Testing, debugging and verification

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

Sebastian Pålsson

Last updated: **January 07, 2024**

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:

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

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!

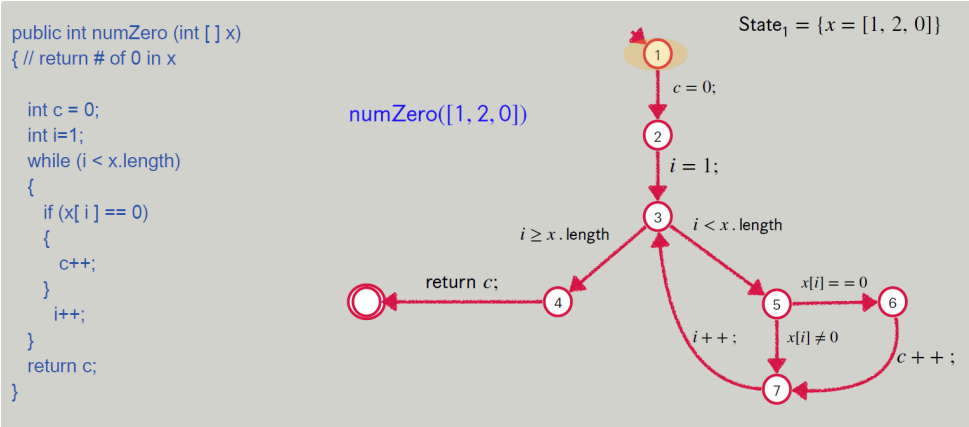

Figure 1: A graph showing the states of a program

2.1.1. Program to graph

How does one convert a program to a state graph?

A labelled graph G is of the form

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

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

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

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

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

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 $\perp$ to mean an exit state (for return statements).

The transformation rule are as follows:

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

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.

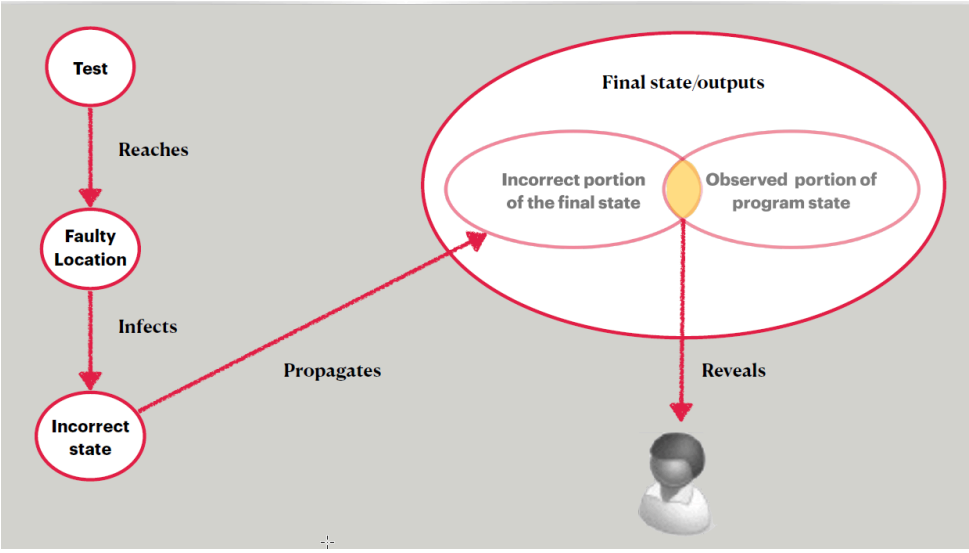

Figure 2: The RIPR model

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

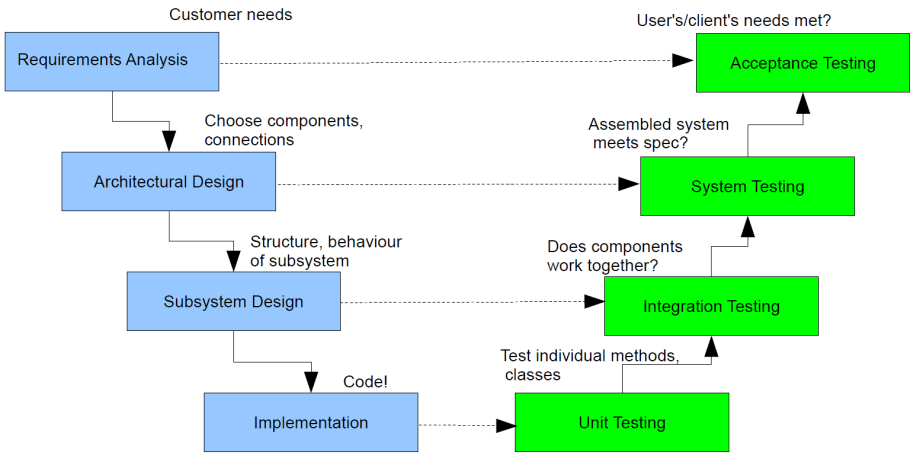

Figure 3: The V model

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

2.9. Coverage criteria

Most metrics used as quality criteria for test suites, describe the degree of some kind of coverage. These metric are called coverage criteria.

These are crucial for testing *saftey critical software*.

There are certain categories of coverage criteria:

- Control flow graph coverage (white box)
- Logic coverage (white box)
- Input space partitioning (black box)

2.10. Control flow graph

Represent a method to test as a graph, where:

- Statements are nodes
- Edges describe control flow between statements
- Edges are labelled with conditions

Important

Rules of transformation can be read in Section 7.3.1 "Structural Graph coverage for source code" from the book: Introduction to software testing. **This is required**

2.10.1. Control flow graph notions

- **Execution path:** a path through the control flow graph that starts at the entry point and is either

infinite or ends at one of the exit points.

- **Path condition:** a condition causing execution to take some path p
- **Feasible execution path:** an execution path for which a satisfiable path condition exists.

Note

A branch or statement is feasible if it is contained in at least one feasible execution path.

2.11. Types of coverage criteria

2.11.1. Statement coverage (SC)

Satisfied by a test suite *TS*, *iff* for every node n in the control flow graph there is at least one test in *TS* causing an execution path via n.

2.11.2. Branch coverage (BC)

Satisfied by a test suite *TS*, *iff* for every edge e in the control flow graph there is at least one test in *TS* causing an execution path via e. *Note that this is a stronger criterion than statement coverage.*

2.11.3. Path coverage (PC)

Satisfied by a test suite *TS*, *iff* for every feasible execution path p in the control flow graph there is at least one test in *TS* causing an execution path via p. *Note that this is a stronger criterion than branch coverage.*

2.11.4. Logic coverage (LC)

Satisfied by a test suite *TS*, *iff* for every predicate p in the control flow graph there is at least one test in *TS* causing an execution path via p.

Logical (boolean) expressions can come from many sources. We focus on decisions in the source code *(if, while, for, etc.)*.

2.11.5. Decision coverage (DC)

Let the decisions of a program p, $D(p)$, be the set of all logical expressions which p branches on.

For a given decision d, *DC* is satisfied by a test suite *TS* if it:

- Contains at least two tests,
- one where d evaluates to true,
- one where d evaluates to false

For a given program p, *DC* is satisfied by *TS* if it satisfies *DC* **for all** decisions $d \in D(p)$.

Example

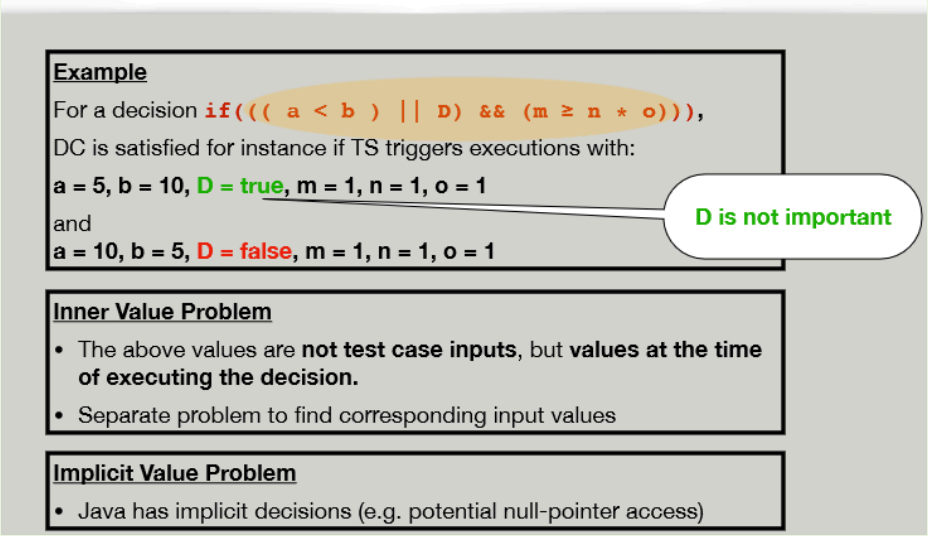

Figure 4: Decision coverage example

2.11.6. Condition Coverage (CC)

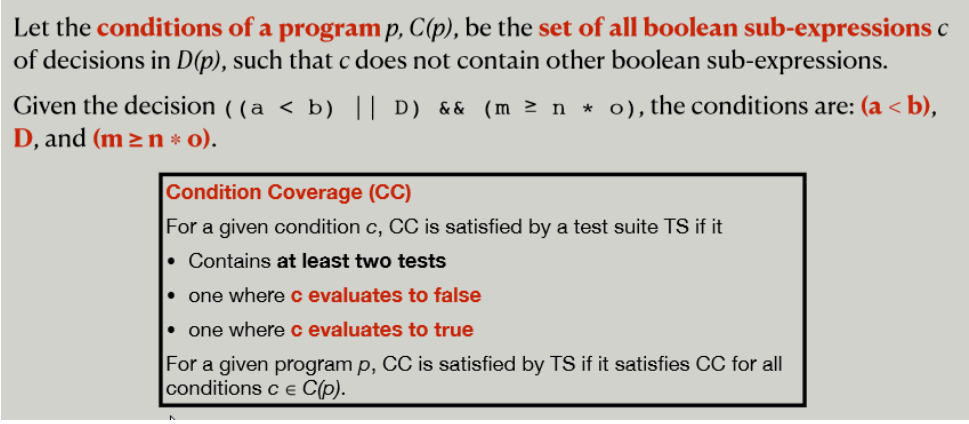

Figure 5: Condition coverage

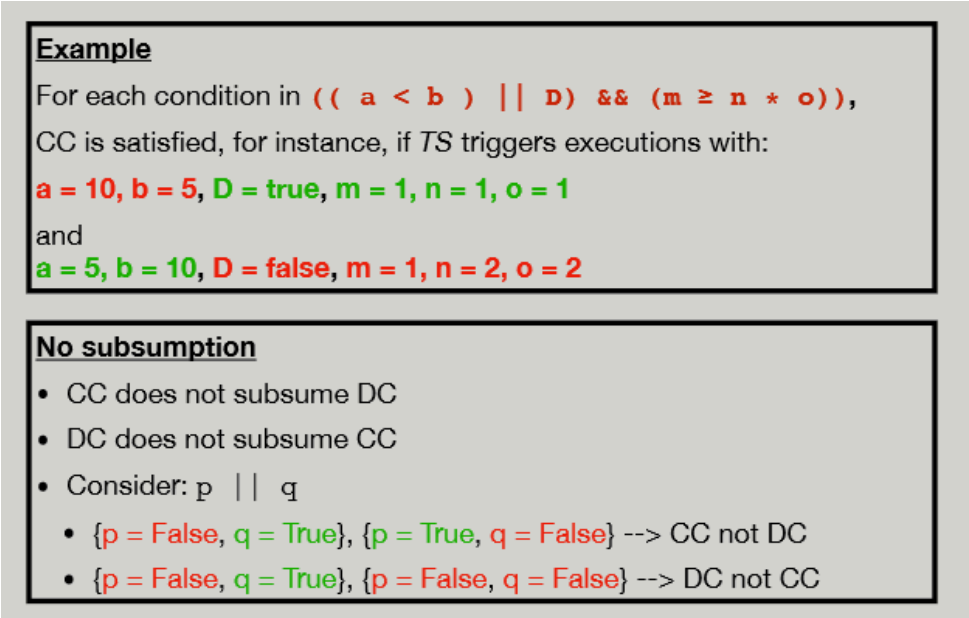

Figure 6: Condition coverage example

2.11.7. Modified Condition Decision Coverage (MCDC)

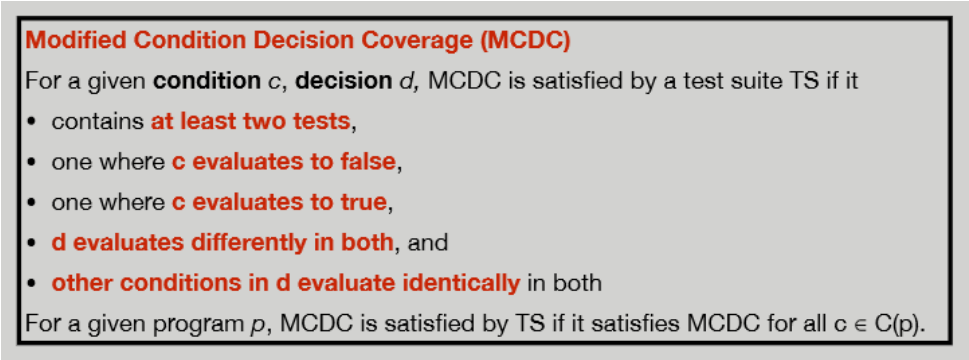

Figure 7: Modified Condition Decision Coverage

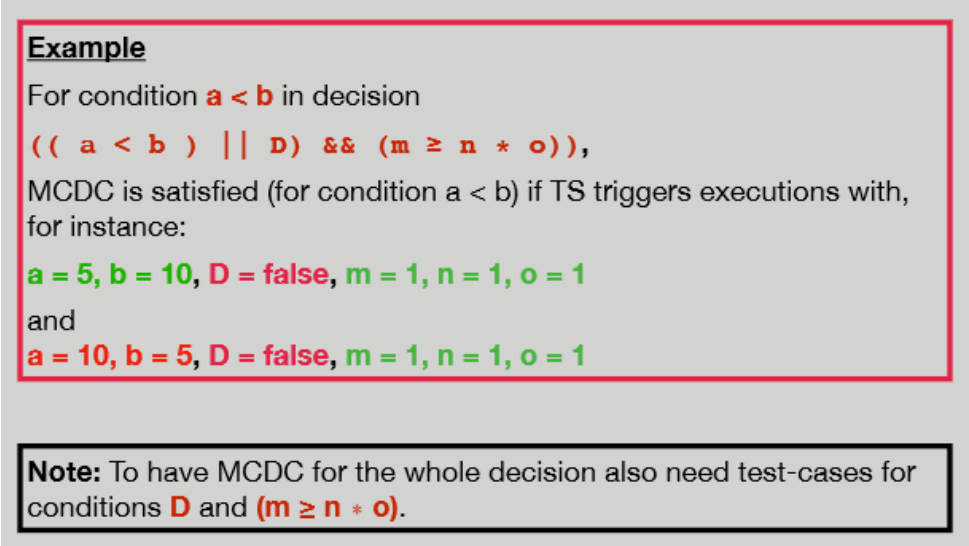

Figure 8: Modified Condition Decision Coverage example

2.12. Input domain modeling

- The input domain D: the possible values that input parameters can have.

- A partition q defines a set of equvilence classes (or simply blocks B_q) over D

A partition q satisfies: (completeness)

$$\bigcup_{b \in B_q} b = D \quad [6]$$

Blocks are pairwise disjoint:

$$b_i \cup b_j = \emptyset, \qquad i \neq j, \qquad b_i, b_j \in B_q \quad [7]$$

- Test values in the same block are assumed to contain *equally useful values.*
- Test cases contain values from each block.

Example

- Consider the domain of integer arrays.
- Are the following blocks a valid partitioning?
  - b1 = sorted in ascending order
  - b2 = sorted in descending order
  - b3 = arbitrary order
- Answer: no!
  - The array [1] belongs to all blocks
  - Unclear whether the array null belongs to any block

Figure 9: Partitioning example

2.12.1. Strategies for generating blocks

- Valid vs. invalid values: all valid and all invalid (completeness)
- Sub-partition: valid values serving different functionality.
- Boundaries: values at or close boundaries often cause problems (stress testing).
- Normal use (happy path): the desired outcome.
- Missing blocks vs overlapping blocks (complete vs disjoint)
- Special values Pointers: (null and not null), collection: (empty and non empty), integer (zero a special value).

Example

- Identify testable functions. (triang() has one testable function)
- Identify all parameters of that affect the testable functions. (The sides of triang)
- Generate input domain model (IDM) (Abstraction interpretation of the input). (Characteristics of the triangle)
- Generate complete set of blocks for each characteristic. (e.g. Side relation to zero)
- Select representative values from each block.

```
public enum Triangle {Scalene, Isosceles, Equilateral, Invalid}

public static Triangle triang(int Side1, int Side2, int Side3)

//Side1, Side2, Side3 represent the lengths of the triangle sides
// Returns the appropriate enum value
```

Figure 10: Example: triang(0)

Partition	b1	b2	b3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Length of Side 2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Length of Side 3"	greater than 1	equal to 1	equal to 0	less than 0

Parameter	b_1	b_2	b_3	b_4
Side 1	2	1	0	-1
Side 2	2	1	0	-1
Side 3	2	1	0	-1

If Side are floats, this is defective partition

Figure 11: Example: triang(0) - Solution

All combinations of blocks from all characteristics must be used. The number of tests is the product of the number of blocks for each partition. *All Combinations Coverage* ***ACoC***

For *triang()*, we have three characteristics, each with 4 blocks. Thus, we need $4 \cdot 4 \cdot 4 = 64$ test cases.

Recall: *different choices of values from the same block are equivalent from testing perspective*.

3. Debugging

- How to systematicly find source of failer
- Test-case to reproduce errors
- Finidng a small failing input (if possible)
- Observing exceution: Debbuggers and Logging
- Program dependencies: data- and control

3.1. Motivation

- Debuging needs to be systematic
- Debuging may involve large inputs

- Programs may have have thousands of memory locations
- Program may pass through milions of states before failure occurs

3.2. The Steps Of Debugging

1. Reproduce the error and try to understand the cause.
2. Isolate and minimise the diffrent factors. (**Simplification**)
3. Eyeball the code, where could it be? (**Reason backwards**)
4. Devise experiments to test your hypothesis. (**Test hypothesis**)
5. Repeat step 3 and 4 until the cause of the bug is determined
6. Fix the bug and verify the fix
7. Create a regression test (See below)

Regression testing is a type of software testing that checks if recent code changes have negatively impacted existing features. It involves re-running previously created test cases after code modifications to catch any unintended side effects and ensure the ongoing stability of the software.

3.3. Problem Simplification

As described in the diffrent steps of debugging, simplification is a way to determine the bug. The idea behing simplification is to minimize the failing input, so that it will be easier to understand what inputs causes the bug.

Simplification can be reached by **Divide-and-Conquer**, where you ->

1. Cut away one half of the test input
2. Check if any of the halves still exhibit failure.
3. Repeat, until minimal input has been found.

Although this works in some scenarios, this method has the following problems ->

- Tedious to re-run test manually
- Boring, cut and paste, re-run ...
- What, if none of the halves exhibits a failure?

Because of the problems with **Divide-and-Conquer**, in most cases automation of input simplificartion is more favourable.

3.3.1. AUTOMATION OF INPUT SIMPLIFICATION

Automation of input simplification in debugging involves using tools or algorithm to automatically reduce the complexity of input data. This helps identify and isolate bugs more efficiently, especially in large or complex software systems.

One exampel of such a algorrithm is **Delta Debugging**. Delta debugging is a software debugging technique that aims to isolate and identify the cause of a failure in a program by systematically narrowing down the input that triggers the failure. The term “delta” refers to the minimal change needed to reproduce the failure.

The algorithm **Delta debugging** or **DD-Min** as it is also called, works as seen in figure 4 below.

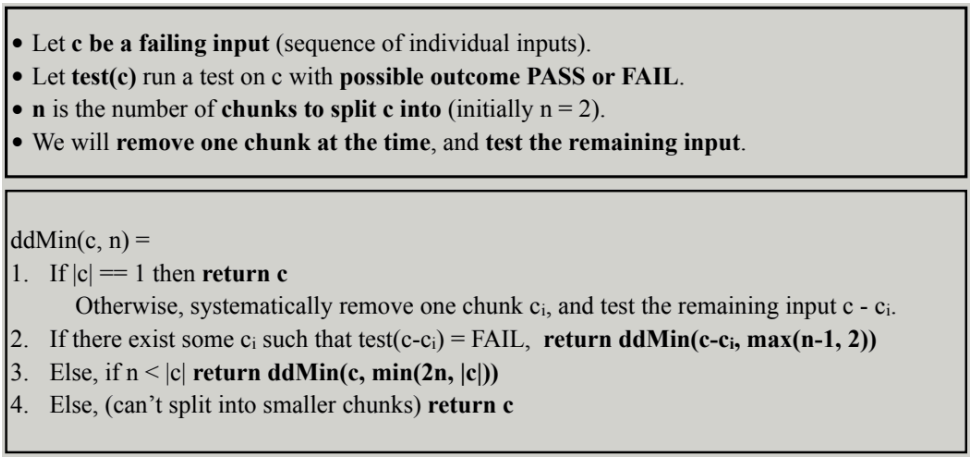

Figure 12: Delta Debugging

3.3.2. Short Quiz On Delta debugging

Question :

Suppose test(c) returns FAIL whenever ccontains two or more occurrences of the letter X. Apply the ddMin algorithm to minimise the failing input array [X, Z, Z, X] . Write down each step of the algorithm, and the values of n (number of chunks). Initially, n is 2.

Solution :

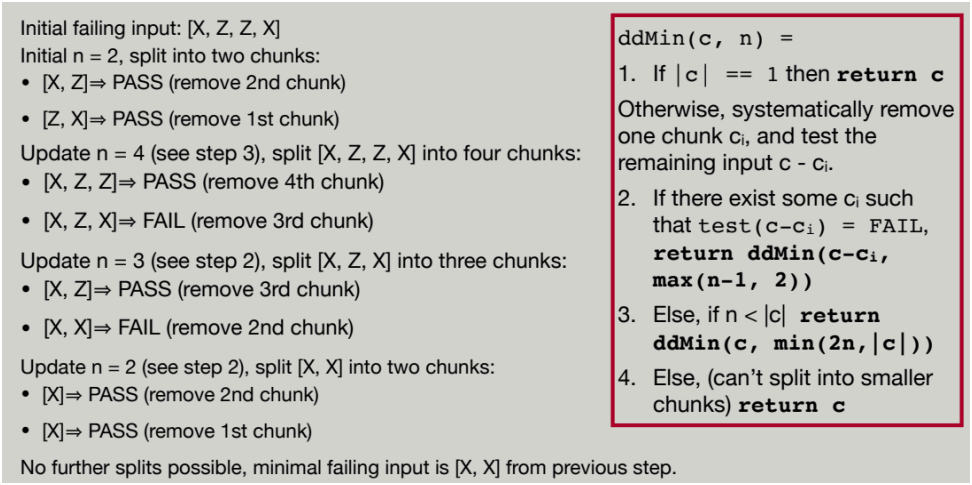

Figure 13: Delta Debugging Quiz Answer

3.4. Observing outcome, State Inspection

Mainly there are three diffrent ways to perform state inspections of a program ->

- **Simple logging :** print statements
- **Advanced loggin :** configureable what is printed based on level (e.g. OFF < FINE < INFO < WARNING < SEVERE), using e.g. Java's logging package.
- **Debugging tools :** e.g. Eclipse debugger or the Java debugger jbd (hand-in assignment 2)

3.4.1. The Quick-And-Dirty Approach : Print Logging

- **Manually add prit statements at code locations to be observed**
- System.out.println("size = "+ size);

Pros ->

- Simple and easy.
- No tools or infrastructure needed, works on any platform.

Cons ->
- Code cluttering.
- Output cluttering.
- Performance penalty, possibly changed behaviour (real time apps).
- Buffered output lost on crash
- Source code access required, recompilation necessary.

3.4.2. BASIC LOGGING IN JAVA

- Each class can have its own logger-object.
- Each logger has level:
- OFF < FINE… < INFO < WARNING < SEVERE
- Setting the level controls which messages gets written to log.
- Quick Demo: Dubbel.java

Pros ->
- Output cluttering can be mastered
- Small performance overhead
- Exceptions are loggable
- Log complete up to crash
- Etc.

Cons ->
- Code cluttering - don't try to log everything

3.4.3. Using Debuggers

Assume we have found a small failing test case and identified the faulty component.

Basic Functionality of a Debugger
- Execution Control: Stop execution at specific locations, breakpoints
- Interpretation: Step-wise execution of code
- State Inspection: Observe values of variables and stack
- State Change: Change state of stopped program
- Debugging tools: Eclipse GUI debugger or the Java debugger jbd

4. Formal Specification

Describing contracts of units (methods) in a mathematically precise (formal) language.

Motivation:
- Higher degree of precision
- Automation of program analysis
  - program verification
  - static checking
  - test case generation

4.1. Unit specification

Methods can be specified by referring to:
- result value
- initial values of formal parameters
- pre-state and post-state

4.2. Writing formal Specifications

When writing a formal specification one should list the required pre and post-conditions. How to find these and formalize them can be tricky.

Example

Consider the very informal specification of `enterPIN (pin:int)`:

"Enter the PIN that belongs to the currently inserted bank card into the ATM, when not yet authenticated. If a wrong PIN is entered three times in a row, the card is invalidated and confiscated. After having entered the correct PIN, the customer is regarded as authenticated."

Contract states *what is guaranteed* (postcondition), *under which conditions* (precondition).

Preconditions:

- Card is inserted, user not yet authenticated.

Postconditions:

- If PIN is correct, then the user is authenticated.
- If PIN is incorrect and `wrongPINCounter` was < 2 when entering the method, then `wrongPINCounter` is increased by 1 and user is not authenticated.
- If PIN is incorrect and `wrongPINCounter` was ≥ 2 when entering the method, then card is conficaded and user is not authenticated.

There are also some implicit ones:

Implicit preconditions:

- ATM card slot is free
- Card is valid
- Card is not null

Implicit postconditons:

- ATM card slot is occupied
- User is not authenticated
- `wrongPINCounter` is 0

Important

Implicit pre/postconditions should also be formalised, for example that the arguments of a method should not be null, as shown in the above example

4.3. Validity

In context of formal specification, we want to know if some **formula hold true in a particular program state.**

- A formula is **valid** if it is true in **all possible states**
- Valid formulas are **useful to simplify other formulas**
- **A formula is satisfiable if it is true at least once**

Example

The following **useful** formulas are valid, i.e. you can replace the formula on the left side by the one on the right.

1. $\neg(\exists x : t.\ P) \leftrightarrow \forall x : t.\ \neg P$
2. $\neg(\forall x : t.\ P) \leftrightarrow \exists x : t.\ \neg P$
3. $(\forall x : t.\ P \wedge Q) \leftrightarrow (\forall x : t.\ P) \wedge (\forall x : t.\ Q)$
4. $(\exists x : t.\ P \vee Q) \leftrightarrow (\exists x : t.\ P) \vee (\exists x : t.\ Q)$

1. ¬(P ∧ Q) ↔ ¬P ∨ ¬Q
2. ¬(P ∨ Q) ↔ ¬P ∧ ¬Q
3. (true ∧ P) ↔ P
4. (false ∨ P) ↔ P
5. true ∨ P
6. ¬(false ∧ P)
7. (P → Q) ↔ (¬P ∨ Q)
8. P → true
9. false → P
10. (true → P) ↔ P
11. (P → false) ↔ ¬P
12. ¬(∃x : t. P) ↔ ∀x : t. ¬P
13. ¬(∀x : t. P) ↔ ∃x : t. ¬P
14. (∀x : t. P ∧ Q) ↔ (∀x : t. P) ∧ (∀x : t .Q)
15. (∃x : t. P ∨ Q) ↔ (∃x : t. P) ∨ (∃x : t. Q)

Figure 14: Some other useful valid formulas

5. Dafny

Object oriented language desinged to make it easy to write **correct** code.

- Allow annotations specifying program behaviour, as part of the language.
- Automatically proves that the code adheres to specifications.
- Absence of run-time errors, e.g. null-pointers, index-out-of-bounds etc.
- Termination checking of loops.

5.1. Fields

Declaring fields.

- Variables declared with keyword `var`
- Type annotations given by `:`
- Assignment written `:=`
- Several variables can be declared at once.
- Parallel assignments possible.

```
1  var insertedCard : BankCard?;
2  var wrongPINCounter : int;
3  var customerAuthenticated : bool;
```

```
1  var x : int;
2  x := 34;
3  var y, z := true, false //parallel assignment
```

Note

The prefix `?` clarifies that the field is allowed to be `null`.

5.2. Constructor

Example of a constructor.

```
constructor (n: int)
  modifies this
{
  ...
}
```

5.3. Methods

Example of some methods

```
method insertCard (card : BankCard){ ... }
method enterPIN (pin : int) { ... }
method add (num1 : int, num2 : int) returns (result : int) { ... }
```

5.4. Propositional logic

All variables P, Q, R (aka propositions) are booleans, i.e. take values True or False.

Connectiv	Meaning	Dafny
$\neg P$	not P	`!P`
$P \wedge Q$	P and Q	`P && Q`
$P \vee Q$	P or Q	`P || Q`
$P \rightarrow Q$	P implies Q	`P ==> Q`
$P \leftrightarrow Q$	P is equivalent to Q	`P <==> Q`

5.5. First order logic: Quantifiers

Writing quantifiers in Dafny.

Connectiv	Meaning	Dafny
$\forall x : t.\ P$	For all x of type t, P holds	`forall x : t :: P`
$\exists x : t.\ P$	There exists an x of type t, such that P holds	`exists x : t :: P`

Example

The array `a` only holds values less than or equal to 2

```
forall i : int :: 0 <= i < a.Length ==> arr[i] <= 2;
```

At least one entry holds the value 1

```
exists i : int :: 0 <= i < a.Length && a[i] == 1
```

5.6. Requires & Ensures

The `requires` and `ensures` prefixes represents pre/postconditions which can consist of:

- quantifiers, logic connectives
- functions (`function fibonacci`, etc.)
- predicates

They have no impact on runtime and are just for checking the validity of the program.

```
method example(x : int, y : int) returns (m : int)
  requires x >= 0 && y < = // precondition
  ensures m <= 0 // postcondition
{ return y*x }
```

5.7. Classes

- Keyword `class`
- No access modifiers like public, private, etc.
- Fields declared by `var` keyword (local variables)
- Objects declared with `new` keyword + `constructor` methods
  - Can have one anonymous (unamed constructor) + several named ones.

```
class MyClass {
  var field : int;

  constructor() {
    field := 0;
  }

  constructor Init(x : int) {
    field := x;
  }

  constructor Init2(x : int, y :int) {
    field := x + y;
  }
}
```

```
var myObject  := new MyClass();
var myObject2 := new MyClass.Init(5);
var myObject3 := new MyClass.Init2(5, 6)
```

5.8. Assertions

- Keyword `assert`
- Placed in method body
- Written in specification language
- Evaluated at compile-time

Dafny tries to prove **assertion hold for all executions of code**.

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r {
  if (x < 0) {r := -x;}
  else {r := x;}
}

method Test() {
  var v := Abs(-3)
  assert 0 <= v;
}
```

Note

Assertions can only be proved based on the annotions (`requires` , `ensures`) of other methods. The previous method `test` works because the assertion is based on the `ensures` of the method `Abs` .

The following test case would not work:

```
method Test() {
  var v := Abs(-3)
  assert v == 3;
}
```

This is because Dafny only knows that `Abs` returns a value greater than or equal to 0. It does not know that it returns 3.

However, if we edit the postcondition to the following:

```
ensures 0 <= x ==> r == x; // result same as x for positive input x
ensures x < 0 ==> r == -x; // result positive x for negative input x
```

Then the test case would work. Dafny now knows that if x is negative, the result will be $-x$ (positive x).

5.9. (Ghost) Functions

Part of the specification language, thus functions cannot modify objects and write to memory (unlike methods). *so it is SAFE to use in spec.*

- Can only be used in spec (annotations)
- Single unnamed return value
- body is a single statement (no semicolon)

```
ghost function abs (x : int) : int {
  if x < 0 then -x else x
}
```

Now we can use `abs` in our specification.

```
method Abs(x : int) returns (r : int)
  ensures r == abs(x) {
  if (x < 0) {r := -x;}
  else {r := x;}
}
```

5.10. Function methods

A function method can be used in both spec and execution. If we consider the previus method `Abs`, it might not even be necessary.

```
function abs (x : int) : int {
  if x < 0 then -x else x
}

method Test() {
  var v := abs(-3)
  assert v == 3;
}
```

Note

Dafny remembers function method bodies, unlike methods.

5.11. Predicates

Recall, a predicate in first-order logic is a function returning a boolean value. In Dafny, predicates are used similarly to functions. They can be used both in spec and execution.

```
ghost predicate isEven (x : int) { x % 2 == 0}
```

is equivalent to

```
ghost function isEven (x : int) : bool { x % 2 == 0 }
```

and

```
predicate isEven (x : int) { x % 2 == 0}
```

is equivalent to

```
function isEven (x : int) : bool { x % 2 == 0 }
```

5.12. Modifies & Reads clauses

Automated proofs are hard and can be slow. In Dafny, this means that we need to add certain annotations to help.

- Dafny methods manipulating objects must declare what fields they might modify.
- Dafny functions/predicates must declare what memory locations (objects) they might read.

```
class BankCard {
  var pin : int;
  var accNo : int;
  var valid : bool;

  ...

  predicate isValid()
    reads this`valid; // read clause
    { this.valid }

  method invalidateCard()
    modifies this`valid; // modifies clause
    ensures !isValid();
    { valid := false;}

  ...
}
```

Note

back-tick character is used to refer to fields of the object (class) such as int, bool, etc. In the case of objects inside the class this is not needed.

5.13. Old keyword

old-keyword in specification refers to the **value of a field before the method was called**.

Example

`old(wrongPINCounter)` refers to its value before `insertPIN` method was called.

5.14. Arrays

Declaring and initialising an array

```
var a : array<int>;
a := new int[3];
assert a.Length == 3;
a[0], a[1], a[2] := 0, 0, 0;
```

Declaring a matrix

```
var matrix : array2<int>;
matrix := new int[3, 4];
assert matrix.Length0 == 3 && matrix.Length1 == 4;
```

Parallel assignment: set all entries to 0

```
forall(i | 0 <= i < a.Length)
{a[i] := 0;}
forall (i,j | 0 <= i < m.Length0 && 0 <= j < m.Length1)
{m[i,j] := 0; }
```

Parallel assignment: increment all entries by 1. Note that all right-hand side expressions is evaluated before assignments.

```
forall(i | 0 <= i < a.Length)
{a[i] := a[i] + 1;}
```