Programming Embedded Systems

Lecture 8

Overview of software testing

Wednesday Feb 8, 2012

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se

Lecture outline

- Testing in general
- Unit testing
- Coverage criteria
- System testing

- Some slides borrowed from the course "Testing, Debugging, Verification" at Chalmers

Overview

- Ideas and techniques of testing have become essential knowledge for all software developers.
- Expect to use the concepts presented here many times in your career.
- Testing is not the only, but the primary method that industry uses to evaluate software under development.

Overview (2)

- Testing field is HUGE
  - Many different approaches, many different opinions
  - Creating an exhaustive overview is a futile endeavour
- This lecture will cover some of the most important/common notions, in particular w.r.t. embedded systems

Correctness

- Software is called **correct** if it complies with its specification
    - Spec. might be *implicit* or *explicit,* *formal* or *informal,* *declarative* or *imperative,* etc.
    - Often: spec. is a set of requirements and/or use cases
- Software that violates spec. contains **bugs/defects**

Validation vs. Verification

The **V&V** of software engineering

- **Validation:** assessment whether program has intended behaviour (this can mean: check whether chosen *requirements* are correct or consistent)

$\subseteq$

- **Verification:** assessment whether program complies to specification/requirements

Validation vs. Verification (2)

- Testing is the most common approach to V&V
- Testing is form of **dynamic V&V**
  - Works through concrete execution of software
- Alternatives:
  - Stativ V&V: model checking, etc.
  - Code inspection/reviews

Faults, errors, failures

- **Fault:** abnormal condition or defect in a component (software or hardware)
- **Error:**
  - deviation from correct system state caused by a fault
  - human mistake when using the system
- **Failure:** observably incorrect behaviour of system, caused by a fault

Many different definitions exist!

Fault/failure example

```
for (;;) {
  if (GPIO_ReadInputD              SwitchPin)) {
    ++count;
  } else if (count != -      {
    GPIO_WriteBit(GPI  C, ON1Pin, Bit_RESET);
    GPIO_WriteBit(GPIOC            it_RESET);
    count = -1;   0
  }

  if (count == 10)           // 0.2 seconds
    GPIO_WriteBit(GPIOC, ON1Pin, Bit_SET);
  else if (count == 100) {   // 0.2 + 1.8 second
    GPIO_WriteBit(GPIOC, ON1Pin, Bit_RESET);
    GPIO_WriteBit(GPIOC, ON2Pin, Bit_SET);
  }

  vTaskDelayUntil(&lastWakeupTime,
                  PollPeriod / portTICK_RATE_MS);
}
```

Software **fault**: wrong RHS

Error: variable has wrong value

Failure: wrong value on pin, rockets are launched

Testing consists of ...

- designing test inputs
- running tests
- analysing results

- Done by **test engineer**
- Each of the steps can be automated

Test engineers vs. developers

- In practice, often different people
- Various opinions on this ...

Can you think of advantages/disadvantages?

Opinion 1
(Glenford J. Myers, ...)

- Programmer should avoid testing his/her own program (misunderstanding of specs carry over to testing)
- A programming organisation should not test its own programs
→ Conflict of interest

Opinion 2
(Kent Beck, Erich Gamma, ...)

- **Test-driven development:** Developers create tests *before* implementation
- Test cases are **form of specification**
- Re-run tests on all incremental changes

What is testing good for?
Bit of philosophy ...

Boris Beizer's levels of test process maturity

- **Level 0:** Testing = debugging
- **Level 1:** Purpose of testing: show that software works
- **Level 2:** Purpose of testing: show that software does not work
- **Level 3:** Purpose of testing: reduce risk due to software
- **Level 4:** Testing is mental discipline to develop high-quality software

Level 0:
testing = debugging

- Naïve starting point:
  debug software by writing test cases
  and manually observing the outcome
- No notion of *correctness*
- Does not help much to develop
  software that is reliable or safe

Level 1:
show that software works

- Correctness is (almost) impossible to achieve
- Danger: you are subconsciously steered towards tests likely to not fail the program
- What do we know if no failures?
  good software? or bad tests?
- No strict goal, no real stopping rule, no formal test technique

Level 2: show that software doesn't work

- Goal of testing is to find bugs
- Puts testers and developers into an adversarial relationship
- What if there are no failures?
- Practice in most software companies

Level 3: reduce risk

- Correct software is not achievable
- Evaluate potential risks incurred by software, minimise by
    - writing software appropriately
    - testing guided by risk analysis
- Testers + developers cooperate

Level 4:
testing as mental discipline

- Learn from test outcomes to improve development process
- Quality management instead of just testing
- V&V guides overall development
- Compare with spell checker: purpose is not (only) to find mistakes, but to improve writing capabilities

Categories of testing

Overview

- Acceptance testing
- System testing
- Integration testing
- Unit testing

- *Orthogonal dimension:* regression testing

Unit testing

- Assess software units with respect to **low-level unit design**
  - E.g., pre-/post-conditions formulated for individual functions
- As early as possible during development

Integration testing

- Assess software with respect to **high-level/architectural design**
- Focus on interaction between different modules
- Normally done after unit/module testing

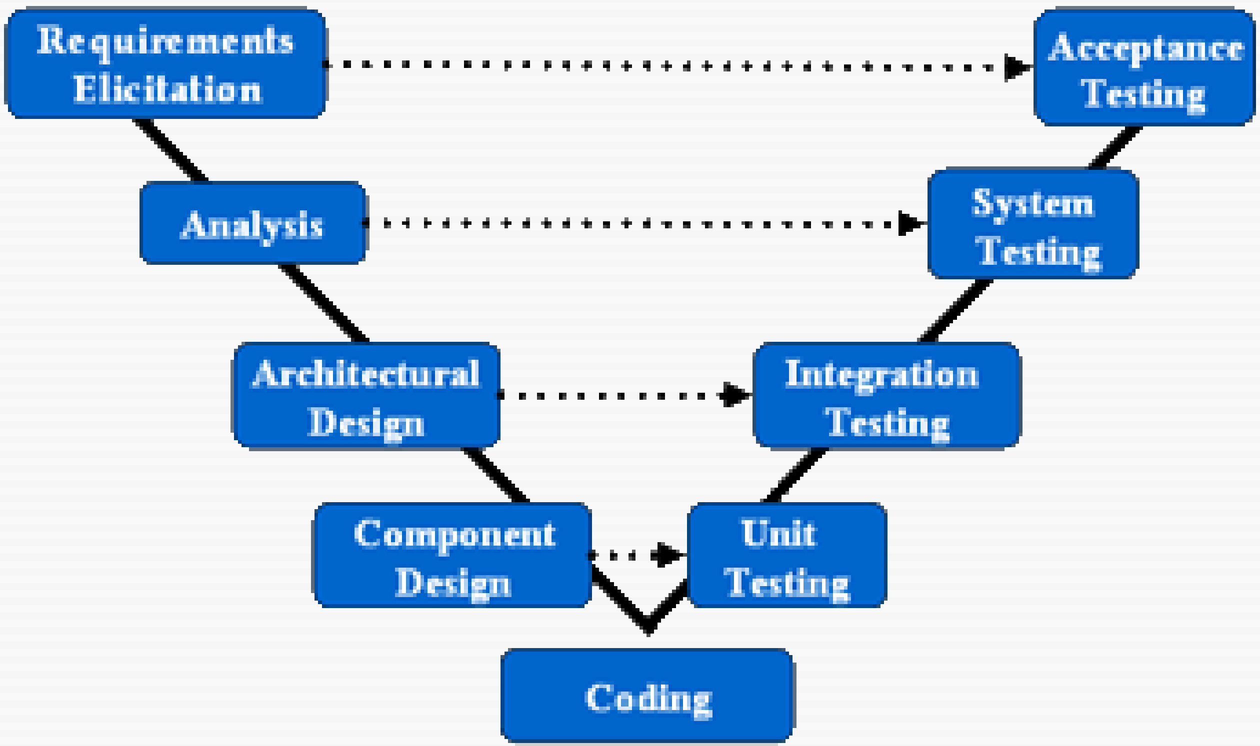
System testing

- Assess software with respect to **system-level specification**
  - Could be overall requirements, or more detailed specification
  - Testing by observing externally visible behaviour (no internals) → Black-box approach
- Usually rather late during development (but also applied to early versions of complete system)

Acceptance testing

- Assess software with respect to **user requirements**
- Can be done either by system provider or by customer (prior to transferring ownership)
- Late in development process

Testing w.r.t. V model



Regression testing

- Testing that is done after changes in the software.
- *Purpose:* gain confidence that the change(s) did not cause (new) failures.
- Standard part of the maintenance phase of software development.
- *Ideally:* every implemented feature is covered (preserved) by regression tests
- *Often:* regression tests are written as reaction to found bugs

Unit testing

Structure of a unit test case

- **Step 1:** Initialisation
  E.g., prepare inputs
- **Step 2:** Call functions of implementation under test (IUT)
- **Step 3:** Decision (oracle) whether the test succeeds or fails

Preferable for many reasons:
Tests can be re-run easily;
Correct behaviour is described formally

- Steps can be performed *manually* or *automatically*

Unit testing example

- Suppose we want to test this function:

```
void sort(int *array) { … }
```

- An executable **test case**:

```
int testSort(void) {
  int a[] = { 3, -1, 5 };

  sort(a);

  return (a[0] <= a[1] &&
          a[1] <= a[2]);
}
```

Initialisation

IUT invocation

Oracle, return true if test succeeded

Test oracles

- Oracles are often independent of particular test case
- Can sometimes be derived mechanically from unit specification
- Oracle is essential for being able to run tests automatically
- Common special case:
  Oracle just compares unit outputs with desired outputs

Common English patterns
(e.g., used in Elevator lab!)

Ambiguous; to clarify, write "either A or B" or "A or B, or both"

English	Logic	C (on `ints`)	Lustre (later in course)
A and B A but B	A & B	A && B	A `and` B
A if B A when B A whenever B	B => A	!B || A	B => A
if A, then B A implies B A forces B	A => B	!A || B	A => B
only if A, B B only if A	B => A	!B || A	B => A
A precisely when B A if and only if B	A <=> B	A == B, (A && B) || (!A && !B)	A = B
A or B either A or B	A (+) B (exclusive or)	A != B, A ^ B, (A && !B) || (!A && B)	A `xor` B
A or B	A v B (logical or)	A || B	A `or` B

Sets and suites

- **Test set:** set of test cases for a particular unit
- **Test suite:** union of test sets for a number of units

Automated, repeatable testing

- By using a tool you can automatically run a large collection of tests
- The testing code can be integrated into the actual code
(side-effect: documentation)
- After debugging, the tests are rerun to check if failure is gone
- Whenever code is extended, all old test cases can be rerun to check that nothing is broken (regression testing)

Automated, repeatable testing (2)

- Supported by unit testing frameworks (also called **test harness**): e.g., “xUnit”
  - SUnit: SmallTalk
  - JUnit: Java
  - CppUnit: C++
- One of the most common commercial frameworks: C++Test, Parasoft → can be integrated with MDK-ARM

Construction of test suites

- **Black-box** (specification-based)
  - Derive test suites from external descriptions of the software, including specifications, requirements, design, and input space knowledge
- **White-box** (implementation-based)
  - Derive test suites from the source code internals of the software, specifically including branches, individual conditions, and statements
- Many approaches in between (e.g., **model-based**)

Test suite construction is related to *coverage criteria*

- Tests are written with a particular goal in mind
  - Exercise program code thoroughly (white-box); or
  - Cover input space thoroughly (black-box)

Assessing quality of test suites: coverage criteria

(used in various kinds of testing)

Common kinds of coverage criteria

- Control-flow graph coverage
- Logic coverage
- Input space partitioning
- Mutation coverage

Structural coverage

- In embedded software, it is often **required** to demonstrate coverage
  - E.g., DO-178B (avionics standard), level A requires some level of MC/DC coverage

Input space partitioning

- Based on **input domain modelling (IDM):**
  abstract description of input space
  - Input space partitioned into blocks (sets of input values)
  - Values of each block are equivalent w.r.t. some characteristic
- **Coverage criteria:** has each block been covered by test cases?

Interface-based IDM

- Characteristics derived from signature, datatypes
- E.g., for integer inputs:
    - interesting blocks are zero, positive, negative, maximum number, etc.

Functionality-based IDM

- Characteristics derived from intended program functionality
- E.g., different expected program outputs

Example: triangle classification

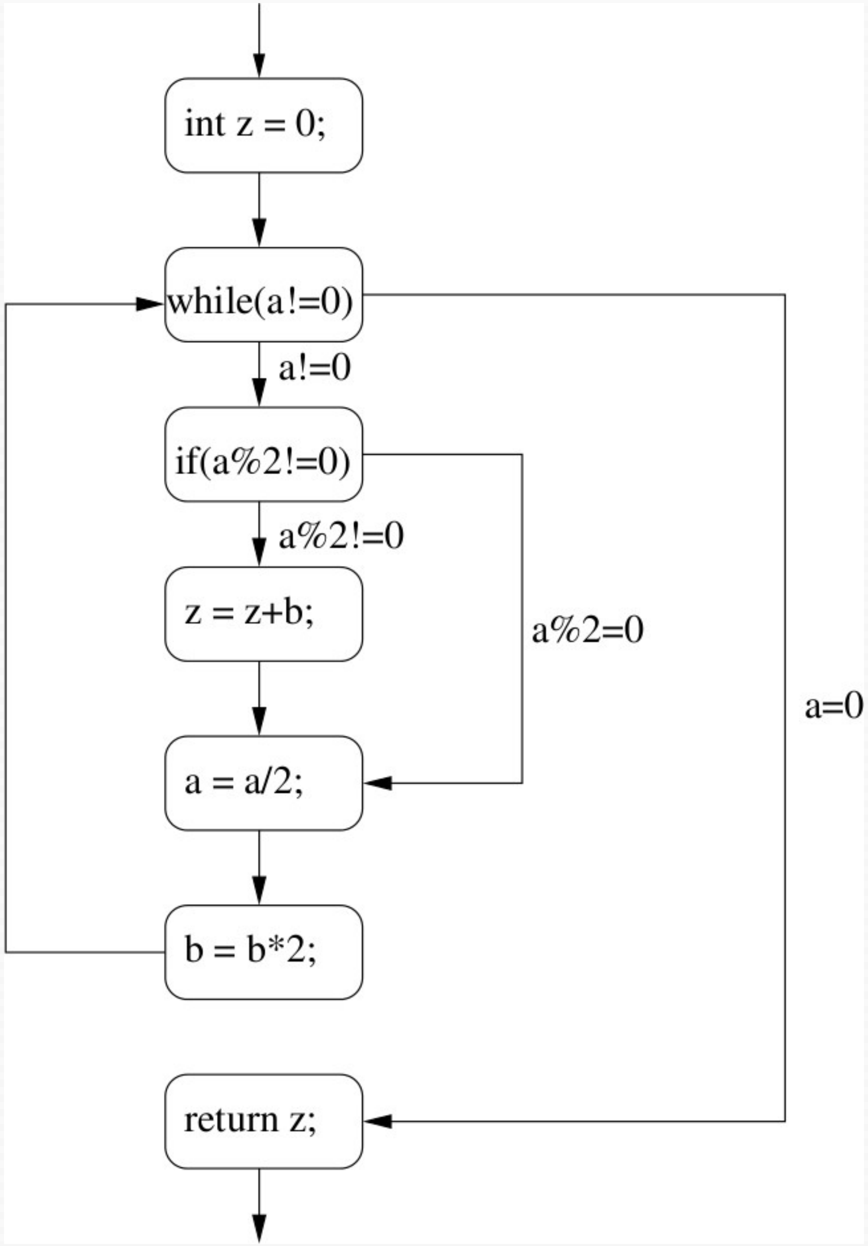
- Consider program

```
typedef enum { Scalene,
               Isosceles,
               Equilateral,
               Invalid } TriType;

TriType determineType(int length1,
                      int length2,
                      int length3) {…}
```

Which test inputs would you choose?

Control-flow graphs (CFGs)




```
int mult(int a, int b){
  int z = 0;
  while ( a != 0) {
    if ( a % 2 != 0) {
      z = z+b;
    }
    a = a /2;
    b = b *2;
  }
  return z;
}
```

Graphical representation of code:

- Nodes are control-flow locations + statements
- Edges denote transitions + guards

Common notions in CFGs

- **Execution Path:** a path through a CFG that starts at the entry point and is either infinite or ends at one of the exit points
- **Path condition:** a path condition PC for an execution path p within a piece of code c is a condition that causes c to execute p if PC holds in the prestate of c
- **Feasible execution path:** path for which a satisfiable path condition exists

Examples!

Statement coverage (SC)

(a kind of CFG coverage)

- A test suite TS **achieves SC** if for every node n in the CFG there is at least one test in TS causing an execution path via n

- Often quantified: e.g., 80% SC

- **Can SC always be achieved?**

Branch coverage (BC)

- A test suite TS **achieves BC** if for every edge e in the CFG there is at least one test in TS causing an execution path via e.

- BC subsumes SC:
  if a test suite achieves BC (for a **strongly connected** CFG), it also achieves SC

Path coverage (PC)

- A test suite TS **achieves PC** if for every execution path ep of the CFG there is at least one test in TS causing ep

- PC subsumes BC
- PC cannot be achieved in practice
  - number of paths is too large (for `mult` example, $\approx 2^{31}$)
  - paths might be infeasible

Decision coverage (DC)

(a kind of logic coverage)

- **Decisions** D(p) in a program p: set of *maximum* boolean expressions in p
- E.g., conditions of `if`, `while`, etc.
- But also other boolean expressions:

```
A = B && (x >= 0);
```

Precise definition is subject of many arguments:
only consider decisions that program branches on?

(`B&&(x>=0)` is a decision, `B` and `(x>=0)` are not)

Decision coverage (DC) (2)

- **NB:** multiple occurrences of the same expression are counted as different decisions!
  E.g.

```
if (x >= 0) { … }
// …
if (x >= 0) { … }
```

Two decisions

Decision coverage (DC)

- For a given decision d, DC is satisfied by a test suite TS if it contains at least one test where d evaluates to false, and one where d evaluates to true (might be the same test)
- A test suite TS **achieves DC** for a program p if it achieves DC for every decision d in D(p)

DC example

- Consider decision

  ```
  ((a < b) || D) && (m >= n * o)
  ```

- **Inputs to achieve DC?**

TS achieves DC if it triggers executions

```
a = 5, b = 10, D = true, m = 1, n = 1, o = 1
```

and

```
a = 10, b = 5, D = false, m = 1, n = 1, o = 1
```