IAF0530 (MSc)
IAF9530 (PhD)

Süsteemide usaldusväärsus ja veakindlus
Dependability and fault tolerance

Lecture 4

Gert Jervan
Department of Computer Engineering (ATI)
Tallinn University of Technology (TTÜ)

Course Schedule

- Major change!
  - No meetings March 21, March 28, April 4!
  - March 28 presentations are cancelled, however you are required to send the draft and slides by deadline!
  - Special reading assignment to cover the missing lecture. Will be published online latest March 18.
  - We'll meet again on April 11.

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.

13

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

14

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

15

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

16

System Testing

HW Testing SW Testing

HW/SW Testing
(system testing)

17

Software Testing

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.

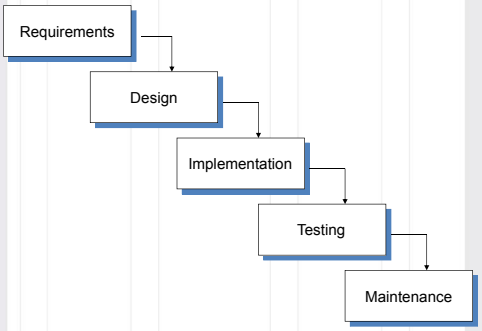
So far the Universe is winning

Software Testing Topics

- Test Economics
- Types of Testing
- Testing coverage

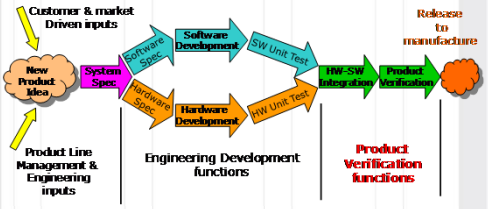
20

Software Life Cycle



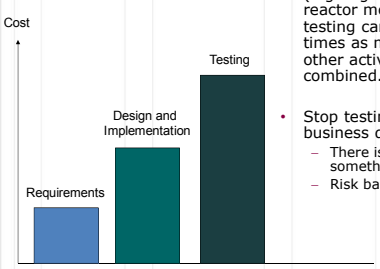
21

The Product Development Cycle



22

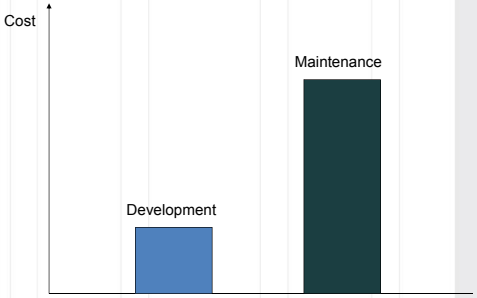
Software Development Costs



- 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

23

Software Life Cycle Costs



24

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

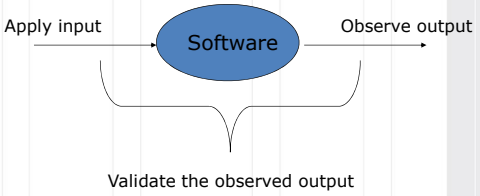
Formal Verification

- 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



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

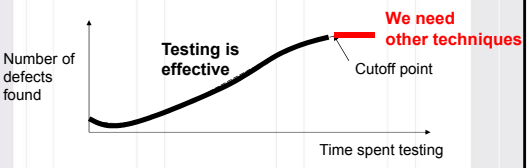
```
int foo(int x) {
        for (index = 1; index < 10000; index++)
                write(x);
}
```


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)

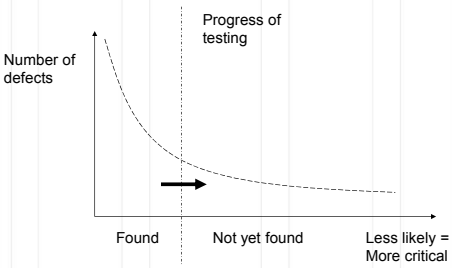
Economics of Testing (I)

- The characteristic S-curve for error removal



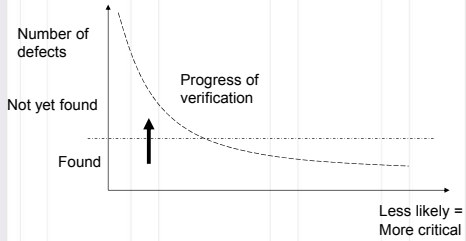
Economics of Testing (II)

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



Economics of Testing (III)

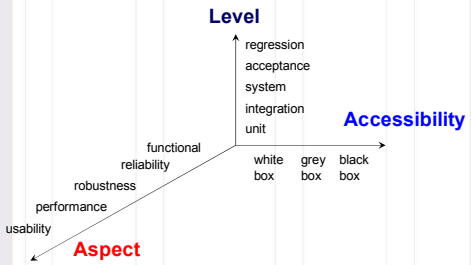
- Verification is insensitive to the probability of occurrence of errors



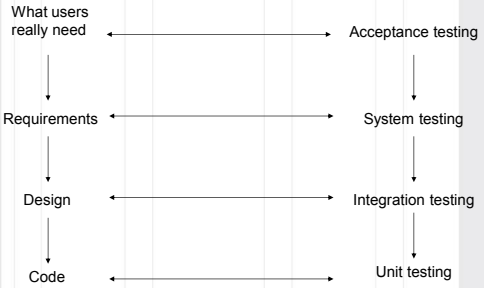
Fundamental Questions in Testing

- 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



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.

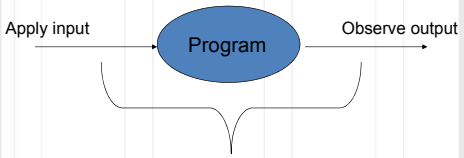
Accessibility of Testing (II)

- 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?

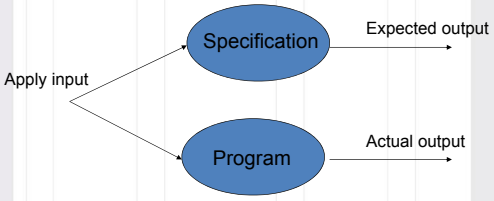
Covergae metrics

- Statement coverage
- Branch coverage
- Path coverage
- Mutation coverage

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

The Budget Coverage Criterion

- A common answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1: Test selection is more important than stopping criteria per se.
  - Implication 2: Practical comparison of approaches must consider the cost of test case selection

Remarks by Bill Gates
17th Annual ACM Conference on Object-Oriented Programming, Seattle, Washington, November 8, 2002

"… When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development. We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time…

… We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing."

Remarks by Bill Gates (cont.)

"…The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one."

"… Well, one of the interesting questions is, when you change a program, … what portion of these test cases do you need to run?"

Testing Real-Time Systems

Distributed
Self-Checking

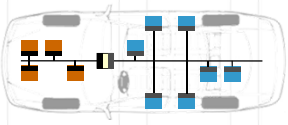
System Testing

HW Testing → HW/SW Testing (system testing) ← SW Testing

52

Real-Time Systems

- *Real-Time System* – system, which is required to adhere not only functional but also tempoal requirements ("timing constraints" or "deadlines")
- RT-systems:
  - Hard RT-systems
  - Soft RT-systems



53

Real-Time Systems Testing

- Inherits issues from concurrent systems
  - Problems becomes harder due to time-constraints
    - More sensitive to probe-effects
    - Timing/order of inputs become more significant
- Adds new potential problems
  - New failure types
    - E.g. Missed deadlines, Too early responses…
  - Test inputs → Execution times
  - Faults in real-time scheduling
    - Algorithm implementation errors
    - Assumption about system wrong

54

Real-Time Systems Testing

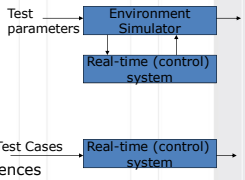
- Pure time-triggered systems
  - Deterministic
  - Test-methods for sequential software usually apply
- Fixed priority scheduling
  - Non-deterministic
    - Limited set of possible execution orders
  - Worst-case w.r.t timeliness can be found from analysis
- Dynamic (online) scheduled systems
  - Non-deterministic
    - Large set of possible execution orders
  - Timeliness needs to be tested

Testing Timeliness

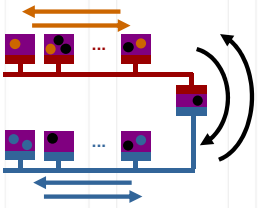
- Aim : Verification of specified deadlines for individual tasks
  - Test if assumptions about system hold
    - E.g. worst-case execution time estimates, overheads, context switch times, hardware acceleration efficency, I/O latency, blocking times, dependency-assumptions
  - Test system temporal behavior under stress
    - E.g. Unexpected job requests, overload management, component failure, admission control scheme
- Identification of potential worst-case execution orders
- Controllability needed to test worst-case situations efficiently

Testing Embedded Systems

- System-level testing differs
  - Performed on target platform to keep timing
- Closed-loop testing
  - Test-cases consist of parameters sent to the environment simulator
- Open-loop testing
  - Test-cases contain sequences of events that the system should be able to handle



Distributed Real-Time Systems



- Distributed applications
  - On a single cluster
  - On several clusters
- Motivation
  - Reduce costs: use resources efficiently
  - Requirements: close to sensors/ actuators
- Distributed applications are difficult to...
  - Analyze (e.g., guaranteeing timing constraints)
  - Design (e.g., efficient implementation)

Testing Distributed RT-Systems

- Problems with distributed systems:
  - Increased complexity
  - The difficulties of observing and monitoring
  - Non-reproducible behaviour of the system
  - The lack of synchronized global clock and, consequently, the difficulties of unambiguously defining a "global state"

Testing Distributed RT-Systems

- Observability
  - What?
  - How?
  - When?
- Controllability
- Auxiliary outputs, interactive debuggers

Observability Issues

- Probe effect (Gait,1985)
  - "Heisenbergs's principle" - for computer systems
  - Common "solutions"
    - Compensate
    - Leave probes in system
    - Ignore
- Must observe execution orders
  - Gain coverage

Controllability Issues

- To be able to test correctness of a particular execution order we need control
  - Input data to all tasks
    - Initial state of shared data/buffers
  - Scheduling decisions
    - Order synchronization/communication between tasks

Testing Distributed RT-Systems

- **Reproducibility**
  - *Regression testing* – retesting after errors have been corrected
    - errors truely corrected
    - no new errors
  - A distributed system may be non-reproducible due to nondeteminism in it's hardware, software or operating system

Testing Distributed RT-Systems

- **Obtaining reproducibility**
  - Language-based approach
    - Enforcing the identified scenarios during execution
    - All solutions rely on source code transformations
  - Implementation based approach
    - Collecting all missing information during an execution of the system
    - Event histories or traces

Testing Distributed RT-Systems

- Disadvantages of implementation based approach:
  - Special dedicated HW (to monitor)
  - Large amount of information
  - Can we guarantee the correctnes of reply?
  - Modified programs. What happens with event histories. Are they still valid?
  - Event histories can be used only on target systems

Testing Distributed RT-Systems

- Interdependence of Obsevability and Reproducibility
  - Not independent!
  - Probe effect

Testing Distributed RT-Systems

- **The host/target approach**
  - Host - development
  - Target - execution
- Testing on the host system is used for (functional) unit testing and preliminary integration testing (as much as possible)
- Testing on the target system involves completing the integration test and performing the system test. Also performance, timing, etc.

Testing Distributed RT-Systems

- Environment simulation (for target system test)
  - Simulated v. real environment:
    - Safety and/or cost considerations.
    - "rare event" situations
    - More control over simulated environment
    - Easier to obtain responses and test results
  - On-line v. off-line test data generation:
    - Need to generate large amounts of input data
    - Runs cost-effectively

Testing Distributed RT-Systems

- Representativity
  - Only small number of real-world scenarios can be anticipated and taken into account.
  - Only a fraction of those anticipated real-world scenarios can be tested due to the combinatorial explosion of possible event and input combinations.
- Test coverage - how many of the anticipated real-time scenarios can be or have been covered by corresponding test scenarios.

Self-checking distributed systems

- Run-time checking of the effects of faults on system behaviors needs to be carried out continuously.
- Reliability – the key to distributed SW quality

Self-checking distributed systems

- Fault-secure systems are systems, where faults may be enforced not to propagate.
  - Faults are not visible or have no effect
  - Faults are visible, but it's easy to notice that an error exists
- Self-testing – System is self testing when there exists testing behavior, occurring during the run-time behavior of the system, such that this fault will be propagated to the output and it's easy to notice, that there is a fault (out of predefined set of values)
- System is self-checking for a set of faults, if whatever a fault belonging to this set, it is fault-secure and self-testing.

Self-checking distributed systems

- Worker-observer
  - the worker is a classical implementation of the system behavior
  - the observer is a given redundant implementation whose outputs are comparable with the outputs of the worker.
- To obtain observing behavior:
  - Redundancy
  - Reference
  - Visibility
    - Worker cooperates with the observer
    - Worker behavior can be spied by the observer

Self-checking distributed systems

- A formal observer is a subsystem designed to check distributed behaviors where:
  - Its SW is independent of the specific protocols to be checked in the considered system;
  - Its data are defined by the protocols to be checked and this data can be formally specified and verified.

Self-checking distributed systems

- Design of the system
  - write a description of the beavior of the system to be implemented;
  - Implement the system itself, i.e., the worker;
  - From the description of the worker, select (based on experience) that part of the behavior which should be observed and write a formal model of it.

Self-checking distributed systems

- The system is quasi self-checking if
  - It is an observer-worker system
  - The observer is a formal observer.
- For "real-life" only part of the system will be modelled.
- Formal model must be able to
  - Express simplified specifications of distributed systems
  - Support verification procedures
  - Be able to act as a basis for implementing the observer.

Few testing criteria exists for concurrent systems

- Number of execution orders grow exponentially with # synchronization primitives in tasks
  - Testing criteria needed to bound and selecting subset of execution orders for testing
- E.g. Branch / Statement coverage not sufficient for concurrent software
  - Still useful on serializations
  - Execution paths may require specific behavior from other tasks
- Data-flow based testing criteria has been adapted
  - E.g. define-use pairs

Determinism vs. Non-Determinism

- Deterministic systems
  - Controllability is high
    - input (sequence) suffice
  - Coverage can be claimed after single test execution with inputs
  - E.g. Filters, Pure "table-driven" real-time systems
- Non-Deterministic systems
  - Controllability is generally low
  - Statistical methods needed in combination with input coverage
  - E.g.
    - Systems that use random heuristics
    - Behavior depends on execution times / race conditions

Test execution in concurrent systems

- Non-deterministic testing
  - "Run, Run, Run and Pray"
- Deterministic testing
  - Select a particular execution order and force it
  - E.g. Instrument with extra synchronizations primitives
    - (No timing constraints make this possible)
- Prefix-based Testing (and Replay)
  - Deterministically run system to a specific (prefix) point
  - Start non-deterministic testing at that specific point