ON SOFTWARE TESTING

VO Software Engineering
Pölzleitner Anton
What is Software Testing?

- "Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service/system under test."
  (Kaner, Cem. “Exploratory Testing“.)

- Goals:
  - responds correctly to all kinds of inputs
  - performs its functions within an acceptable time
  - is sufficiently usable
  - can be installed and run in its intended environment
  - achieves the general result desired
Motivation

• Mars Orbiter (1998)
  • NASA m/s vs. Lockheed Martin ft/s

• Therac-25
  • Radiation therapy machine
  • 6 people died of radiation overdoses, many others injured
  • Two modes, essentially low and high doses of radiation
  • Previous models HW interlocks -> Therac25 SW interlocks
  • A one byte counter would overflow frequently and if an operator provided manual input at that exact moment the SW interlock would fail and provide a high dose, which gave patients severe radiation poisoning
Static vs. dynamic testing

- Executing vs. Non-executing the system under test (SUT)
- Static methods often implicit
  - Proofreading, Review, Walkthrough, Inspection
  - Source code structure/syntax checks of IDEs/Editors or (pre-) compilers
- Automated tools analyze source or object code without execution

- Static testing involves verification
- Dynamic testing involves validation
The box approach

- Illustrates the view the test engineer takes when designing test cases
- White-box testing
  - Tests internal structure
  - Inputs chosen to test specific paths of the code
- Black-box testing
  - Examining functionality without knowledge of internal implementation
- Gray-box testing
  - Involves knowledge of internal structure, but only for the purpose of designing tests
  - Tests executed at user/black box level
Testing levels

• Tests grouped by where they are added in the development process and/or by level of specificity

• Software Engineering Body of Knowledge (SWEBOK) standard:
  • Unit testing
  • Integration testing
  • Component interface testing
  • System testing
  • Operational Acceptance testing
Testing types

• Installation testing
• Compatibility testing
• Smoke and sanity testing
• Regression testing
• Acceptance testing
• Alpha testing
• Beta testing
• Functional vs non-functional testing
• Continuous testing
• Destructive testing

• Software performance testing
• Usability testing
• Accessibility testing
• Security testing
• Internationalization and localization
• Development testing
• A/B testing
• Concurrent testing
• Conformance testing or type testing
Testing process

- A common practice of software testing is that it is performed by an independent group of testers
  - After functionality is developed
  - Before it is shipped
  - Downside: Testing phase often misused as buffer to compensate delays
- Another practice is to start software testing at the same moment the project starts as a continuous process

- Traditional waterfall development model
- Agile or Extreme development model
- Top-down and bottom-up
Waterfall development model

• A sequential design process
The V model

- Also called the bent waterfall model
Agile or Extreme development model

- The Agile Manifesto
  - **Individuals and interactions** over Processes and tools
  - **Working software** over Comprehensive documentation
  - **Customer collaboration** over Contract negotiation
  - **Responding to change** over Following a plan

- Extreme programming (XP)
  - Programming in pairs
  - Extensive code review
  - Unit testing of all code
  - Avoiding programming of features until they are actually needed
  - Simplicity and clarity in code
  - Expecting changes in the customer's requirements
Test driven software development

- Unit tests written first (before the code/software)
- Often in pair programming (XP)
- Keep the unit small

Test driven development cycle:
- 1. Add a test
- 2. Run all tests and see if the new one fails
- 3. Write some code
- 4. Run tests
- 5. Refactor code
- Repeat
Top-down and bottom-up

• Top-down
  • Testing is conducted from main module to sub module.
  • If the sub module is not developed a temporary program called STUB is used

• Bottom-up
  • Integrated testing lowest level components (modules, procedures, and functions) tested first
  • Then integrated and used for testing of higher level components
  • If the main module is not developed a temporary program called DRIVERS is used
A sample testing cycle

• (For the Waterfall development model, but most of the practices commonly found in other development models)

• Requirements analysis
• Test planning
• Test development
• Test execution
• Test reporting
• Test result analysis
• Defect Retesting
• Regression testing
• Test Closure
Automated testing

- Continuous integration software runs tests automatically every time code is checked into a version control system
- Nightly builds
- Best suited for regression testing

- Testing tools help with included features such as:
  - Program monitors
    - Instruction set simulator
    - Hypervisor
    - Program animation (step-by-step execution)
  - Code coverage reports
  - Formatted dump or symbolic debugging
  - Benchmarks
  - Performance analysis (or profiling tools)
Coverage metrics

- Purpose:
  - Identify code that was not fully exercised during testing or cannot be reached through execution
  - Obtain summary measurements of testing completeness

- Function Coverage
- Statement Coverage
  - Also known as line coverage
  -Insensitive to some structures, for example:
    
    ```c
    int* p = NULL;
    if (condition)
        p = &variable;
    *p = 123;
    ```

  - No coverage for loop termination and consecutive switch cases
  - Insensitive to logical operators
Coverage metrics (contd.)

- **Decision/Branch Coverage**
  - Tests whether boolean expressions evaluate to true and false
  - Furthermore, covers switch statement cases, exception handler and all entry and exit points
  - Ignores logical operators within boolean expressions:
    ```java
    if (condition1 || condition2)
        statement1;
    else
        statement2;
    ```

- **Condition Coverage**
  - Reports true or false outcome of each condition independently
  - Condition is an operand without logical operators
  - Full condition coverage doesn‘t mean full decision coverage:
    ```java
    bool a;
    if (a && !a) ...
    ```
Coverage metrics (contd.)

- Modified condition/decision coverage (MC/DC)
  - Extends criteria with requirements that each condition should affect the decision outcome independently

```java
if (a or b) and c then
```

- Test set needed to satisfy MC/DC for above example
  - a=\textit{false}, b=\textit{false}, c=\textit{true}
  - a=\textit{true}, b=\textit{false}, c=\textit{true}
  - a=\textit{false}, b=\textit{true}, c=\textit{true}
  - a=\textit{false}, b=\textit{true}, c=\textit{false}

- Used in safety-critical applications (e.g. avionics software)
Even more coverage metrics

- **Path coverage**
  - Has every possible route through a given part of the code been executed?

- **Entry/exit coverage**
  - Has every possible call and return of the function been executed?

- **Loop coverage**
  - Has every possible loop been executed zero times, once, and more than once?

- **State coverage**
  - Has each state in a finite-state machine been reached and explored?

- **Parameter value coverage**
  - In a method taking parameters, have all the common values for such parameters been considered?
Coverage results (an example)
Concolic Testing

- Hybrid software verification technique
- Performs symbolic execution, treating program variables as symbolic variables, along a *concrete execution* (testing on particular inputs) path
- Involves automated theorem prover or constraint solver based on constraint logic programming to generate new concrete inputs (test cases) with the aim of maximizing code coverage
- Focus is finding bugs in real-world software, rather than demonstrating program correctness
Concolic Testing (contd.)

- **Example**
  ```c
  void f(int x, int y){
    int z = 2*y;
    if (x == 100000){
      if (x < z){
        assert(0); /* error */
      }
    }
  }
  ```

- Random testing would be infeasable to find the error
- x and y treated as symbolic variables
Concolic Testing (contd.)

- Limitations
  - Nondeterministic programs may follow different paths for equal inputs, which may lead to poor search and coverage results
  - Imprecise symbolic representations, incomplete theorem proving, and failure to search the most fruitful portion of a large or infinite path tree, may lead to poor results even in deterministic programs
  - Cryptographic primitives, generate very large symbolic representations that cannot be solved in practice
    
    ```
    if(md5_hash(input) == 0xdeadbeef)
    ```

- Results on large scale heavily tested software
  - KLEE was applied to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in the GNU COREUTILS that had been missed for over 15 years
Testing by simulation

- Simulation is the imitation of the operation of a real-world process or system over time.
- Requires a model to be developed, that represents key characteristics or behaviors/functions of selected physical or abstract systems or processes.
- Plays a key role in embedded system development.

Testing of embedded software

- From pure functional testing to ISS
- Control algorithms developed as block diagrams -> tested via simulation
- Additional challenges:
  - HW and SW engineers usually different group of people; mutual understanding is always an issue
  - Parallel development of HW and SW components
Modeling (Platform & Plant)

- Models are Software
- Component-based with rules of interaction
- Focus on timed models in cyber-physical systems
General simulation picture

- Model in the loop (MIL)
- Software in the loop (SIL)
- Hardware in the loop (HIL)

Host platform (e.g., a PC)
- Plant model
- Controller model
- Plant model
- Controller software
- Platform model

Dedicated hardware
- Plant model software

Target platform
- Controller software
Co-Simulation

- Integrating HW and SW earlier in the design cycle -> better time to market
- Co-simulation means executing the SW in conjunction with a virtual platform model and a plant model on a host computer

- Control semantics
- Communication semantics
Validator

- Platform-aware SIL
- Scheduling with preemption
- Accesses of shared resources
- Execution time

plant simulation in MATLAB/Simulink

communication interface (S-function)

validator simulation

set of sensors

set of instrumented tasks

variable monitoring

set of actuators

OS

timer

discrete event simulation

bus
What about multithreaded software?

• Additional issues with concurrent software
  • Race conditions
  • Deadlocks
  • …

• Solutions/approaches similar to singlethreaded software
  • Static analysis
  • Dynamic detection
  • Formal verification

• No holy grail so far…
Example of a bug

```c
pthread_mutex_t mutex, lock;
int A_count = 0;
int B_count = 0;

class A operation:
1: pthread_mutex_lock(&mutex);
2: A_count++;
3: if (A_count == 1) {
4:    pthread_mutex_lock(&lock);
5: }
6: pthread_mutex_unlock(&mutex);
7: performing class A operation;
8: pthread_mutex_lock(&mutex);
9: A_count--;
10: if (A_count == 0) {
11:    pthread_mutex_unlock(&lock);
12: }
13: pthread_mutex_unlock(&mutex);

class B operation:
1: pthread_mutex_lock(&mutex);
2: B_count++;
3: if (B_count == 1) {
4:    pthread_mutex_lock(&lock);
5: }
6: pthread_mutex_unlock(&mutex);
7: performing class B operation;
8: pthread_mutex_lock(&mutex);
9: B_count--;
10: if (B_count == 0) {
11:    pthread_mutex_unlock(&lock);
12: }
13: pthread_mutex_unlock(&mutex);
```
Inspect

- Framework that combines program analysis and model checking to dynamically verify multithreaded C programs

- Source code is instrumented at global accesses of resources

- Repeated execution of instrumented code to guide the program to systematically explore different interleavings of threads

- Exploring the state space of threads in DFS manner avoiding redundant interleavings
Thank you!