errors using quality development processes
  - verify compliance with safety requirements
  - integrate and test against safety requirements

Design Faults

- Design faults are much more difficult to deal with than random (degradation) faults because:
  - They are hard to anticipate
  - Their effects are hard to predict
  - Component failure semantics tend to be undefined
- This makes all forms difficult to tolerate, especially software faults

Common Design Faults

- All forms of software:
  - System software
  - Application software
  - Embedded software (firmware)
- All forms of computing hardware:
  - Hardware design faults now dominate
  - Degradation faults used to dominate
- Power supply systems
- Component interconnection wiring

Design Diversity

- Idea:
  - Design faults are “aspects” of design
  - Different designs, different faults
  - Produce multiple designs—independent level.
  - Operate in parallel at execution time
- Applies to all types of design fault
- Can be configured using many system architectures, like NMR, TMR, etc.

Hazard Reduction

- Reduce the likelihood of hazards
- Use of barriers, physical or logical
  - Lock-ins
  - Lock-outs
  - Interlocks
- Failure minimization
  - Redundancy
  - Recovery

Forms of Redundancy

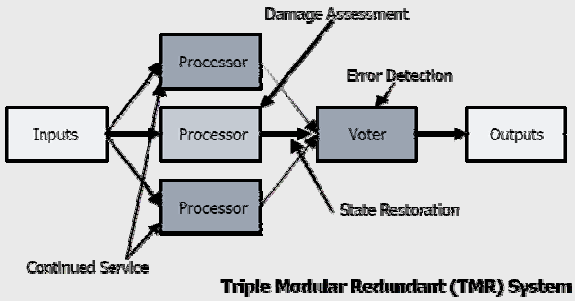
- Hardware redundancy
- Software redundancy
- Information redundancy
- Temporal (time) redundancy

- Design diversity, for hardware/software
  - Develop different implementations of the same hardware/software component
  - Called N-version programming
  - Then apply static or dynamic redundancy

Hardware Redundancy

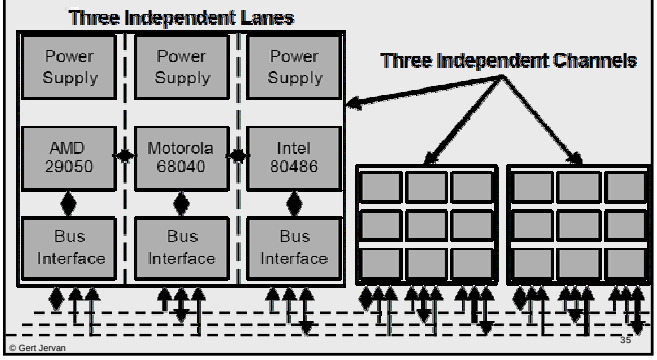
- Static redundancy
  - Component (at least) triplicated
    - Triple Modular Redundancy (TMR), N-Modular Redundancy (NMR)
  - Voting element used to remove effects of single failure
  - Loss Of Unit Implies:
    - Removal Or Containment
    - Service Provided By Those That Remain
- Dynamic redundancy
  - Component has a mirror that is invoked when fault occurs
  - Cold or Hot Standby, spares
  - Loss Of Unit Implies:
    - Removal Or Containment
    - Introducing Standby Unit

Fault Tolerant System Example



Risk of single-point failure

B777 Primary Flight Computer Architecture



N Modular Redundancy

- Independent development of modules
- This is what Boeing did with N = 3 for processors
- Operation:
  - Parallel—forward error recovery
  - Serial—backward error recovery
- In software with forward error recovery, referred to as N-version programming
- In software with backward error recovery, referred to as recovery block

B777 PFC CPUs

- Problem:
  - Processors often (essentially always) contain design faults, need to deal with them
  - 777 channel is a TMR system
- Three manufacturers, three designs
- Are these designs different?
- How would you measure the difference?
- What metric is there for design diversity?

Redundancy

- Software redundancy, e.g. N-version programming
- Information redundancy, e.g., checksums, cyclic redundancy codes, error correcting codes
- Hybrid redundancy

N-Version Programming

- NMR for software
- Practical issues:
  - Cost of development, team separation
  - Resources during execution
  - Different execution times for different versions
  - Different but similar output values
  - Different but valid output values (multiple correct solutions)

N-Version Programming

- Performance:
  - Assumed statistical independence
  - If not independent, then no lower bound
  - Common specification defects
  - Common implementation (design) faults
- Problem compounded by comparison checking during testing

Hybrid Redundancy

- N-S modular redundancy with ‘‘S’’ spares
- As members of the N-S fail, spares switched in
- Able to tolerate up to N-2 failures
- Spares may be unpowered:
  - Saves power
  - Unpowered units much more reliable than powered
  - Attention required to infant mortality
- Clearly applicable to:
  - Long-duration systems
  - Systems with no repair opportunity

Space Shuttle Computer System

- Uses combination of
  - Redundancy, fault detection and design diversity
- Hardware voting on sensors and actuators
- Five identical computers
  - During critical stages, four computers work in NMR with voting for fault detection
  - Fifth computer performs non-critical functions, e.g. communications
- Fault tolerance
  - Tolerates failure of two computers
  - In case of third failure, crew/ground control decide which computer wins
  - Fifth computer can take over control, uses different software

Recovery

- Can reduce failures by recovering after error detected but before component or system failure occurs
- Recovery can only take place after detection of error
  - Backward recovery
  - Forward recovery

Error Detection

- Based on check that is independent of implementation of the system
  - coding - parity checks and checksums
  - reasonableness - range and invariants
  - reversal - calculate square of square root
  - diagnostic - hardware built-in tests
  - timing - timeouts or watchdogs

Error Detection (cont.)

- Timing of error detection important
  - early error detection can be used to prevent propagation
  - late error detection requires a check of the entire activity of system
- Checking may be in several forms
  - monitor, acting after a system function, checking outputs after production but before use
  - kernel, encapsulating (safety-critical) functions in a subsystem that allows all inputs to and outputs from the kernel to be checked

Backward Recovery

- Corrects errors through reversing previous operations
- Return system to a previous known safe state
- Allows retry
- Requires checkpoints or saved states (and the expenses involved with producing them)
- Rollback usually impossible with real-time system

Forward Recovery

- Corrects errors without reversing previous operations, finding safe (but possibly degraded) state for system
  - data repair, use redundancy in data to perform repairs
  - reconfiguration, use redundancy such as backup or alternate systems
  - coasting, continue operations ignoring (hopefully transient) errors
  - exception processing, only continue with selection of (safetycritical) functions
  - failsafe, achieve safe state and cease processing
    - use passive devices (e.g., deadman switch) instead of active devices (e.g., motor holding weight up)

Hazard Control

- Detect and control hazard before damage occurs
- Reduce the level or duration of the hazard
- Hazard control mechanisms include:
  - Limiting exposure: reduce the amount of time that a system is in an unsafe state (e.g. don't leave rocket in armed state)
  - Isolation and containment
  - Fail safe design

Damage Limitation

- In addition to eliminating hazards or employing safety devices, consider
  - warning devices
  - procedures
  - training
  - emergency planning
  - maintenance scheduling
  - protective measures

Architectural Design

- Suitable architectures may allow a high integrity system to be built from lower integrity components
  - combinations of components must implement a safety function independently
  - overall likelihood of failure should be the same or less
  - be wary of common failure causes
- Apportionment approaches can be quantitative and/or qualitative
  - quantitative: numerical calculations
  - qualitative: judgement or rules of thumb

TTÜ
TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Arvutitehnika instituut
ati.ttu.ee

Fault Tolerance

Basics

- Computing systems are characterized by five fundamental properties:
  - functionality
  - usability
  - performance
  - cost
  - dependability

Faults

- Faults are there!
- Either prevent, **tolerate**, remove or forecast
- We need redundancy
  - System that is more complex than needed for performing the required task

Means to Achieve Dependability

- Fault prevention
  - Good design processes, avoid design flaws
  - Good procedures for runtime faults
- Fault tolerance
  - Fault detection
  - Redundancy
  - Diversity
- Fault removal
  - Verification and validation during design
  - Corrective/preventive action during maintenance
- Fault forecasting
  - Simulation, modelling, prediction
  - Analysis based on history statistics

Fault Tolerance

- Automobile:
  - Spare Tires
  - Dual Braking Systems
- Power Supplies:
  - UPS/battery backup
  - Power-fail interrupts
- Multiple engines on aircraft
- Emergency lighting in buildings
- Tape backups of disk files
- Checkpoint/restart of long-running programs
- Parity and SECDED in computer memories

Faults

- Random faults (Degradation faults)
  - Arise during operation
  - Usually hardware component failure
- Systematic faults (Design Faults)
  - mistakes in the spec
  - mistakes in the hardware
  - mistakes in the software

Faults

- Faults are either permanent, transient or intermittent
- Design faults are always permanent
- Dealing with faults:
  - During development: fault avoidance & removal
  - During operation: fault tolerance & detection

Hardware Faults

- Use of fault models
- Decomposition into modules
  - Gates, transistors, etc
- Connection faults
  - Single stuck-at model, bridging model (shorts), stuck-open
- Used to model hardware faults
  - Design testing schemes for digital circuits
  - Fault removal coverage usually less than 100%
  - Guard against physical defects, not design faults
- In safety critical systems
  - Combined with Failure Modes and Effects Analysis (FMEA)
  - Need fault avoidance by verification…

Other Faults

- Hardware design and specification faults
  - Few fault models available
  - Many faults cannot be modelled
  - System must meet the spec, but spec might be incorrect as well
  - Spec errors may manifest as either hardware or software failures
  - Use of formal methods (formal spec. languages, automata theory, formal verification, model checking, etc.)

Software Faults

- Bugs:
  - Software spec faults
  - Coding faults
  - Logical errors within calculations
  - Stack overflows or underflows
  - Uninitialized variables
- No random failures and it does not degrade with age
- Always systematic
- Exhaustive testing almost impossible
- Must be tolerated

SW Testing - i.e. Verification

- Verification:
  - SW testing
  - formal verification
- Functional and structural testing
- Path testing, transaction flow testing, data-flow testing, domain testing, mutation testing etc.

Fault Detection Techniques

- Functionality checking
  - march test
- Consistency checking
  - range checking, overflow
- Signal comparison
- Information redundancy
  - checksums, cyclic redundancy codes, error correcting codes
- Monitoring techniques
  - Loopback testing
  - Power supply monitoring

Watchdog Timer

- An inexpensive method of error detection
- Process being watched must reset the timer before the timer expires, otherwise the watched process is assumed as faulty
- Watchdog timers only detect errors which manifest themselves as a control-flow error such that the system does not continue to reset the timer
- Only processes with relatively deterministic runtimes can be checked, since the error detection is based entirely on the time between timer resets

Heartbeats

- A common approach to detecting process and node failures in a distributed (networked) computing environment.
- Periodically, a monitoring entity sends a message (a heartbeat) to a monitored node or process and waits for a reply.
- If the monitored node does not respond within a predefined timeout interval, the node is declared as failed and appropriate recovery action is initiated.
- Adaptive or smart

System Testing

HW Testing

SW Testing

HW/SW Testing
(system testing)

TTÜ TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Arvutitehnika instituut
ati.ttu.ee

Software Testing

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Arvutitehnika instituut
ati.ttu.ee

Programmers are in a race with the Universe to create bigger and better idiot-proof programs.

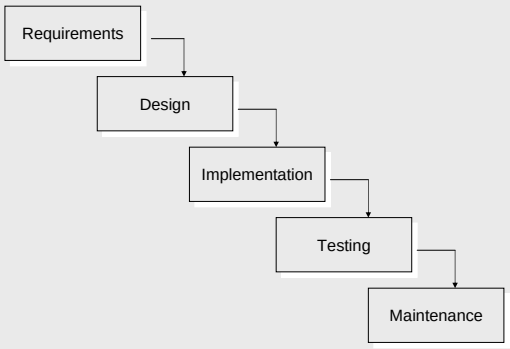
While the Universe is trying to create bigger and better idiots.

So far the Universe is winning

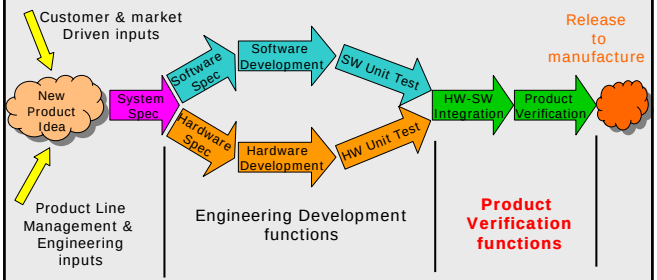
Software Testing Topics

- ✓ Test Economics
- ✓ Types of Testing
- ✓ Testing coverage

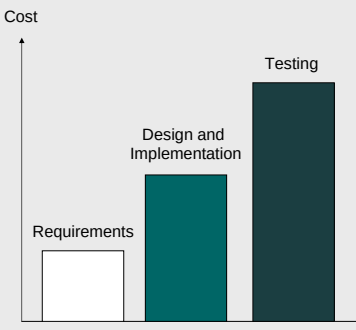
Software Life Cycle



The Product Development Cycle

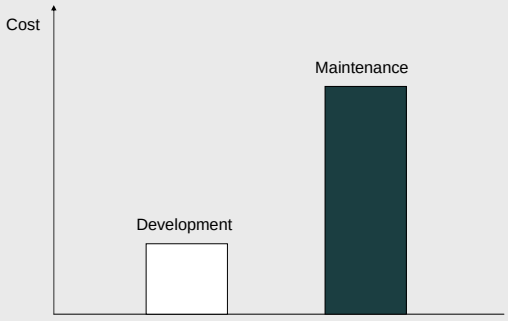


Software Development Costs



- ✓ For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined.
- ✓ Stop testing is a business decision
  - There is always something more to test
  - Risk based decision

Software Life Cycle Costs



Software Qualities

- Correctness
- Reliability (dependability)
- Robustness
- Safety
- Security (survivability)
- Performance
- Productivity
- Maintainability, portability, interoperability, ...

Software Verification and Validation

- Verification
  - Are we building the product right?
  - Process-oriented
    - Does the product of a given phase fulfill the requirements established during the previous phase?
- Validation
  - Are we building the right product?
  - Product-oriented
    - Does the product of a given phase fulfill the user's requirements?

Techniques for V&V

- Static
  - Collects information about a software without executing it
    - Reviews, walkthroughs, and inspections
    - Static analysis
    - Formal verification
- Dynamic
  - Collects information about a software with executing it
    - Testing: finding errors
    - Debugging: removing errors

Static Analysis

- Control flow analysis and data flow analysis
  - Extensively used for compiler optimization and software engineering
- Examples
  - Unreachable statements
  - Variables used before initialization
  - Variables declared but never used
  - Variables assigned twice but never used between assignments
  - Variables used twice with no intervening assignment
  - Possible array bound violations

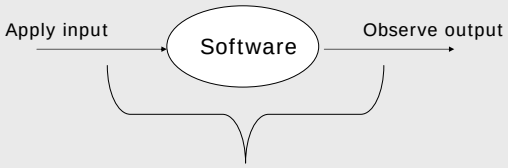
Formal Verification

- Given a model of a program and a property, determine whether the model satisfies the property based on mathematics
- Examples
  - Safety
    - If the light for east-west is green, then the light for south-north should be red
  - Liveness
    - If a request occurs, there should be a response eventually in the future

Introduction to Testing

- Debugging and testing are not the same thing!
- Testing is a systematic attempt to break a program.
  - Correct, bug-free programs by construction are the goal but until that is possible (if ever!) we have testing.
  - Since testing is basically **destructive** in nature, it requires that the tester discard *preconceived* notions of the *correctness* of the software to be tested

Testing



Validate the observed output

Is the observed output the same as the expected output?

Software Testing Fundamentals

- Testing objectives include
  - Testing is a process of executing a program with the intent of finding an error.
  - A **good** test case is one that has a high probability of finding an as yet undiscovered error.
  - A **successful** test is one that uncovers an as yet undiscovered error.

Limitations of Testing (I)

- To test all possible inputs is impractical or impossible

```
int foo(int x) {
        y = very-complex-computation(x);
        write(y);
}
```

- To test all possible paths is impractical or impossible

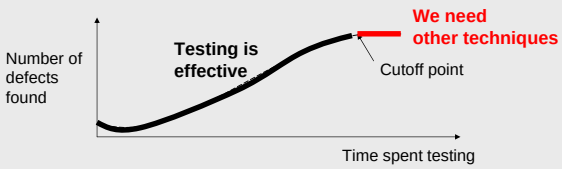
```
int foo(int x) {
        for (index = 1; index < 10000; index++)
                write(x);
}
```


Limitations of Testing (II)

- Dijkstra, 1972
  - Testing can be used to show the presence of bugs, but never their absence
- Goodenough and Gerhart, 1975
  - Testing is successful if the program fails
- The (modest) goal of testing
  - Testing cannot guarantee the correctness of software but can be effectively used to find errors (of certain types)

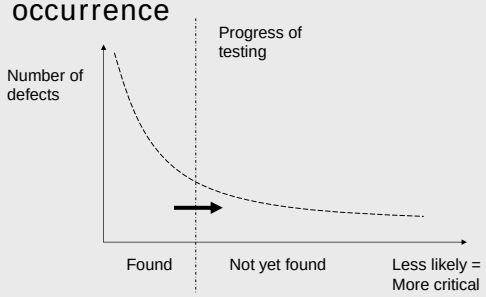
Economics of Testing (I)

- The characteristic S-curve for error removal



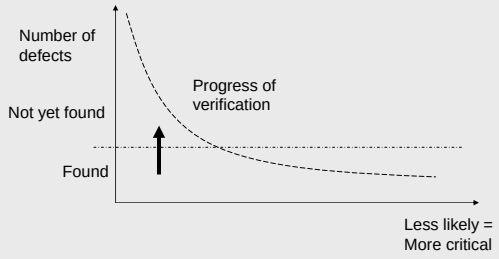
Economics of Testing (II)

- Testing tends to intercept errors in order of their probability of occurrence



Economics of Testing (III)

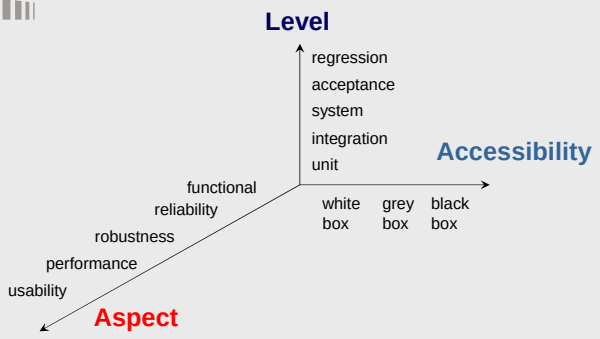
- Verification is insensitive to the probability of occurrence of errors



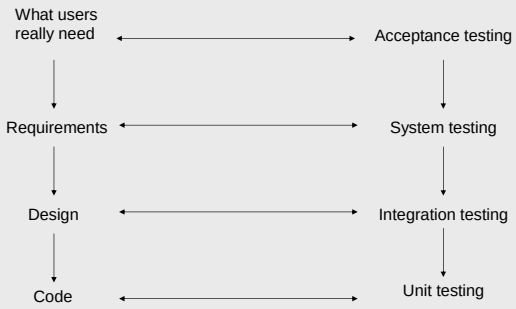
Fundamental Questions in Testing

- When can we stop testing?
  - Test coverage
- What should we test?
  - Test generation
- Is the observed output correct?
  - Test oracle
- How well did we do?
  - Test efficiency
- Who should test your program?
  - Independent V&V

Types of Testing



Levels of Testing



Component/Unit Testing (I)

- A unit of testing
  - Functions in procedural programming languages such as C, Fortran, …

Test driver
```
F1(int x1, y1) {
        ……
        F2(x1+1, y1-1);
}
```

Test unit
```
F2(int x2, y2) {
        ……
        F3(x2+2, y2-1);
}
```

Test stub
```
F3(int x3, y3) {
        ……
}
```


Component/Unit Testing (II)

- Require knowledge of code
  - High level of detail
  - Deliver thoroughly tested components to integration
- Stopping criteria
  - Code Coverage
  - Quality

Component/Unit Testing (III)

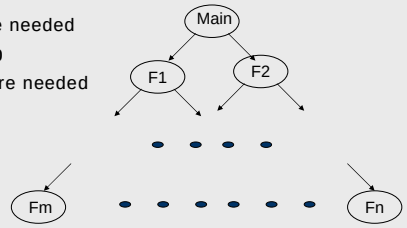
- Test case
  - Input, expected outcome, purpose
  - Selected according to a strategy, e.g., branch coverage
- Outcome
  - Pass/fail result
  - Log, i.e., chronological list of events from execution

Integration Testing (I)

- Interactions among units (assembled components that must be tested and accepted previously)
  - Import/export type compatibility
  - Import/export range errors
    - F1 calls F2 with a parameter of array
    - F1 assumes array of size 8, while F2 assumes an array of size 10
  - Import/export representation
    - F1 calls F2 with a parameter Elapsed_time
    - F1 thinks in seconds, while F2 thinks in miliseconds

Integration Testing (II)

- Strategies for integration testing
  - Top-down
    - Stubs are needed
  - Bottom-up
    - Drivers are needed
  - Big-bang
  - Functional
  - Drivers & stubs have to tested as well!



System Testing (I)

- Tests the overall system (the integrated hardware and software) to determine whether the system meets its requirements
- Focuses on the use and interaction of system functionalities rather than details of implementations
- Test cases derived from specification
- Should be carried out by a group independent of the code developers
- Should be planned with the same rigor as other phases of the software development
- Use-case focus

System Testing (II)

- Non-functional testing
- Quality attributes
  - Performance, can the system handle required throughput?
  - Reliability, obtain confidence that system is reliable
  - Timeliness, testing whether the individual tasks meet their specified deadlines
  - etc.

Acceptance Testing

- User (or customer) involved
- Environment as close to field use as possible
- Focus on:
  - Building confidence
  - Compliance with defined acceptance criteria in the contract

Re-Test and Regression Testing (I)

- Conducted after a change
- Re-test aims to verify whether a fault is removed
  - Re-run the test that revealed the fault
- Regression test aims to verify whether new faults are introduced
  - Re-run all tests
  - Should preferably be automated

Re-test & Regression Testing (II)

- Development versus maintenance
  - Development costs: 1/3
  - Maintenance costs: 2/3
- Testing in maintenance phase
  - How can we test modified or newly inserted programs?
    - Ignore old test suites and make new ones from the scratch or
    - Reuse old test suites and reduce the number of new test suites as many as possible

Accessibility of Testing

- White box testing (structural testing, program-based testing)
- White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived that
  - guarantee that all independent paths within a module have been exercised at least once,
  - exercise all logical decisions on their true and false sides,
  - execute all loops at their boundaries and within their operational bounds, and
  - exercise internal data structures to ensure their validity.

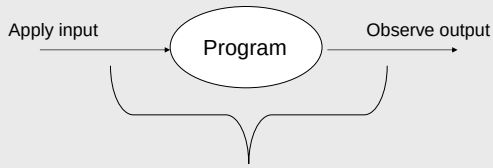
Accessibility of Testing (II)

- Black box testing (functional testing, specification-based testing)
  - Assumes that the program is unavailable or testers do not want to look at the details of the program
    - Derives test cases from the requirements of the program
    - Controls and observes the program only through external interfaces
    - Ideally done by independent test group (not original programmer)
- Grey box testing

Program-Based Testing (I)

- Main steps
  - Examine the internal structure of a program
  - Design a set of inputs satisfying a coverage criterion
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- Limitations
  - Cannot catch omission errors
    - What requirements are missing in the program?
  - Cannot provide test oracles
    - What is the expected output for an input?

Program-Based Testing (II)



Validate the observed output against the expected output

Who will take care of test oracles?

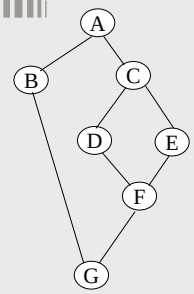
Statement Coverage

- Statement coverage of a set of test cases is defined to be the proportion of statements in a unit covered by those test cases.
- 100% statement coverage for a set of tests means that all statements are covered by the tests. That is, all statements will be executed at least once by running the tests.

Branch Coverage

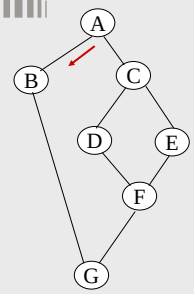
- Branch coverage is determined by the proportion of decision branches that are exercised by a set of proposed test cases.
- 100% branch coverage is where every decision branch in a unit is visited by at least one test in the set of proposed test cases.

Example – Branch coverage



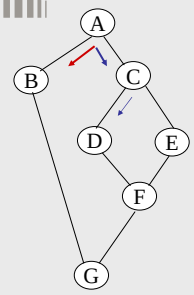
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



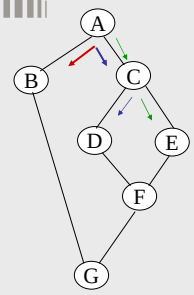
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



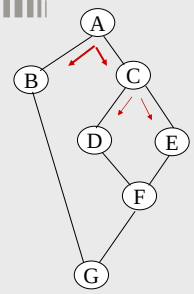
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



What branch coverage is achieved by ABG, ACDFG, ACEFG?

4 in total.

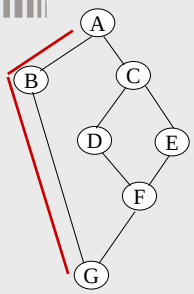
4 covered

So 4/4 = 100% branch coverage

Path Coverage

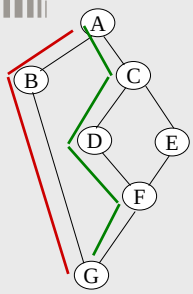
- Path coverage is determined by assessing the proportion of execution paths through a unit exercised by the set of proposed test cases.
- 100% path coverage is where every path in the unit is executed at least once by the set of proposed test cases.
- 100% path coverage is achieved by an ideal test set. As we saw the other week, it is all but impossible or infeasible in most programs of any size.

Example – Path coverage



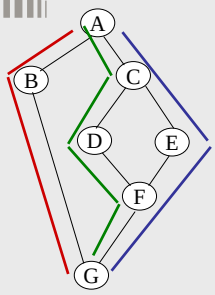
What path coverage is achieved by ABG, ACDFG, ACEFG?

Example – Path coverage



What path coverage is achieved by ABG, ACDFG, ACEFG?

Example – Path coverage



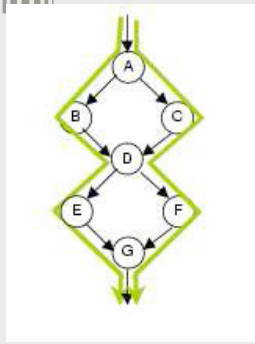
What path coverage is achieved by ABG, ACDFG, ACEFG?

3/3=100%

Coverage

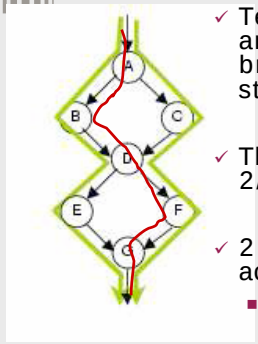
- It is possible to have 100% statement coverage without 100% branch coverage
- It is possible to have 100% branch coverage without 100% path coverage
- 100% path coverage implies 100% branch coverage and 100% branch coverage implies 100% statement coverage

An example



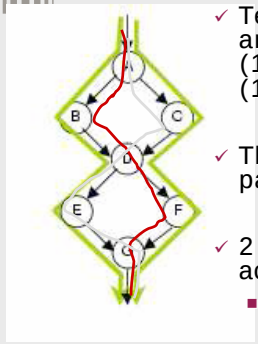
- ✓ Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- ✓ They, however, only cover 2/4 paths (50%).

An example



- ✓ Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- ✓ They, however, only cover 2/4 paths (50%).
- ✓ 2 more tests are required to achieve 100% path coverage
  - ABDFG

An example



- ✓ Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- ✓ They, however, only cover 2/4 paths (50%).
- ✓ 2 more tests are required to achieve 100% path coverage
  - ABDFG, ACDEG

Loop Testing

- ✓ It is usually impossible or infeasible to test all paths in a program involving loops
- ✓ Basis Path Testing
  - Zero path: Test zero iterations of the loop body (Guard is negated by loop initialisation)
  - One path: Test a single iteration of the loop body (Good idea to try for 100% path coverage of loop body if loop body is not iterative)
  - Does not consider maximum iteration termination in many cases
  - Does not consider combinations of loop body paths in successive iterations

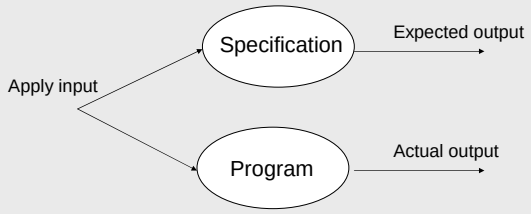
Mutation testing

- ✓ Create a number of mutants, i.e., faulty versions of program
  - Each mutant contains one fault
  - Fault created by using mutant operators
- ✓ Run test on the mutants (random or selected)
  - When a test case reveals a fault, save test case and remove mutant from the set, i.e., it is killed
  - Continue until all mutants are killed
- ✓ Results in a set of test cases with high quality
- ✓ Need for automation

Specification-Based Testing (I)

- ✓ Main steps
  - Examine the structure of the program's specification
  - Design a set of inputs from the specification satisfying a coverage criterion
  - Apply the inputs to the specification and collect the expected outputs
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- ✓ Limitations
  - Specifications are not usually available
    - Many companies still have only code, there is no other document.

Specification-Based Testing (II)



Validate the observed output against the expected output

Object-Oriented Program Testing

- Unit testing for OO Programs
  - A class is a set of variables and member functions
  - 50% of member functions are just 10 lines of code
  - A class is often a unit of testing in C++ or Java
- Integration testing for OO Programs
  - Rule of thumb in OO development
    - Make a large number of small classes in a bottom-up fashion
  - There are several relationships between classes
    - Association, aggregation, inheritance, concurrency

Steps to Testing Nirvana

- Think about potential problems as you design and implement. Make a note of them and develop tests that will exercise these problem areas.
  - Document all loops and their boundary conditions, all arrays and their boundary conditions, all variables and their range of permissible values.
  - Pay special attention to parameters from the command line and into functions and what are their valid and invalid values.
  - Enumerate the possible combinations and situations for a piece of code and design tests for all of them.
  - GIGO - what happens when garbage goes in?

Kernighan, Pike, "The Practice of Programming"

Steps to Testing Nirvana

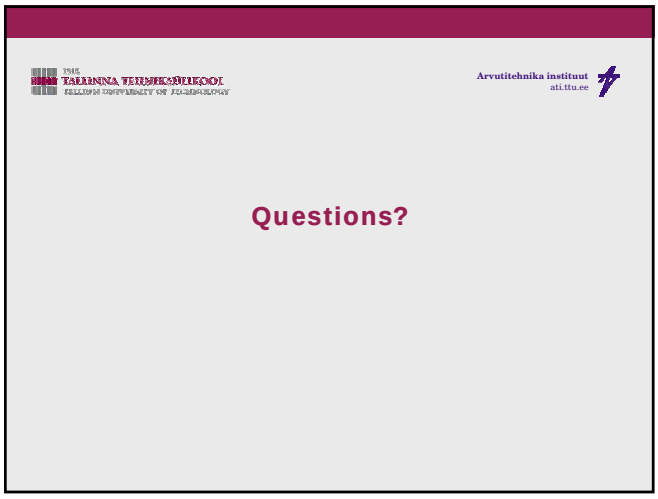
- Test systematically, starting with easy tests and working up to more elaborate ones.
  - Often leads to "bottom up" testing, starting with simplest modules at the lowest level of calling
  - When those are working, test their callers
  - Document (and/or automate) this testing so that it can be repeated (regression testing) constantly as the code grows and changes.

Steps to Testing Nirvana

- Within a module, test *incrementally* as you code
  - Write, test, add more code, test again, repeat
  - The earlier that errors are detected, the easier they are to locate and fix.
  - Testing is not only concerning code
    - Documents and models should also be subject to testing

Tricks of the Trade

- Test boundary conditions.
  - loops and conditional statements should be checked to ensure that loops are executed the correct number of times and that branching is correct
  - if code is going to fail, it usually fails at a boundary
  - check for off-by-one errors, empty input, empty output


1918
TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
Arvutitehnika instituut
ati.ttu.ee
Questions?