TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

IAF0030

Arvutitehnika erikursus I

Süsteemide usaldusväärsus ja veakindlus
Dependability and fault tolerance

Loeng 4
Testing Software and Systems

gert.jervan@pld.ttu.ee

Tallinn University of Technology
Department of Computer Engineering
Estonia

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

Software Testing

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

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

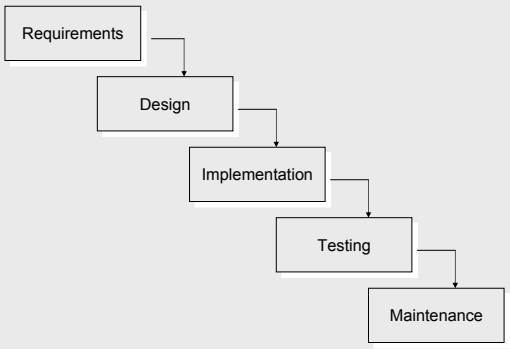
While the Universe is trying to create bigger and better idiots.

So far the Universe is winning

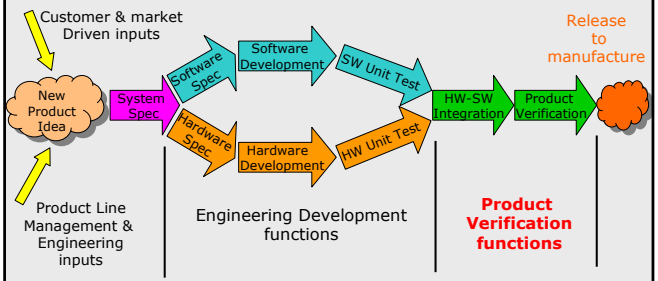
Software Testing Topics

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

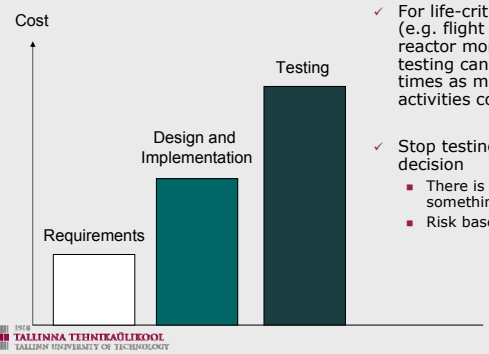
Software Life Cycle



The Product Development Cycle

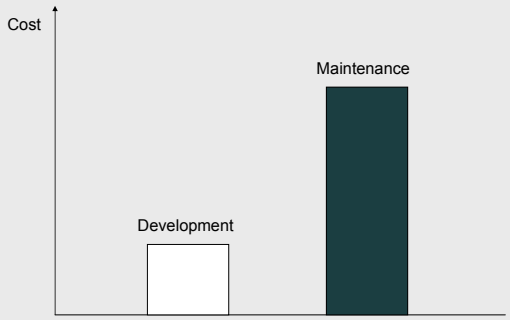


Software Development Costs



- 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

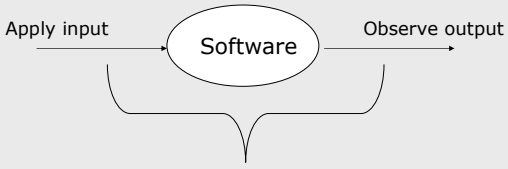
Formal Verification

- 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

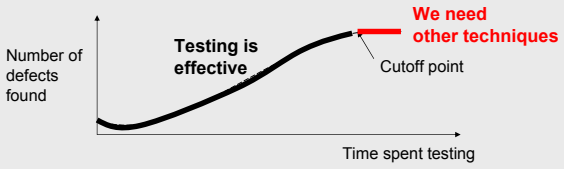
```
int foo(int x) {
        for (index = 1; index < 10000; index++)
                write(x);
}
```


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)

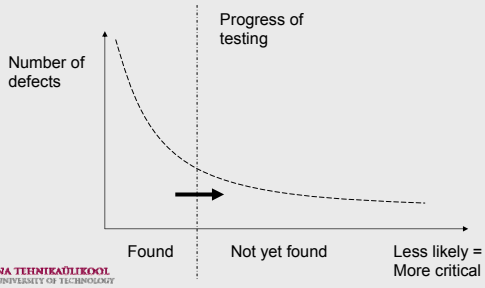
Economics of Testing (I)

- The characteristic S-curve for error removal



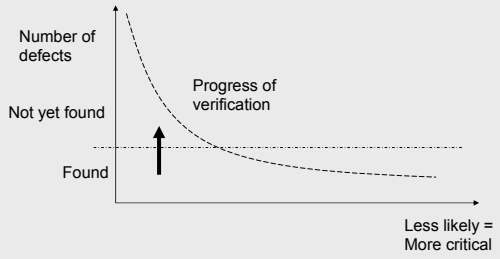
Economics of Testing (II)

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



Economics of Testing (III)

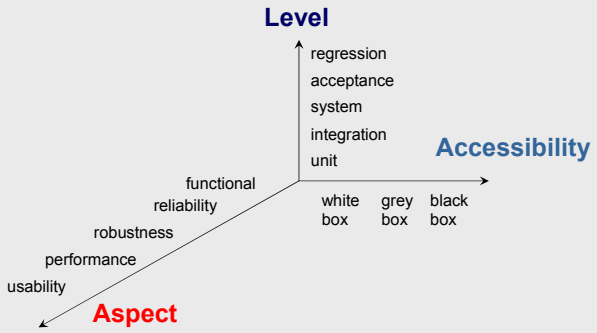
- Verification is insensitive to the probability of occurrence of errors



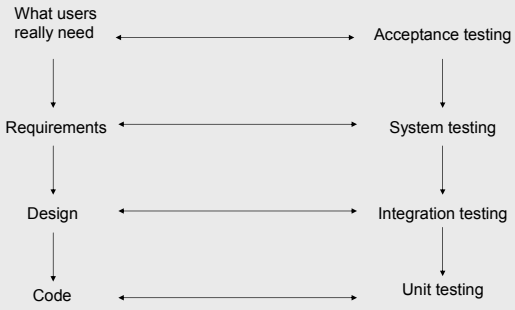
Fundamental Questions in Testing

- 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)

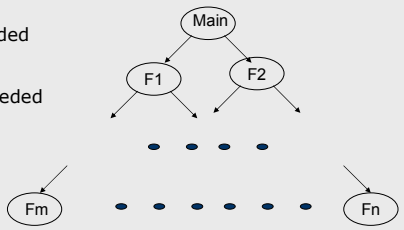
- Test case
  - Input, expected outcome, purpose
  - Selected according to a strategy, e.g., branch coverage
- Outcome
  - Pass/fail result
  - Log, i.e., chronological list of events from execution

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

Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

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.

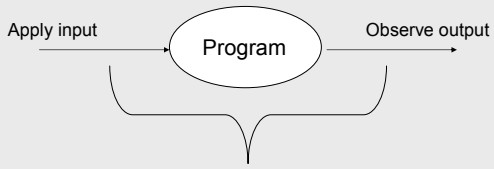
Accessibility of Testing (II)

- 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?

Coverage

- Estimate how much of the software's behavior is covered
- Coverage is a mean to estimate how rigorous is the testing effort
- We can use coverage information in order to guide the process of test generation (some times even automatically)

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.

Statement Coverage - Example

```
int a, b, sum;
int list1[10] = {00, 11, 22, 33, 44, 55, 66, 77, 88, 99};
int list2[10] = {99, 88, 77, 66, 55, 44, 33, 22, 11, 00};
cin >> a >> b;
if (a >= 0 && a <= 9)
    sum = list1[a];
if (b >= 0 && b <= 9)
    sum = sum + list2[b];
cout << sum << "\n";
```


Statement Coverage - Example

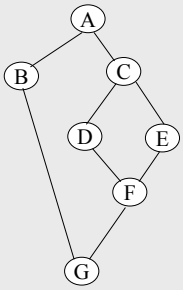
```
if (a >= 0 && a <= 9)
   sum = list1[a];
if (b >= 0 && b <= 9)
   sum = sum + list2[b];
```

- ✓ But statement coverage may not cater for all conditions
  - such as when a and b are beyond the array size.

Branch Coverage

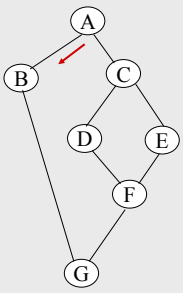
- ✓ Branch coverage is determined by the proportion of decision branches that are exercised by a set of proposed test cases.
- ✓ 100% branch coverage is where every decision branch in a unit is visited by at least one test in the set of proposed test cases.

Example – Branch coverage



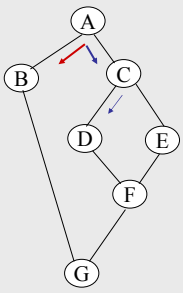
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



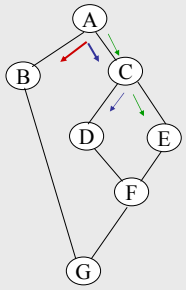
What branch coverage is achieved by ABG, ACDFG, ACEFG?

Example – Branch coverage



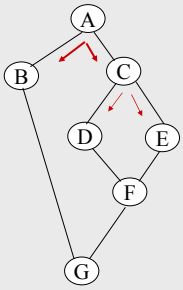
What branch coverage is achieved by ABG, ACDFG, ACEFG?

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.

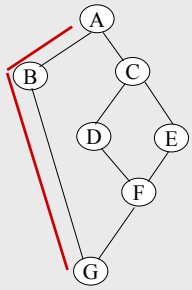
4 covered

So 4/4 = 100% branch coverage

Path Coverage

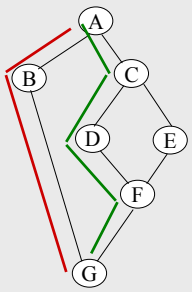
- Path coverage is determined by assessing the proportion of execution paths through a unit exercised by the set of proposed test cases.
- 100% path coverage is where every path in the unit is executed at least once by the set of proposed test cases.
- 100% path coverage is achieved by an ideal test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

Example – Path coverage



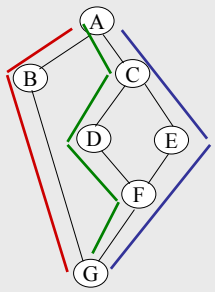
What path coverage is achieved by ABG, ACDFG, ACEFG?

Example – Path coverage



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

Example – Path coverage



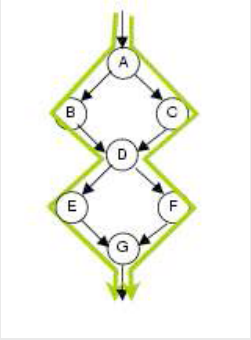
What path coverage is achieved by ABG, ACDFG, ACEFG?

3/3=100%

Coverage

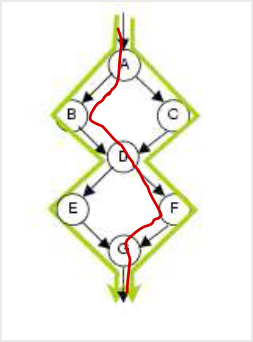
- It is possible to have 100% statement coverage without 100% branch coverage
- It is possible to have 100% branch coverage without 100% path coverage
- 100% path coverage implies 100% branch coverage and 100% branch coverage implies 100% statement coverage

An example



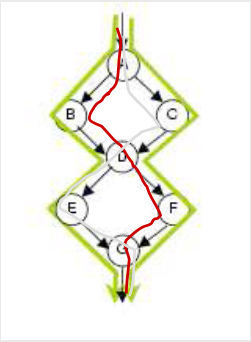
- Test cases covering ABDEG and ACDFG cover 4/4 branches (100%) and 7/7 statements (100%)
- They, however, only cover 2/4 paths (50%).

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

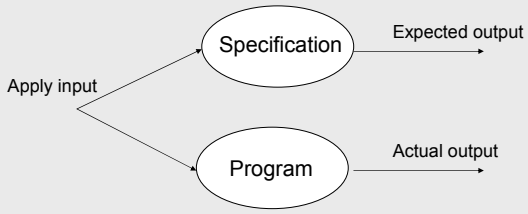
Mutation testing

- 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

Object class testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object;
  - Setting and interrogating all object attributes;
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

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

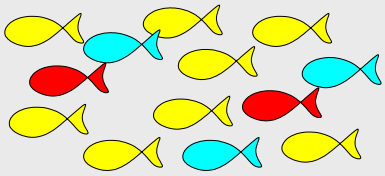
Tricks of the Trade

- 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

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

Test Selection vs. Test Adequacy



Mutation Testing Example

- Red fish = real program faults (unknown population)
- Blue fish = seeded faults (e.g., mutations) or representative behaviors (known population)
- Adequacy: count blue fish caught, estimate red fish
- Misuse for selection: use special bait to catch blue fish

Test Selection: Standard Advice

- Specification coverage is good for selection as well as adequacy
  - applicable to informal as well as formal specs
- + Fault-based tests
  - usually ad hoc, sometimes from check-lists
- Program coverage last
  - to suggest uncovered cases, not just to achieve a coverage criterion

The Importance of Oracles

- Much testing research has concentrated on adequacy, and ignored oracles
- Much testing practice has relied on the "eyeball oracle"
  - Expensive, especially for regression testing
    - makes large numbers of tests infeasible
  - Not dependable
- Automated oracles are essential to cost-effective testing

Sources of Oracles

- Specifications
  - sufficiently formal (e.g., SCR tables)
  - but possibly incomplete (e.g., assertions in Anna, ADL, APP, Nana)
- Design, models
  - treated as specifications, as in protocol conformance testing
- Prior runs (capture/replay)
  - especially important for regression testing and GUIs; hard problem is parameterization

What can be automated?

- Oracles
  - assertions; replay; from some specifications
- Selection (Generation)
  - scripting; specification-driven; replay variations
  - selective regression test
- Coverage
  - statement, branch, dependence
- Management

Design for Test: Principles

Adapted from circuit and chip design

- Observability
  - Providing the right interfaces to observe the behavior of an individual unit or subsystem
- Controllability
  - Providing interfaces to force behaviors of interest
- Partitioning
  - Separating control and observation of one component from details of others

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

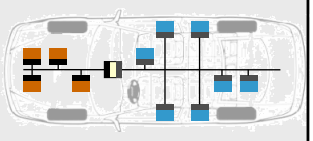
HW Testing

SW Testing

HW/SW Testing
(system testing)

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



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

Real-Time Systems Testing

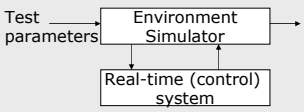
- Pure time-triggered systems
  - Deterministic
  - Test-methods for sequential software usually apply
- Fixed priority scheduling
  - Non-deterministic
    - Limited set of possible execution orders
  - Worst-case w.r.t timeliness can be found from analysis
- Dynamic (online) scheduled systems
  - Non-deterministic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

Testing Timeliness

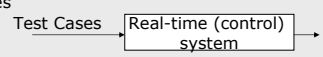
- Aim : Verification of specified deadlines for individual tasks
  - Test if assumptions about system hold
    - E.g. worst-case execution time estimates, overheads, context switch times, hardware acceleration efficency, I/O latency, blocking times, dependency-assumptions
  - Test system temporal behavior 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

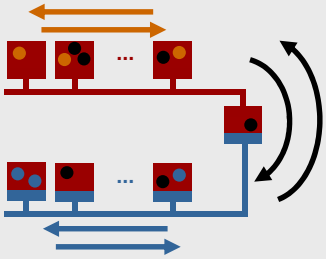
- System-level testing differs
  - Performed on target platform to keep timing
- Closed-loop testing
  - Test-cases consist of parameters sent to the environment simulator



- 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

- Aspects to design correct SW:
  - Reliability with which the SW specifications are adequately described and correctly implemented in the actual implementation.
  - Run-time checking

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

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Department of computer Engineering
ati.ttu.ee

Questions?