Testing, debugging and verification

Notes based on lectures for DIT084 (Testing, debugging and verification)
at Gothenburg University, Autumn 2023

Sebastian Pålsson

Last updated: **December 25, 2023**

Contents

1. Introduction

How do we judge failure in software development? In general, we need a measure to judge failure. This is given by means of satisfaction of a specification. A specification is a set of requirements that a system must satisfy. A specification can be given in many forms, such as a formal mathematical description, a set of examples, or a set of requirements in natural language. The specification is the basis for testing, debugging and verification.

In simple terms:

$$\text{Spec} = \text{Require} + \text{Ensure} \qquad [1]$$

where **Require** is the set of requirements the program must satisfy in order to behave correctly, and **Ensure** is the set of guarantees that the program must satisfy.

Example

Imagine a programs that sorts an array of integers. The program must satisfy the following specification:
- **Require:** Accept only input as an array of integers.
- **Ensure:** Return a sorted array of integers.

1.1. The contract methaphor

Same priciple as a **legal contract**, between supplier and client.

- **Supplier:** aka implementer, here a *class or method* in Java.
- **Client:** mostly *caller object or human user* calling main-method.
- **Contract:** One or more *pairs of ensures-requires clauses*, defining *mutual obligations* between supplier and client.

*"If the caller (the client) of the method **m** (the supplier) gives inputs which fulfils the required **precondition**, then the method **m** ensures that the **postcondition** holds after **m** finishes execution."*

1.2. Specification, failure, correctness

A failure/correctness is relative to a specification.

- A method **fails** when called in a *state fulfilling the required precondition* of its contract and it *does not terminate in a state fulfulling the postcondition* to be ensured.
- A method is **correct** whenever it is started in a *state fulfilling the required precondition*, then it *terminates in a state fulfilling the postcondition to be ensured.*
- **Correctness:** Proving absence of failures.

2. Testing

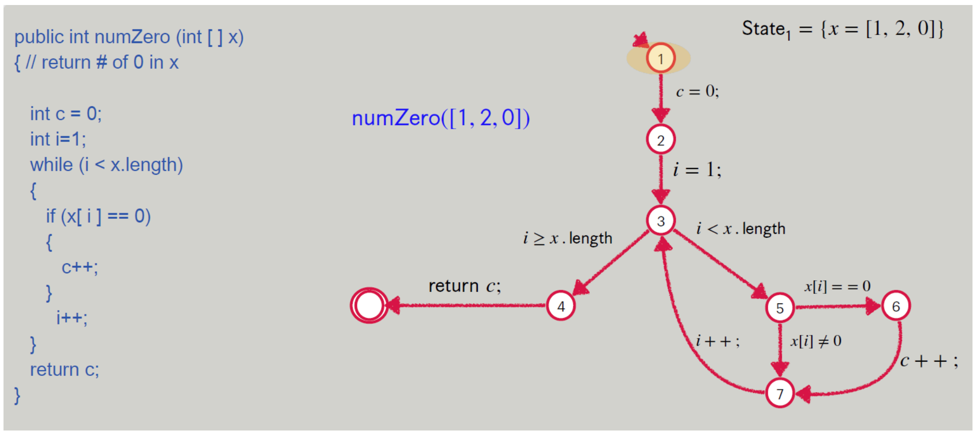
Some terminology regarding faults, errors & failures:

- **Fault:** a static defect in the software.
- **Error:** an incorrect internal state that is the manifestation of some fault.
- **Failure:** external, incorrect behaviour with respect to expected behaviour.

2.1. The State of a program

A program can be seen as a function that maps a state to a state. The state of a program is the values of all variables at a given point in time.

Inputs in the program in Figure 1 does not always trigger failures. This is the case for most programs!



```
public int numZero (int [ ] x)
{ // return # of 0 in x

  int c = 0;
  int i=1;
  while (i < x.length)
  {
    if (x[ i ] == 0)
    {
      c++;
    }
     i++;
  }
  return c;
}
```

Figure 1: A graph showing the states of a program

2.1.1. Program to graph

How does one convert a program to a state graph?

A labelled graph G is of the form

$$G = (N, E, L) \qquad [2]$$

where N is a set of nodes, E is a set of edges and L is a set of labels. $E \subseteq N \times L \times N$

A program P is a set of statements

$$P = \{s_1, s_2, ..., s_n\} \qquad [3]$$

The following function is used to transform the program to a graph:

$$[\cdot]^{0,f} : S \to 2^E \qquad [4]$$

The function takes an initial state 0 and a final state f and a **statement** and **return a set of edges**. We use the special state $\bot$ to mean an exit state (for return statements).

The transformation rule are as follows:

$$[s_1; s_2]^{0,f} = [s_1]^{0,1} \cup [s_2]^{1,f} \qquad [5]$$

where 1 is a fresh state.

Example

Some examples of transformations:

- $[x = a;]^{0,f} = \{(0, f = a, f)\}$
- $[\text{return } a;]^{0,f} = \{(0, \text{return } a, \bot)\}$
- $[\text{if}(b)\{s_1\}]^{0,f} = \{(0, b, 1), (0, !b, f)\} \cup \{s_1\}^{1,f}$
- $[\text{while}(b)\{s_2\}]^{0,f} = \{(0, b, 1), (0, !b, f)\} \cup \{s_2\}^{1,0}$

2.2. Complexity of testing (and why we need models)

No other engineering field builds products as complicated as software. The term correctness has no meaning. Like other engineers, we must use abstraction to cope with complexity. We use **discrete mathematics** to raise our level of abstraction.

Abstraction is the purpose of *Model-Driven Test Design* (MDTD). The "model" is an abstract structure.

2.2.1. Software testing foundations

Testing can only show the presence of failures, not their absence.

- **Testing:** Evaluate software by observing its execution.
- **Test Failure:** Execution of a test that results in a software failure.
- **Debugging:** The process of finding a fault given a failure.

Not all inputs will "trigger" a fault into causing a failure

2.2.2. RIPR model (or fault and failure model)

Four conditions necessary for a failure to be observed:

- **Reachability:** The location or locations in the program that contain the fault must be reached.
- **Infection:** The state of the program must be incorrect.
- **Propagation:** The infected state must cause some output/final state to be incorrect.
- **Reveal:** The tester must observe part of the incorrect state.

The RIPR model is a framework for understanding and categorizing software bugs. It stands for Reach, Infection, Propagation, and Reveal. Reach refers to the execution reaching the defect, Infection is when the defect causes an incorrect program state, Propagation is when this incorrect state leads to incorrect behavior or output, and Reveal is when this incorrect behavior is observed.

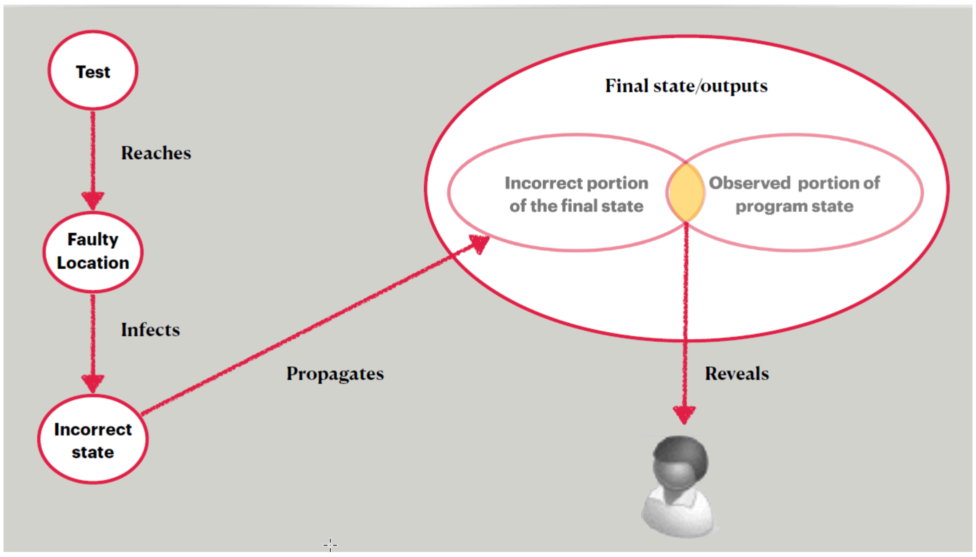


Figure 2: The RIPR model

2.2.3. The V model

The V-model emphasizes a parallel relationship between development and testing stages. Each development stage has a corresponding testing phase, ensuring that issues are identified and fixed early. *It's called the V-model due to its V-like structure, representing the sequence of execution of processes.*

Each testing phase uses a different type of testing level:

- **Acceptance testing:**: Assess software with respect to user requirements
- **System Testing:**: Assess software with respect to system-level specification.
- **Integration Testing:** Assess software with respect to high-level design
- **Unit Testing:** Assess software with respect to low-level unit design

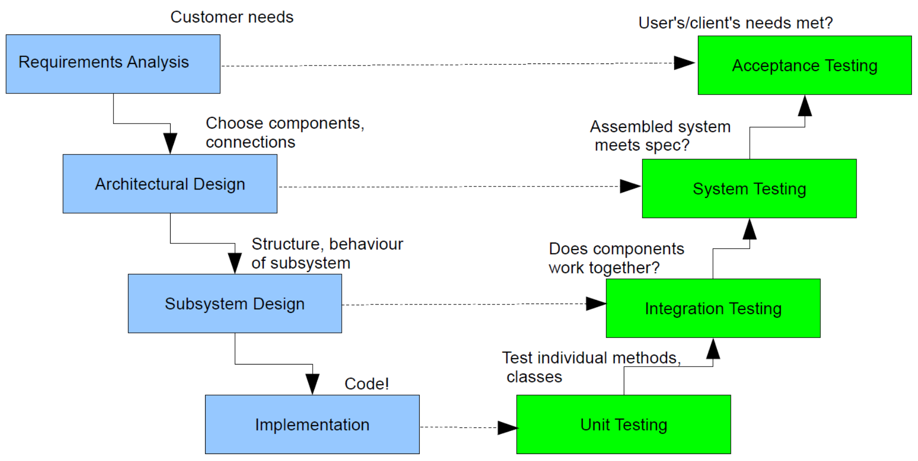

Customer needs
Requirements Analysis
User's/client's needs met?
Acceptance Testing
Choose components, connections
Architectural Design
Assembled system meets spec?
System Testing
Structure, behaviour of subsystem
Subsystem Design
Does components work together?
Integration Testing
Test individual methods, classes
Code!
Implementation
Unit Testing

Figure 3: The V model

2.3. Test automation

The use of software to control *the execution* of tests, the *comparison* of actual outcomes to predicted outcomes, the *setting up* of test preconditions, and other test *control* and test reporting functions.

- Reduces cost
- Reduces human error
- Reduces variance in test quality from different individuals
- Reduces cost of **regression testing**

2.4. Regression testing

Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes such as enhancements, patches or configuration changes, have been made to them.

- Orthogonal to other mentioned tests
- Testing that is done **after changes** in the software (updates)
- Standard part of maintenance phase of software development

The purpose of regression testing is to gain confidence that changes did not cause new failures.

2.5. Software testability

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. *In simple terms: how hard is it to finds faults?*

Testability is a condition of two factors:
- How to provide the test values to the software?
- How to observe the results of test execution?

2.6. Observability

How easy it is to observe the behaviour of a program in terms of its outputs, effects on the environment and other hardware and software components.

Software that affects hardware devices, databases, or remote files have low observability!

2.7. Controllability

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviours.

- Easy to control software with inputs from keyboards
- Inputs from hardware sensors or distributed software is harder

2.8. Test suit construction

A test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours.

- Most central actitivy of testing
- Determines if we have enough test cases (stopping criteria)
- Determines if we have the right test cases (represenative test cases)

The quality of test suites defines the quality of the overall testing effort. When presenting test suites, we show only relevant parts of test cases.

2.8.1. Black box testing

Deriving test suites from external descriptions of the software, e.g.
- Specifications
- Requirements / Design
- Input space knowledge

2.8.2. White box testing

Deriving test suites from the source code, e.g.

- Conditions
- Branches in execution
- Statements

Modern techniques are a hybrid of both black- and white box

2.8.3. Coverage criteria