Foundations of Software Testing 2E

ADITYA P. MATHUR

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 21, 2013

Contents

Chapter 1:

Preliminaries: Software Testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 17, 2013

Contents

Learning Objectives

- Errors, Testing, debugging, test process, CFG, correctness, reliability, oracles.
- Finite state machines
- Testing techniques

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

1.1 Humans, errors and testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Errors

Errors are a part of our daily life.

Humans make errors in their thoughts, actions, and in the products that might result from their

actions.

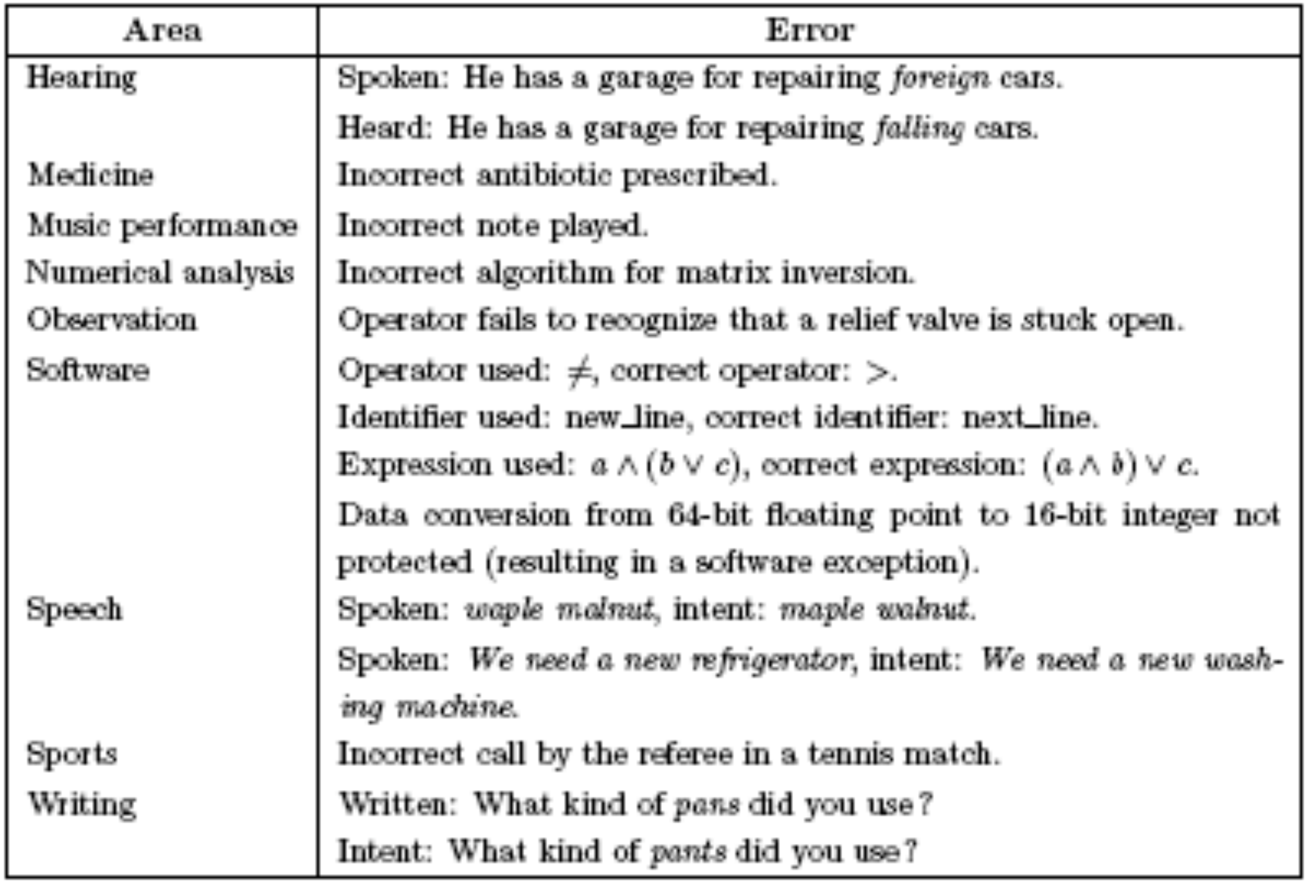
Errors occur wherever humans are involved in taking actions and making decisions.

These fundamental facts of human existence make testing an essential activity.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Errors: Examples

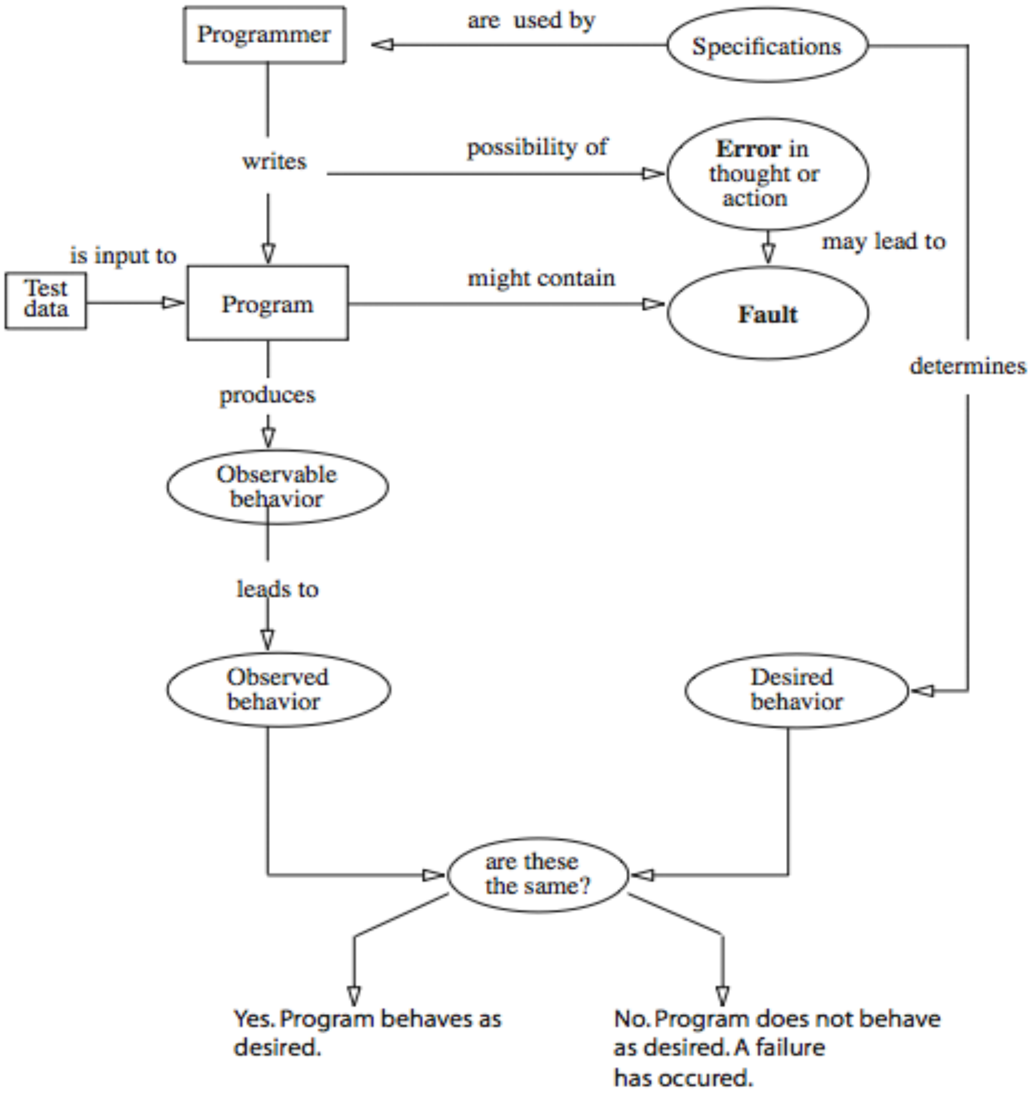


Area	Error
Hearing	Spoken: He has a garage for repairing *foreign* cars. Heard: He has a garage for repairing *falling* cars.
Medicine	Incorrect antibiotic prescribed.
Music performance	Incorrect note played.
Numerical analysis	Incorrect algorithm for matrix inversion.
Observation	Operator fails to recognize that a relief valve is stuck open.
Software	Operator used: $\neq$, correct operator: $>$. Identifier used: new_line, correct identifier: next_line. Expression used: $a \wedge (b \vee c)$, correct expression: $(a \wedge b) \vee c$. Data conversion from 64-bit floating point to 16-bit integer not protected (resulting in a software exception).
Speech	Spoken: *waple malnut*, intent: *maple walnut*. Spoken: *We need a new refrigerator*, intent: *We need a new washing machine*.
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of *pans* did you use? Intent: What kind of *pants* did you use?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Error, faults, failures



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

1.2 Software Quality

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Software quality

Static quality attributes: structured, maintainable, testable code as well as the availability of correct and complete documentation.

Dynamic quality attributes: software reliability, correctness, completeness, consistency, usability, and performance

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Software quality (contd.)

Completeness refers to the availability of all features listed in the requirements, or in the user manual. An incomplete software is one that does not fully implement all features required.

Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Software quality (contd.)

Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing.

Performance refers to the time the application takes to perform a requested task. It is considered as a *non-functional requirement*. It is specified in terms such as ``This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory."

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

1.3 Requirements, behavior, and correctness

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Requirements, behavior, correctness

Requirements leading to two different programs:

Requirement 1: It is required to write a

program that inputs two integers and outputs the maximum of these.

Requirement 2: It is required to write a

program that inputs a sequence of integers and outputs the sorted version of this sequence.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Requirements: Incompleteness

Suppose that program **max** is developed to satisfy Requirement 1. The expected output of **max** when the input integers are 13 and 19 can be easily determined to be 19.

Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Requirements: Ambiguity

Requirement 2 is ambiguous. It is not clear whether the input sequence is to sorted in ascending or in descending order. The behavior of **sort** program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing **sort**.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Input domain (Input space)

*The set of all possible inputs to a program **P** is known as the input domain or input space, of **P**.*

Using Requirement 1 above we find the input domain of **max**

to be the set of all pairs of integers where each element in the pair integers is in the range -32,768 till 32,767.

Using Requirement 2 it is not possible to find the input domain for the sort program.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Input domain (Continued)

Modified Requirement 2:

It is required to write a program that inputs a

sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request character which should be ``A'' when an ascending sequence is desired, and ``D'' otherwise.

While providing input to the program, the request character is input first followed by the sequence of integers to be sorted; the sequence is terminated with a period.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Input domain (Continued)

Based on the above modified requirement, the input domain for **sort** is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Valid/Invalid Inputs

The modified requirement for sort mentions that the

request characters can be ``A'' and ``D'', but fails to answer the question ``What if the user types a different character ?’’

When using **sort** it is certainly possible for the user to type a

character other than ``A'' and ``D''. Any character other than ``A'’ and ``D'' is

considered as invalid input to **sort**. The requirement for **sort** does not specify

what action it should take when an invalid input is encountered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

1.4 Correctness versus reliability

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Correctness

Though correctness of a program is desirable, it is almost never the objective of testing.

To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish.

Thus, correctness is established via mathematical proofs of programs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Correctness and Testing

While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it.

Thus, completeness of testing does not necessarily demonstrate that a program is error free.

Testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Software reliability: two definitions

Software reliability [ANSI/IEEE Std 729-1983]: is the probability of failure free operation of software over a given time interval and under given conditions.

Software reliability is the probability of failure free operation of software in its intended environment.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Operational profile

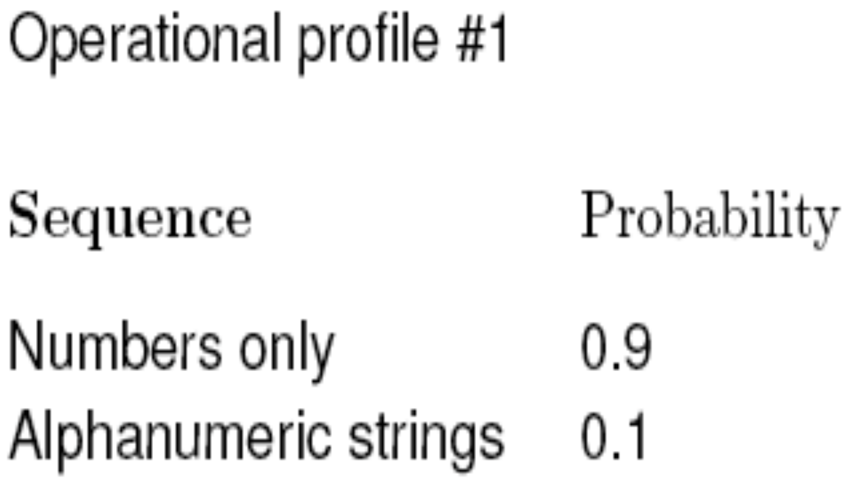
An operational profile is a numerical description of how a program is used.

Consider a sort program which, on any given execution, allows any one of two types of input sequences. Sample operational profiles for sort follow.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Operational profile



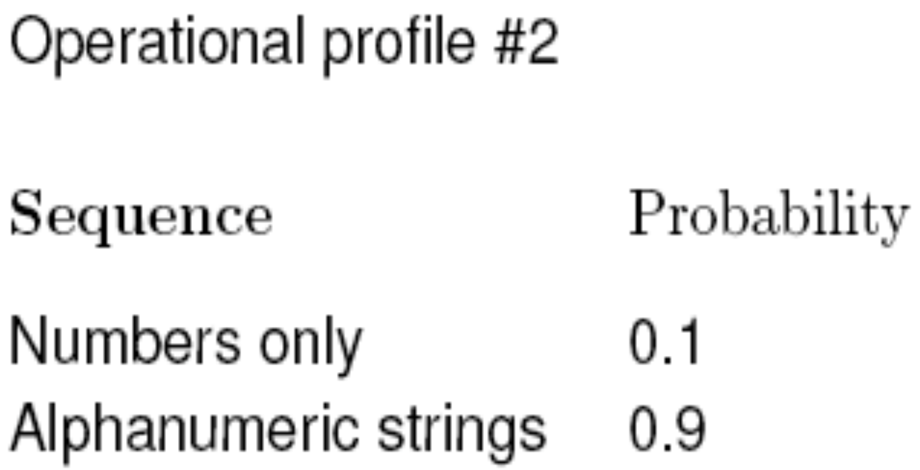
Operational profile #1

Sequence	Probability
Numbers only	0.9
Alphanumeric strings	0.1

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Operational profile



Operational profile #2

Sequence	Probability
Numbers only	0.1
Alphanumeric strings	0.9

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

1.5 Testing and debugging

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

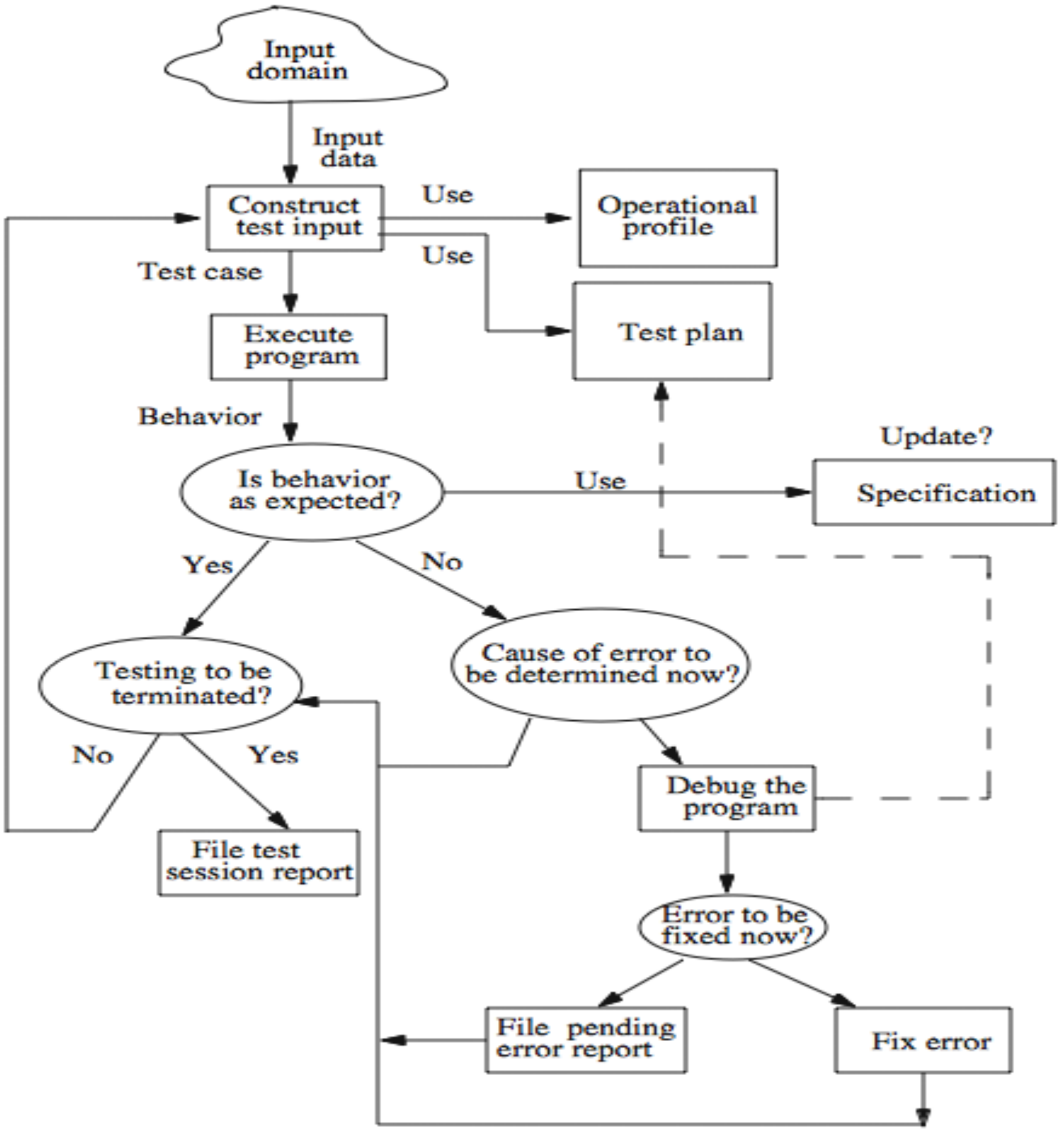
Testing and debugging

Testing is the process of determining if a program has any errors.

When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as debugging.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

A test/debug cycle



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test plan

A test cycle is often guided by a test plan.

Example: The sort program is to be tested to meet the requirements given earlier. Specifically, the following needs to be done.

- Execute sort on at least two input sequences, one with ``A'' and the other with ``D'' as request characters.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test plan (contd.)

- Execute the program on an empty input sequence.
- Test the program for robustness against erroneous inputs such as ``R" typed in as the request character.
- All failures of the test program should be recorded in a suitable file using the Company Failure Report Form.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test case/data

A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A test set is a collection of zero or more test cases.

Sample test case for sort:

Test data: <"A'’ 12 -29 32 >

Expected output: -29 12 32

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Program behavior

Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.

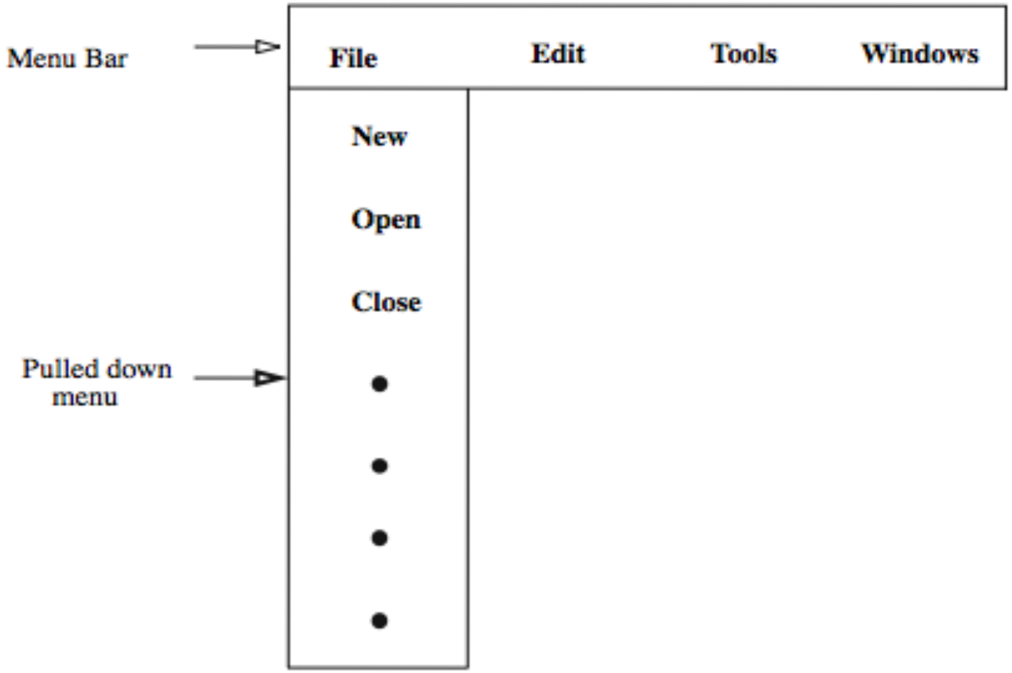
A state diagram specifies program states and how the program changes its state on an input sequence. inputs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Program behavior: Example

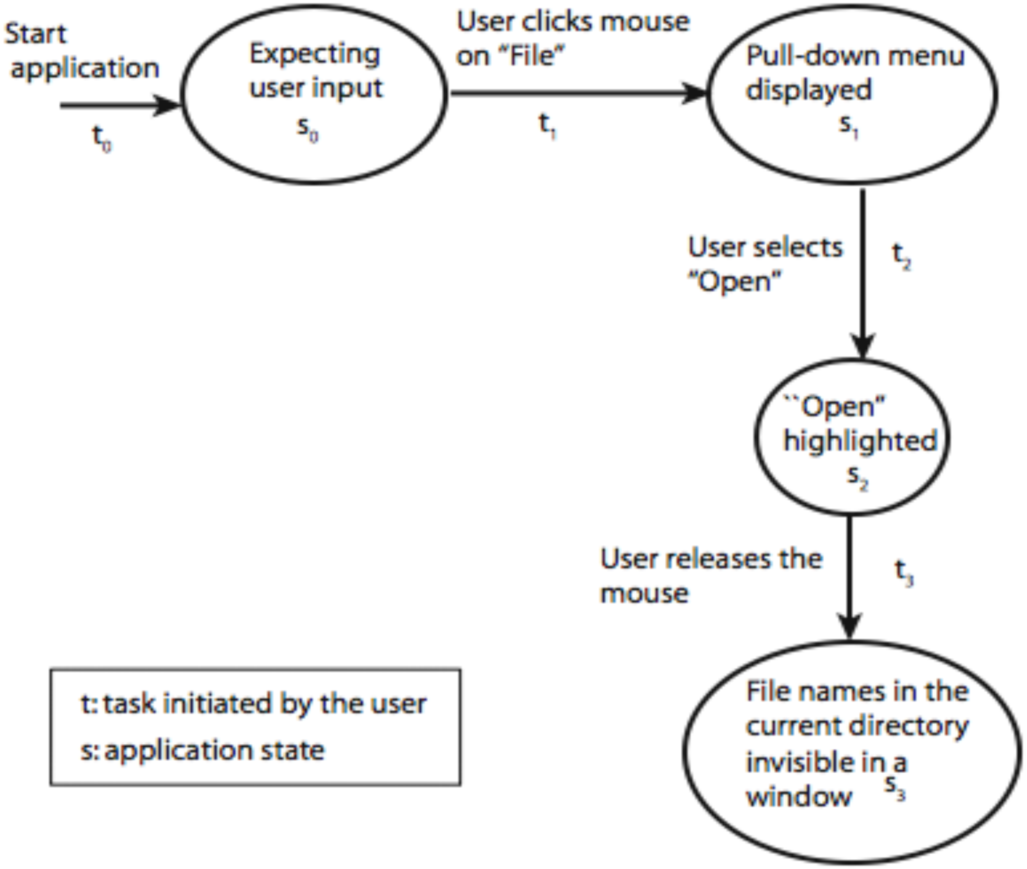
Consider a menu driven application.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Program behavior: Example (contd.)



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Behavior: observation and analysis

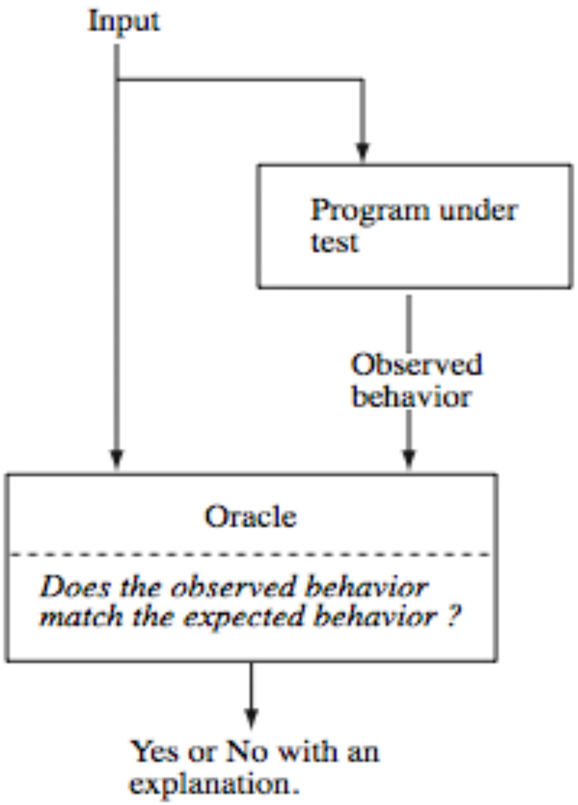
In the first step one observes the behavior.

In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Oracle: Example


Input
Program under test
Observed behavior
Oracle
Does the observed behavior match the expected behavior ?
Yes or No with an explanation.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Oracle: Programs

Oracles can also be programs designed to check the behavior of other programs.

For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

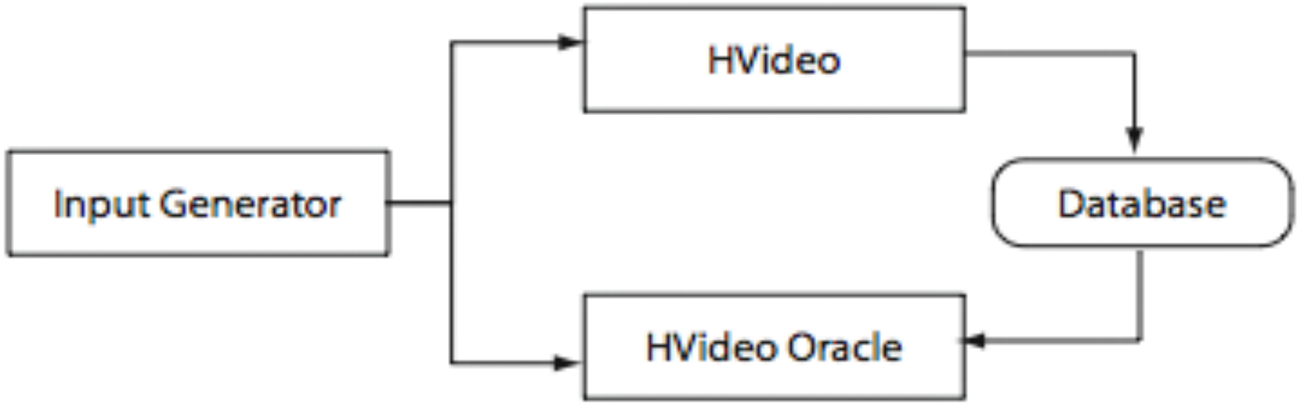
Oracle: Construction

Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires the determination of input-output relationship.

In general, the construction of automated oracles is a complex undertaking.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Oracle construction: Example



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Testing and verification

Program verification aims at proving the correctness of programs by showing that it contains no errors. This is very different from testing that aims at uncovering errors in a program.

Program verification and testing are best considered as complementary techniques. In practice, program verification is often avoided, and the focus is on testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Testing and verification (contd.)

Testing is not a perfect technique in that a program might contain errors despite the success of a set of tests.

Verification promises to verify that a program is free from errors. However, the person/tool who verified a program might have made a mistake in the verification process; there might be an incorrect assumption on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on.

Verified and published programs have been shown to be incorrect.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

1.10. Test generation strategies

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation

Any form of test generation uses a source document. In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements.

In several commercial environments, the process is a bit more formal. The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source. In more advanced test processes, requirements serve as a source for the development of formal models.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test generation strategies

Model based: require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri net, etc.

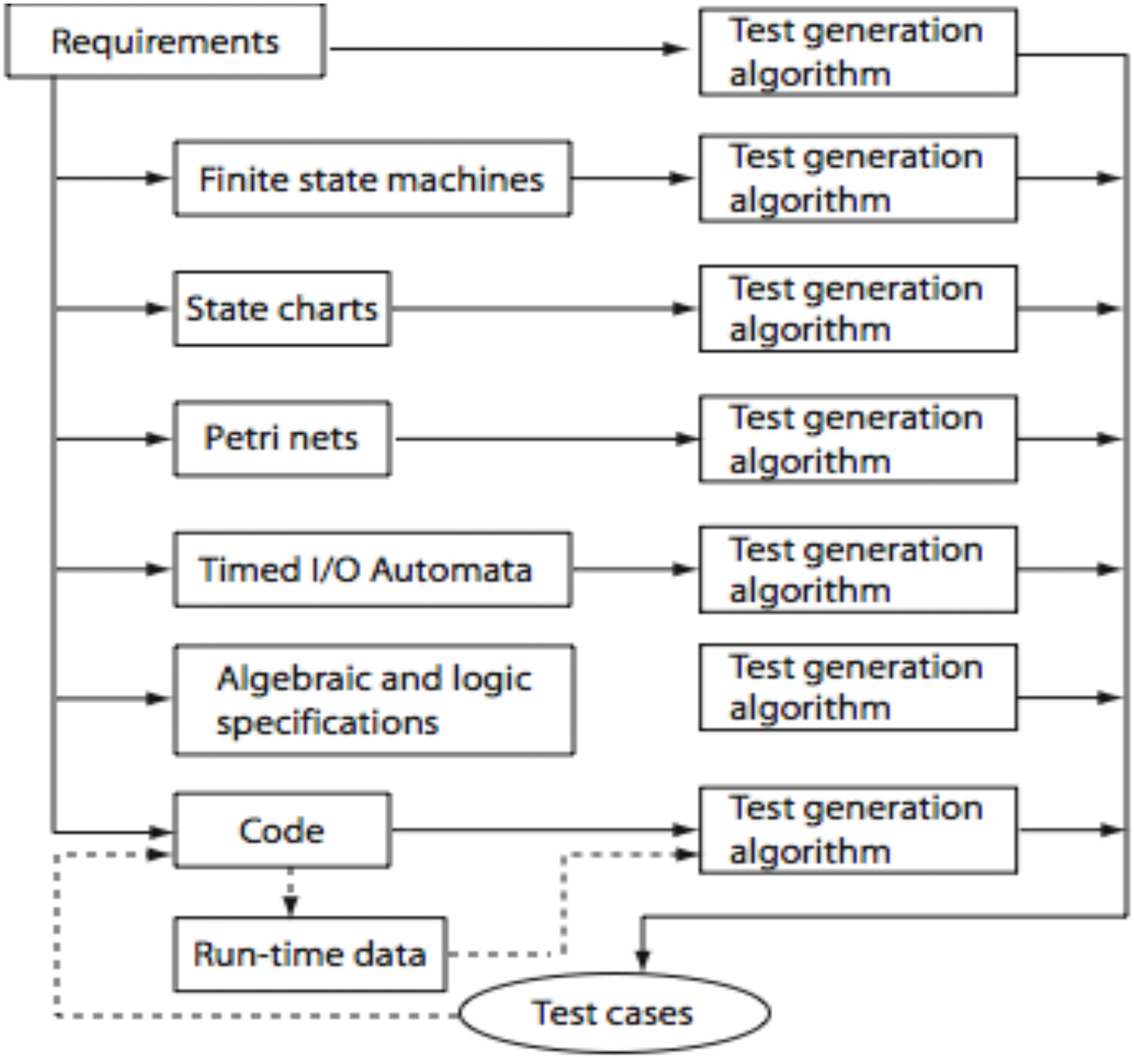
Specification based: require that a subset of the requirements be modeled using a formal mathematical notation. Examples: B, Z, and Larch.

Code based: generate tests directly from the code.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation strategies (Summary)


Requirements
Finite state machines
State charts
Petri nets
Timed I/O Automata
Algebraic and logic specifications
Code
Run-time data
Test generation algorithm
Test generation algorithm
Test generation algorithm
Test generation algorithm
Test generation algorithm
Test generation algorithm
Test generation algorithm
Test cases

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

1.13 Types of software testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Types of testing

One possible classification is based on the following four classifiers:

C1: Source of test generation.

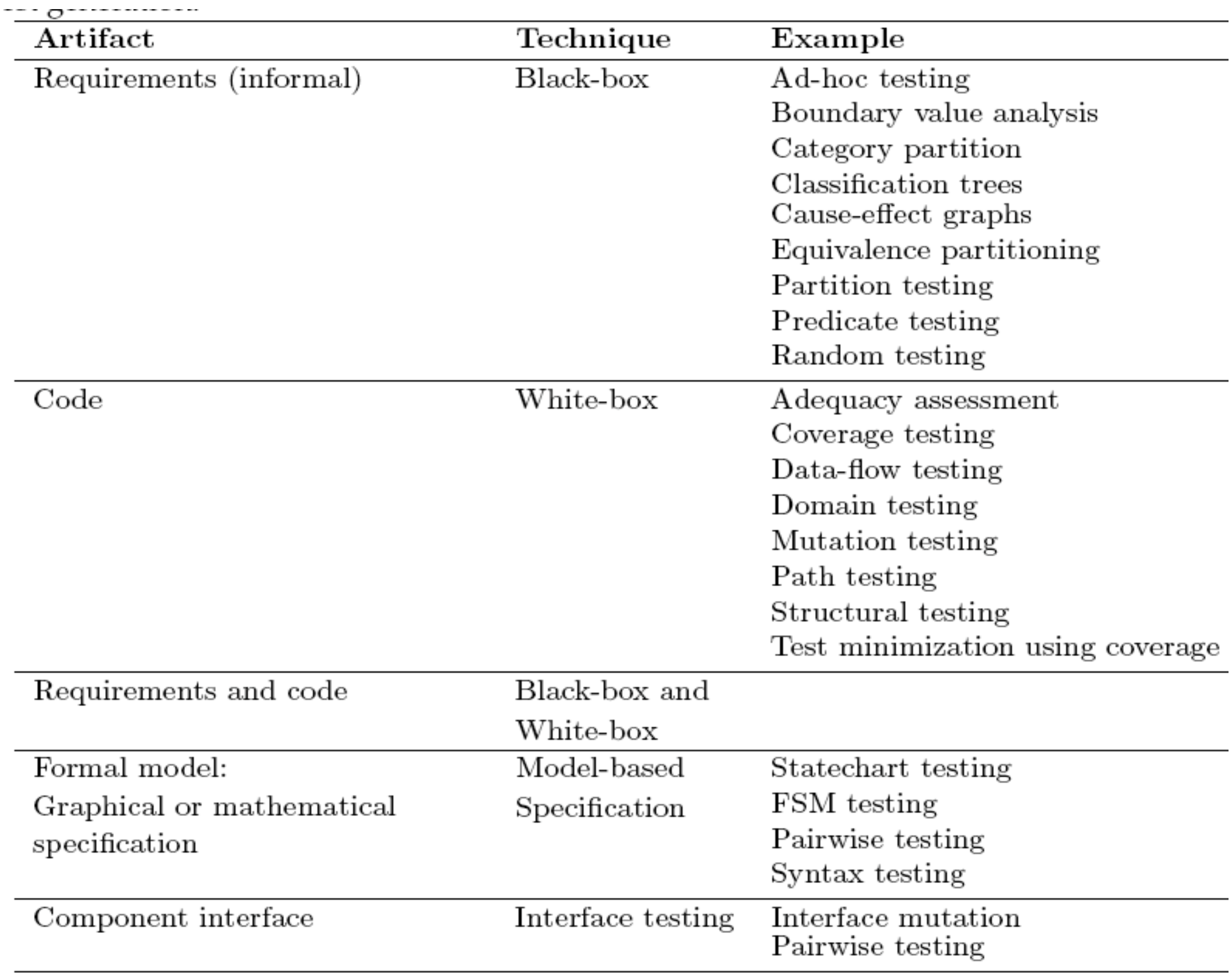
C2: Lifecycle phase in which testing takes place

C3: Goal of a specific testing activity

C4: Characteristics of the artifact under test

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

C1: Source of test generation

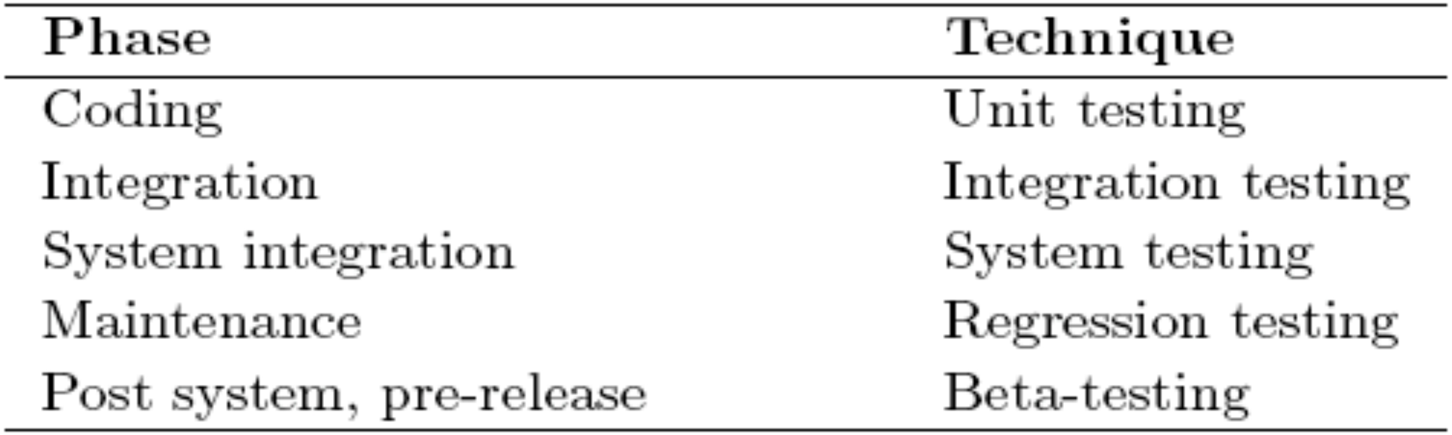


Artifact	Technique	Example
Requirements (informal)	Black-box	Ad-hoc testing Boundary value analysis Category partition Classification trees Cause-effect graphs Equivalence partitioning Partition testing Predicate testing Random testing
Code	White-box	Adequacy assessment Coverage testing Data-flow testing Domain testing Mutation testing Path testing Structural testing Test minimization using coverage
Requirements and code	Black-box and White-box	
Formal model: Graphical or mathematical specification	Model-based Specification	Statechart testing FSM testing Pairwise testing Syntax testing
Component interface	Interface testing	Interface mutation Pairwise testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C2: Lifecycle phase

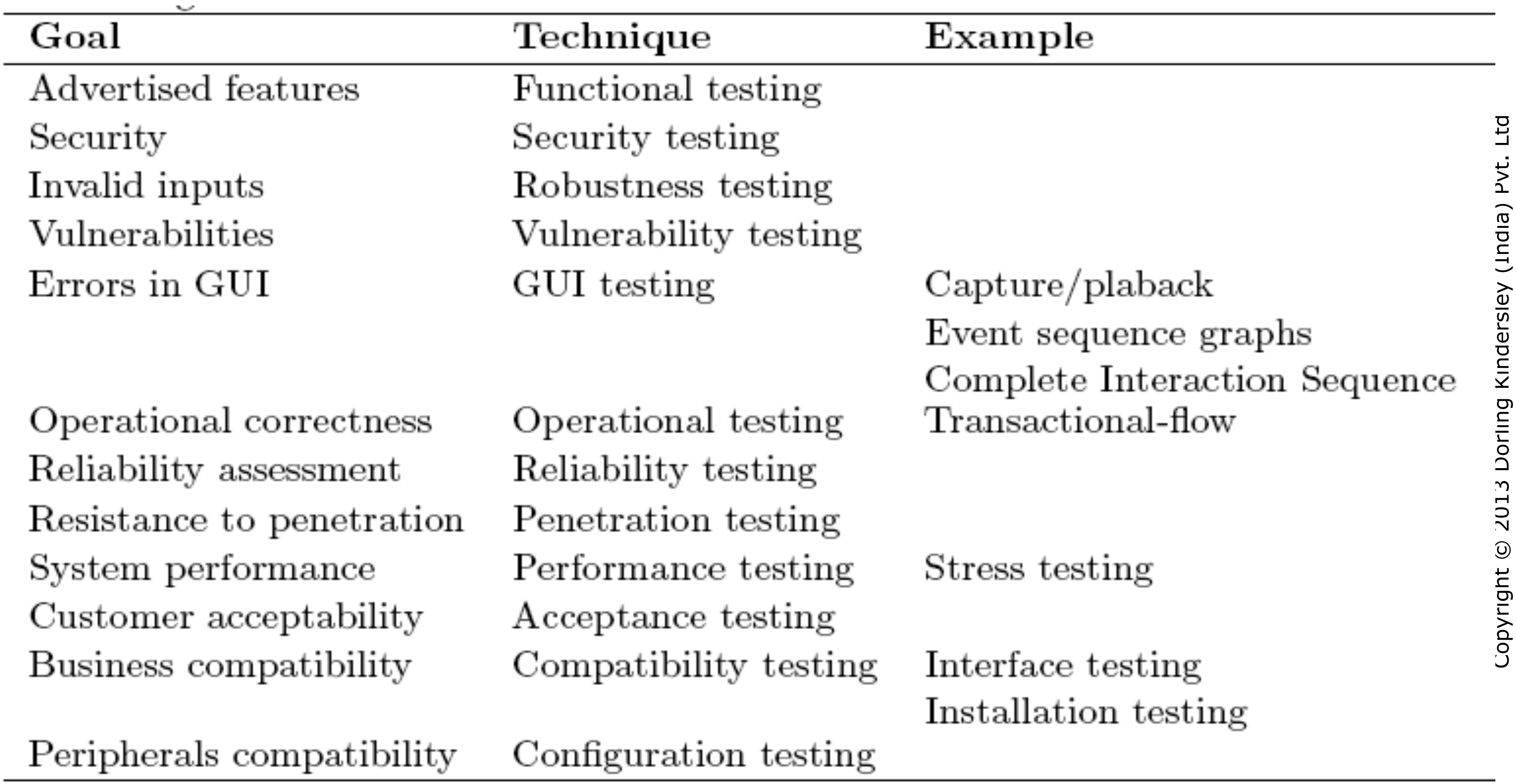


Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Post system, pre-release	Beta-testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C3: Goal of specific testing activity

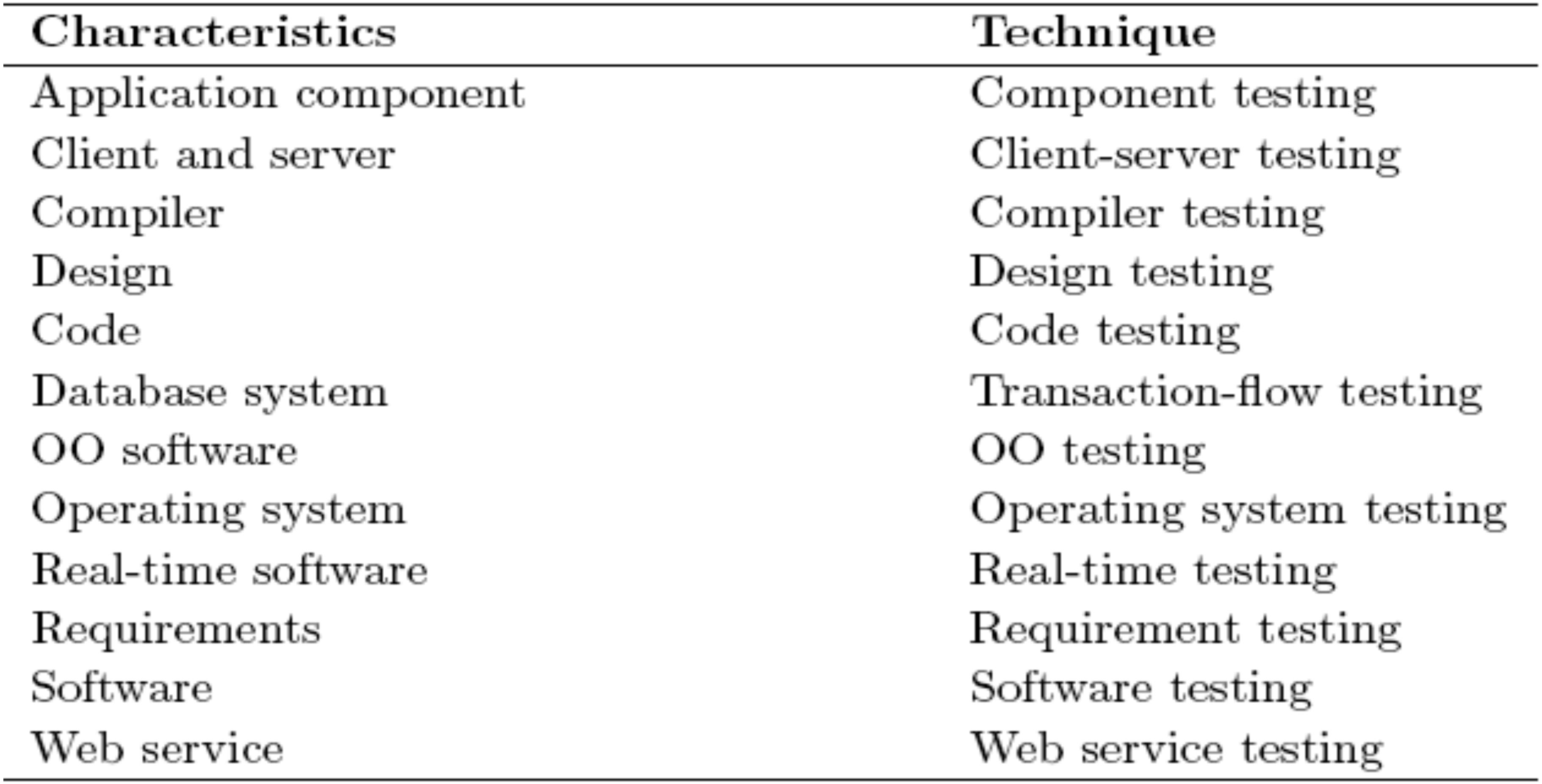


Goal	Technique	Example
Advertised features	Functional testing	
Security	Security testing	
Invalid inputs	Robustness testing	
Vulnerabilities	Vulnerability testing	
Errors in GUI	GUI testing	Capture/plaback Event sequence graphs Complete Interaction Sequence
Operational correctness	Operational testing	Transactional-flow
Reliability assessment	Reliability testing	
Resistance to penetration	Penetration testing	
System performance	Performance testing	Stress testing
Customer acceptability	Acceptance testing	
Business compatibility	Compatibility testing	Interface testing Installation testing
Peripherals compatibility	Configuration testing	

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C4: Artifact under test



Characteristics	Technique
Application component	Component testing
Client and server	Client-server testing
Compiler	Compiler testing
Design	Design testing
Code	Code testing
Database system	Transaction-flow testing
OO software	OO testing
Operating system	Operating system testing
Real-time software	Real-time testing
Requirements	Requirement testing
Software	Software testing
Web service	Web service testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary

We have dealt with some of the most basic concepts in software testing. Exercises at the end of Chapter 1 are to help you sharpen your understanding.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 2:

Preliminaries: Mathematical

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 12, 2013

2.1 Predicates and Boolean expressions

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Where do predicates arise?

Predicates arise from requirements in a variety of applications. Here is an example from Paradkar, Tai, and Vouk, "Specification based testing using cause-effect graphs, Annals of Software Engineering," V 4, pp 133-157, 1997.

A boiler needs to be to be shut down when the following conditions hold:

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Boiler shutdown conditions

1. The water level in the boiler is below X lbs. (a)
2. The water level in the boiler is above Y lbs. (b)
3. A water pump has failed. (c)
4. A pump monitor has failed. (d)
5. Steam meter has failed. (e)

Boiler in degraded mode when either is true.

The boiler is to be shut down when a or b is true or the boiler is in degraded mode and the steam meter fails. We combine these five conditions to form a compound condition (predicate) for boiler shutdown.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boiler shutdown conditions

Denoting the five conditions above as a through e, we obtain the following Boolean expression E that when true must force a boiler shutdown:

$$E=a+b+(c+d)e$$

where the + sign indicates "OR" and a multiplication indicates "AND."

The goal of predicate-based test generation is to generate tests from a predicate p that guarantee the detection of any error that belongs to a class of errors in the coding of p.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Another example

A condition is represented formally as a predicate, also known as a Boolean expression. For example, consider the requirement

``*if the printer is ON and has paper then send document to printer*."

This statement consists of a condition part and an action part. The following predicate represents the condition part of the statement.

p_r: (printer_status=ON) ∧ (printer_tray!= empty)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation from predicates

We will now examine two techniques, named BOR and BRO for generating tests that are guaranteed to detect certain faults in the coding of conditions. The conditions from which tests are generated might arise from requirements or might be embedded in the program to be tested.

Conditions guard actions. For example,

if condition then action

is a typical format of many functional requirements.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Predicates

Relational operators (relop): {<, ≤, >, ≥, =, ≠.}

= and == are equivalent.

Boolean operators (bop): {!,∧,∨, xor} also known as

{not, AND, OR, XOR}.

Relational expression: e1 relop e2. (e.g. a+b<c)

e1 and e2 are expressions whose values can be compared using relop.

Simple predicate: A Boolean variable or a relational

expression. (x<0)

Compound predicate: Join one or more simple predicates

using bop. (gender==“female”∧age>65)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Boolean expressions

Boolean expression: one or more Boolean variables joined by bop.

(a∧b∨!c)

a, b, and c are also known as literals. Negation is also denoted by placing a bar over a Boolean expression such as in

$$\overline{(a \wedge b)}$$

We also write ab for a∧b and a+b for a∨b when there is no confusion.

Singular Boolean expression: When each literal appears

only once, e.g., in (a∧b∨!c)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Boolean expressions (contd.)

Disjunctive normal form (DNF): Sum of product terms:

e.g. (p q) +(rs) + (a c).

Conjunctive normal form (CNF): Product of sums:

e.g.: (p+q)(r+s)(a+c)

Any Boolean expression in DNF can be converted to an equivalent CNF and vice versa.

e.g., CNF: (p+!r)(p+s)(q+!r)(q+s) is equivalent to DNF: (pq+!rs)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Boolean expressions (contd.)

Mutually singular: Boolean expressions e1 and e2 are mutually singular when they do not share any literal.

If expression E contains components e_1, e_2,.. then e_i is considered singular only if it is non-singular and mutually singular with the remaining elements of E.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

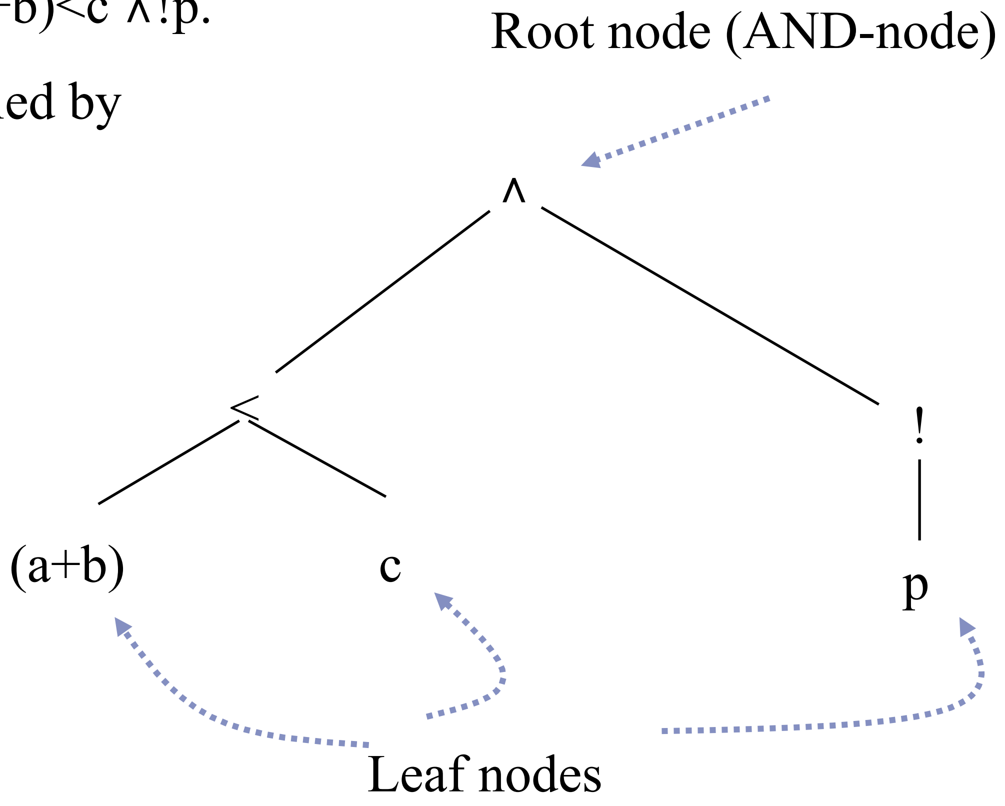
Boolean expressions: Syntax tree representation

Abstract syntax tree (AST) for: (a+b)<c ∧!p.

Notice that internal nodes are labeled by

Boolean and relational operators

Root node: OR-node is labeled as ∨.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

2.2 Program representation: Control flow graphs

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Program representation: Basic blocks

A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus, a block has unique entry and exit points.

Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Basic blocks: Example

Example: Computing x raised to y

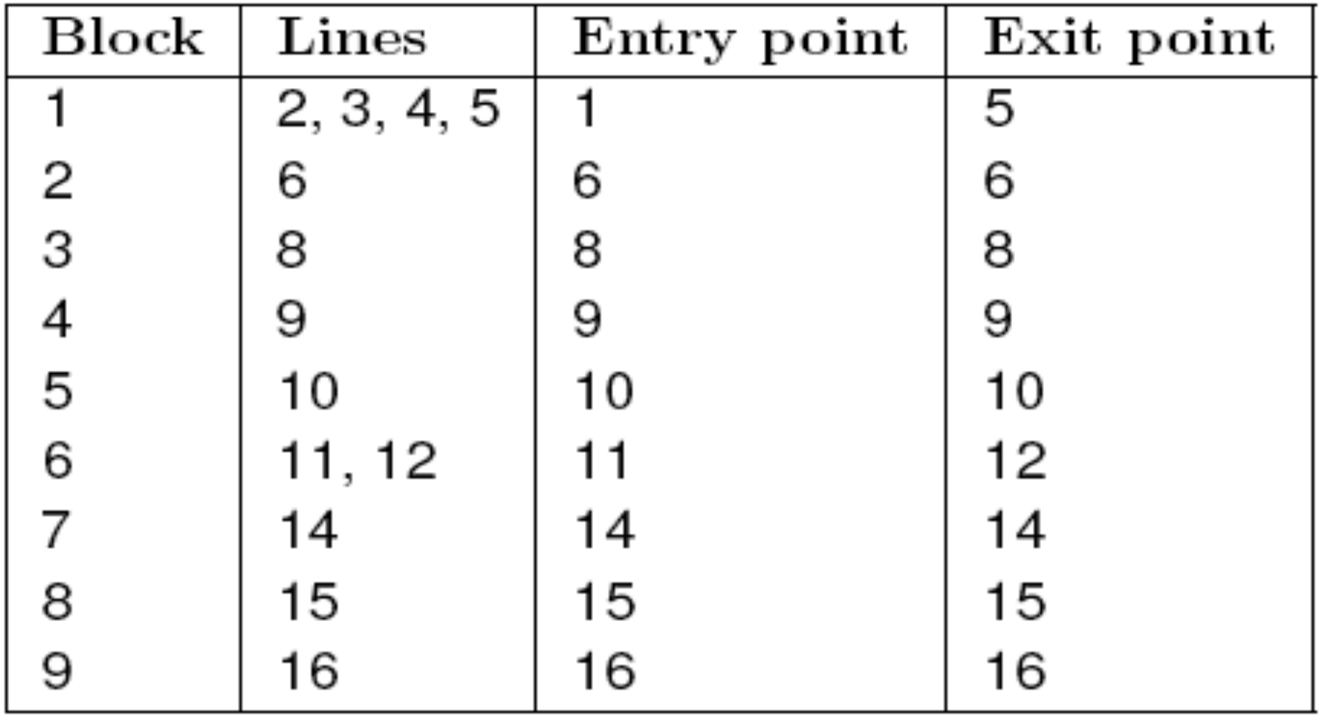
```
begin
  int x, y, power;
  float z;
  input (x, y);
  if (y<0)
    power=-y;
  else
    power=y;
  z=1;
  while (power! =0){
    z=z*x;
    power=power−1;
  }
  if (y<0)
    z=1/z;
  output(z);
end
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Basic blocks: Example (contd.)

Basic blocks



Block	Lines	Entry point	Exit point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Control Flow Graph (CFG)

A control flow graph (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N. We often write G= (N, E) to denote a flow graph G with nodes given by N and edges by E.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Control Flow Graph (CFG)

In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

Blocks and nodes are labeled such that block b_i corresponds to node n_i. An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j.

We also assume that there is a node labeled Start in N that has no incoming edge, and another node labeled End, also in N, that has no outgoing edge.

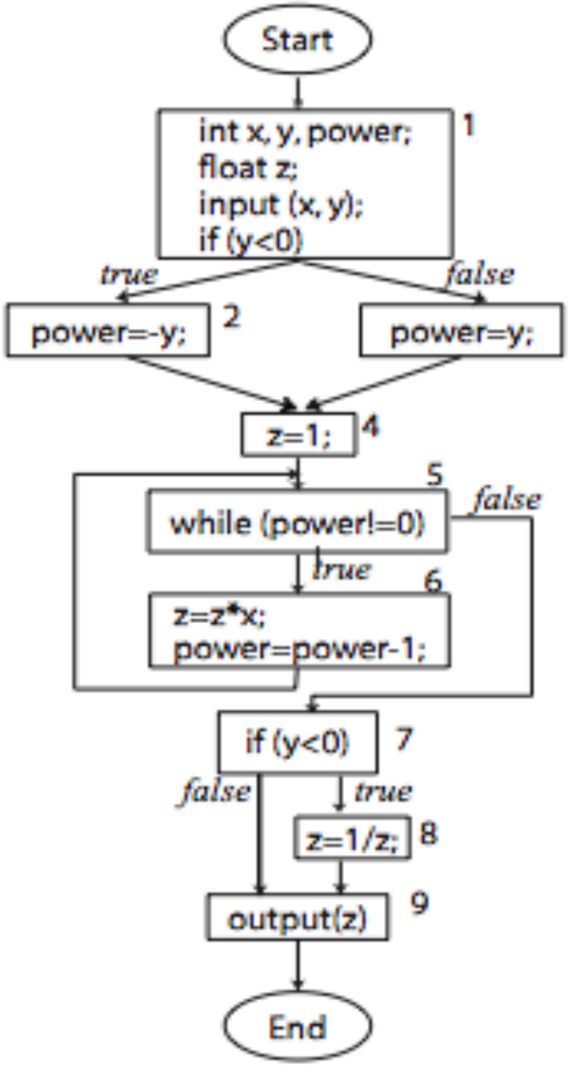
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

CFG Example

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

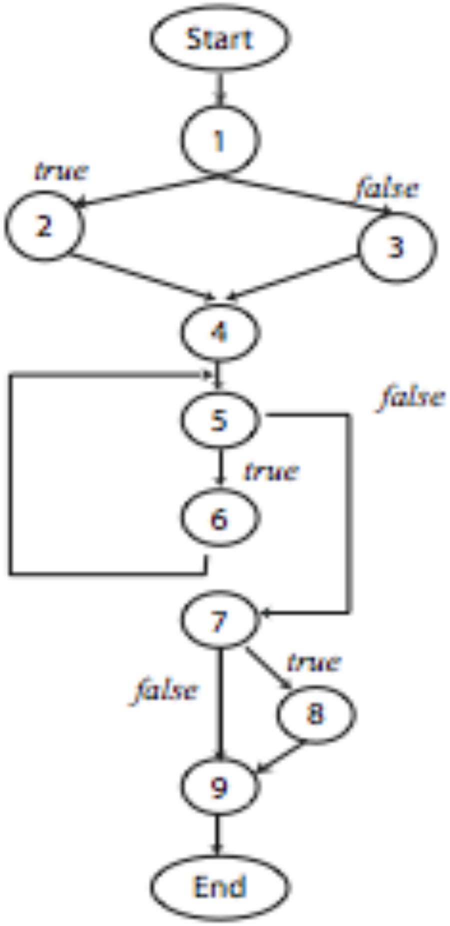
Contents

CFG Example

Same CFG with statements removed.

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Paths

Consider a flow graph G= (N, E). A sequence of k edges, k>0, (e_1, e_2, … e_k) , denotes a path of length k through the flow graph if the following sequence condition holds.

Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and 0< i<k, if e_i = (n_p, n_q) and e_{i+1} = (n_r, n_s) then $n_q = n_r$. }

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Paths: sample paths through the exponentiation flow graph

Two feasible and complete paths:

p_1= (Start, 1, 2, 4, 5, 6, 5, 7, 9, End)

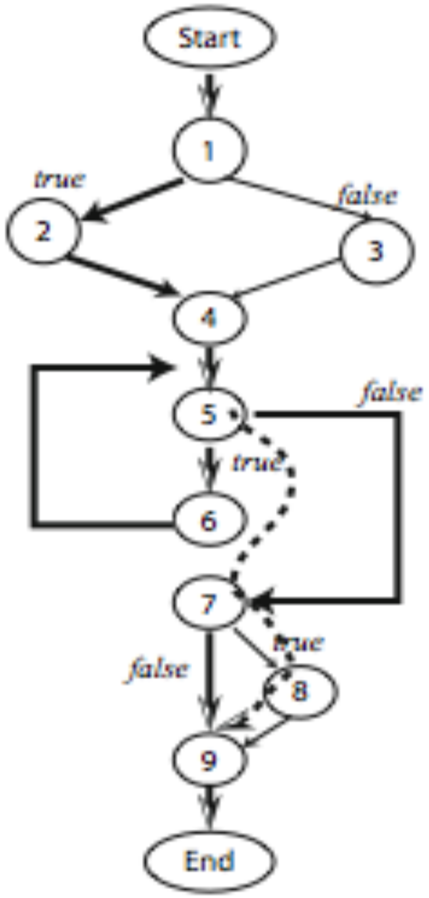
p_2= (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)

Specified unambiguously using edges:

p_1= ((Start, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, End))

Bold edges: complete path.

Dashed edges: subpath.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

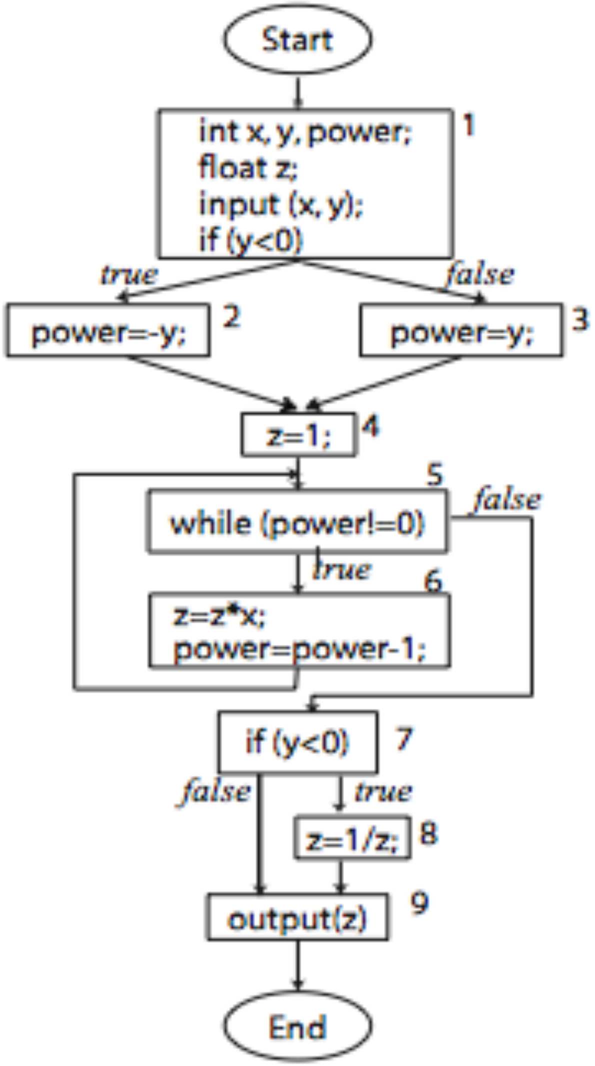
Contents

Paths: infeasible

A path p through a flow graph for program P is considered feasible if there exists at least one test case which when input to P causes p to be traversed.

p_1= (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)

p_2= (Start, 1, 1, 2, 4, 5, 7, 9, , End)



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Number of paths

There can be many distinct paths through a program. A program with no condition contains exactly one path that begins at node Start and terminates at node End.

Each additional condition in the program can increases the number of distinct paths by at least one.

Depending on their location, conditions can have a multiplicative effect on the number of paths.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

2.6 Strings, languages, and regular expressions

Strings

Strings play an important role in testing. A string serves as a test input. Examples: 1011; AaBc; “Hello world”.

A collection of strings also forms a language. For example, a set of all strings consisting of zeros and ones is the language of binary numbers. In this section we provide a brief introduction to strings and languages.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Alphabet

A collection of symbols is known as an alphabet. We use an upper case letter such as X and Y to denote alphabets.

Though alphabets can be infinite, we are concerned only with finite alphabets. For example, X={0, 1} is an alphabet consisting of two symbols 0 and 1. Another alphabet is Y={dog, cat, horse, lion}that consists of four symbols ``dog", ``cat", ``horse", and ``lion".

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Strings over an Alphabet

A string   over an alphabet X is any sequence of zero or more symbols that belong to X. For example, 0110 is a string over the alphabet {0, 1}. Also, dog cat dog dog lion is a string over the alphabet {dog, cat, horse, lion}.

We will use lower case letters such as p, q, r to denote strings. The length of a string is the number of symbols in that string. Given a string s, we denote its length by |s|. Thus, |1011|=4 and |dog cat dog|=3. A string of length 0, also known as an empty string, is denoted by ε.

Note that ε denotes an empty string and also stands for "element of" when used with sets.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

String concatenation

Let s1 and s2 be two strings over alphabet X. We write s1.s2 to denote the concatenation of strings s1 and s2.

For example, given the alphabet X={0, 1}, and two strings 011 and 101 over X, we obtain 011.101=011101. It is easy to see that |s1.s2|=|s1|+|s2|. Also, for any string s, we have s. ε =s and ε.s=s.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Languages

A set L of strings over an alphabet X is known as a language. A language can be finite or infinite.

The following sets are finite languages over the binary alphabet {0, 1}:

∅: The empty set

{ε}: A language consisting only of one string of length zero

{00, 11, 0101}: A language containing three strings

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Regular expressions

Given a finite alphabet X, the following are regular expressions over X:

If a belongs to X, then a is a regular expression that denotes the set {a}.

Let r1 and r2 be two regular expressions over the alphabet X that denote, respectively, sets L1 and L2. Then r1.r2 is a regular expression that denotes the set L1.L2.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Regular expressions (contd.)

If r is a regular expression that denotes the set L then r^+ is a regular expression that denotes the set obtained by concatenating L with itself one or more times also written as L^+ Also, r^* known as the Kleene closure of r, is a regular expression. If r denotes the set L then r^* denotes the set $\{\varepsilon\} \cup L^+$.

If r1 and r2 are regular expressions that denote, respectively, sets L1 and L2, then r1r2 is also a regular expression that denotes the set L1 ∪ L2.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary

We have introduced mathematical preliminaries an understanding of which will be useful while you go through the remaining parts of this book. Exercises at the end of Chapter 2 will help you sharpen your understanding.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 3

Domain Partitioning

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 12, 2013

Contents

Learning Objectives

- Equivalence class partitioning
- Boundary value analysis

Essential black-box techniques for generating tests for functional testing.

Cause effect graphing has been omitted from these slides.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Applications of test generation techniques

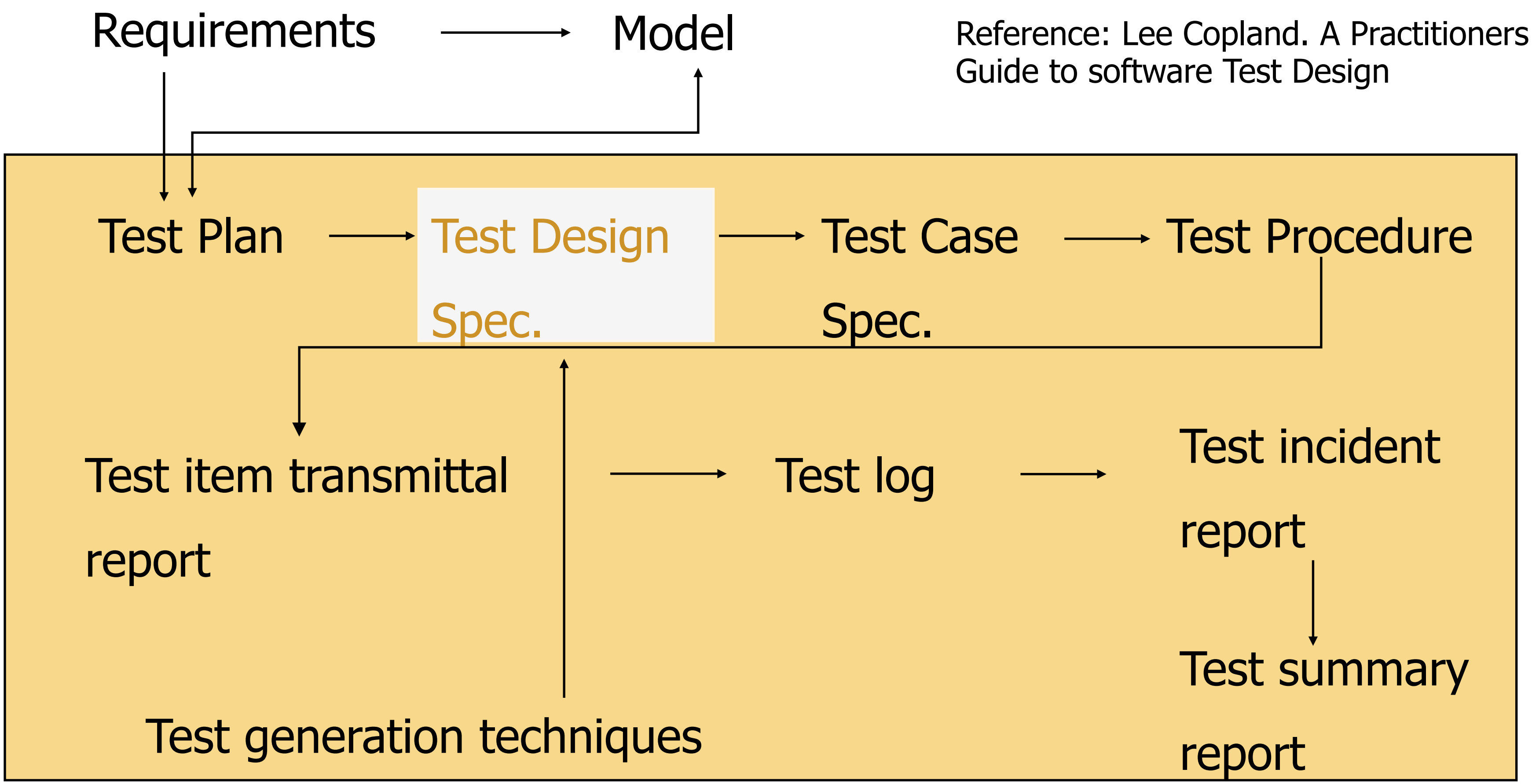
Test generation techniques described in this chapter belong to the black-box testing category.

These techniques are useful during functional testing where the objective is to test whether or not an application, unit, system, or subsystem, correctly implements the functionality as per the given requirements

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Functional Testing: Test Documents (IEEE829 Standard)



Reference: Lee Copland. A Practitioners Guide to software Test Design

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Functional Testing: Documents

Test Plan: Describe scope, approach, resources, test schedule, items to be tested, deliverables, responsibilities, approvals needed. Could be used at the system test level or at lower levels.

Test design spec: Identifies a subset of features to be tested and identifies the test cases to test the features in this subset.

Test case spec: Lists inputs, expected outputs, features to be tested by this test case, and any other special requirements e.g. setting of environment variables and test procedures. Dependencies with other test cases are specified here. Each test case has a unique ID for reference in other documents.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Functional Testing: Documents (contd)

Test procedure spec: Describe the procedure for executing a test case.

Test transmittal report: Identifies the test items being provided for testing, e.g. a database.

Test log: A log observations during the execution of a test.

Test incident report: Document any special event that is recommended for further investigation.

Test summary: Summarize the results of testing activities and provide an evaluation.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation techniques in this chapter

Three techniques are considered: equivalence partitioning, boundary value analysis, and category partitioning.

Each of these test generation techniques is black-box and useful for generating test cases during functional testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

3.2 The test selection problem

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Requirements and test generation

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people.

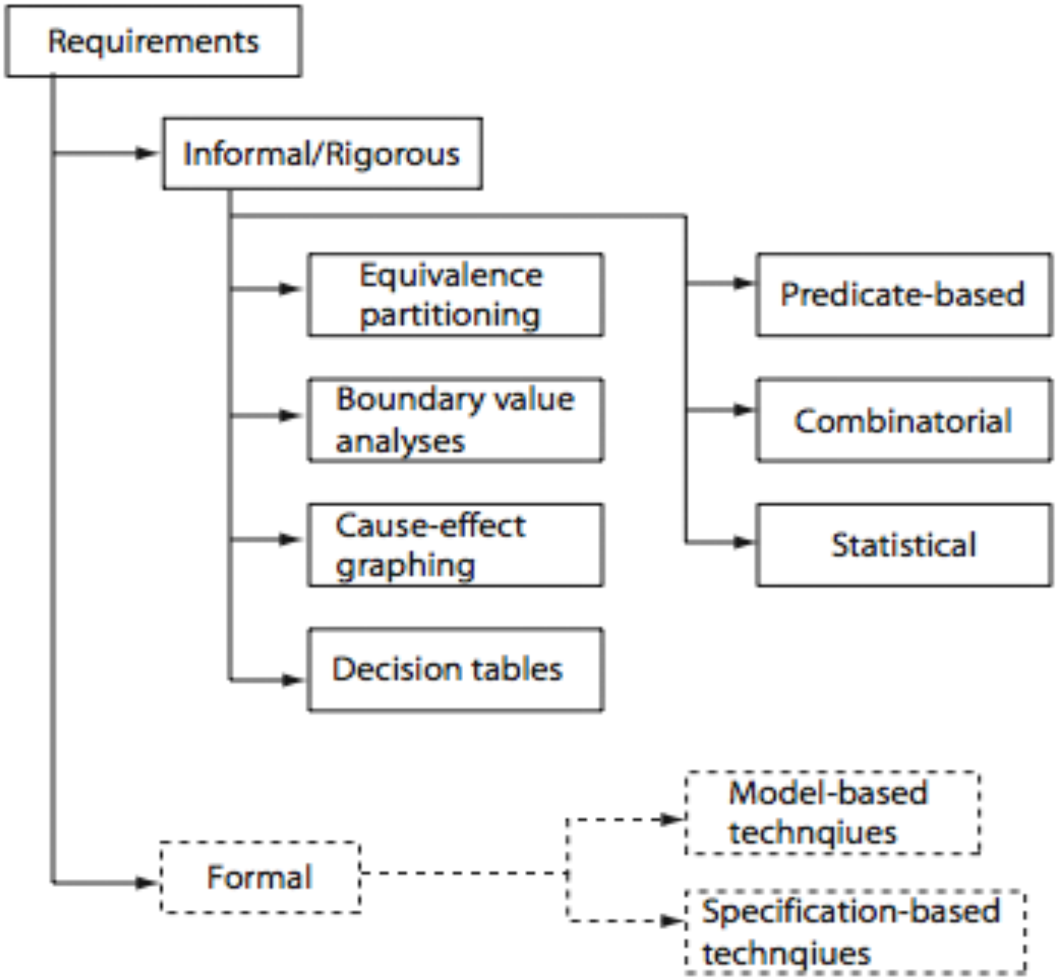
These requirements, more aptly ideas, are then specified rigorously using modeling elements such as use cases, sequence diagrams, and statecharts in UML.

Rigorously specified requirements are often transformed into formal requirements using requirements specification languages such as Z, S, and RSML.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation techniques



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test selection problem

Let D denote the input domain of a program P. The test selection problem is to select a subset T of tests such that execution of P against each element of T will reveal all errors in P.

In general there does not exist any algorithm to construct such a test set. However, there are heuristics and model based methods that can be used to generate tests that will reveal certain type of faults.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test selection problem (contd.)

The challenge is to construct a test set $T \subseteq D$ that will reveal as many errors in P as possible. The problem of test selection is difficult due primarily to the size and complexity of the input domain of P.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Exhaustive testing

The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By ``exhaustive'' testing we mean testing the given program against every element in its input domain.

The complexity makes it harder to select individual tests.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Large input domain

Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of P consists of all possible sequences of integers in the range [-32768, 32767].

If there is no limit on the size of the sequence that can be input, then the input domain of P is infinitely large and P can never be tested exhaustively. If the size of the input sequence is limited to, say $N_{max}>1$, then the size of the input domain depends on the value of N.

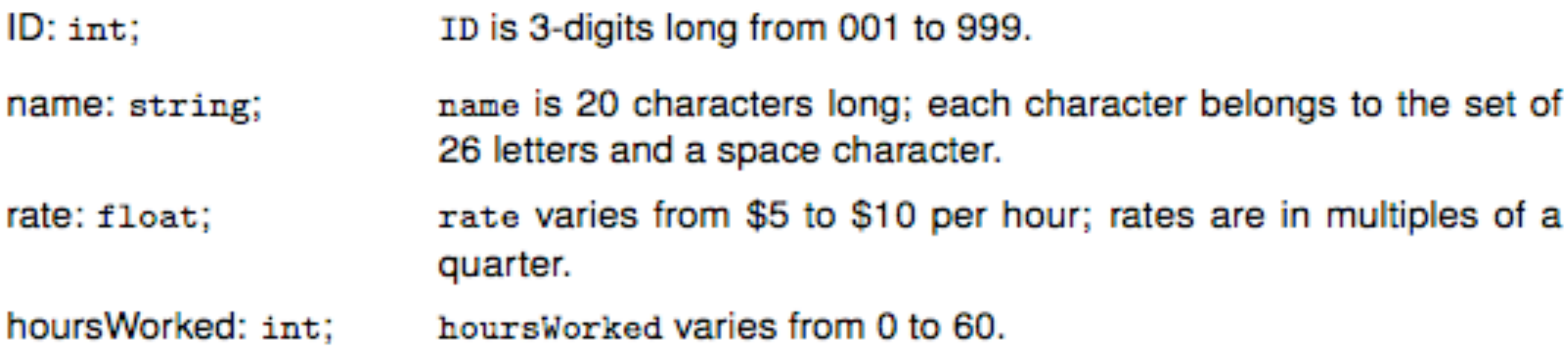
Calculate the size of the input domain.

Contents

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Complex input domain

Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints:



ID: `int;`	`ID` is 3-digits long from 001 to 999.
name: `string;`	`name` is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: `float;`	`rate` varies from $5 to $10 per hour; rates are in multiples of a quarter.
hoursWorked: `int;`	`hoursWorked` varies from 0 to 60.

Calculate the size of the input domain.

Contents

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

3.3 Equivalence partitioning

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence partitioning

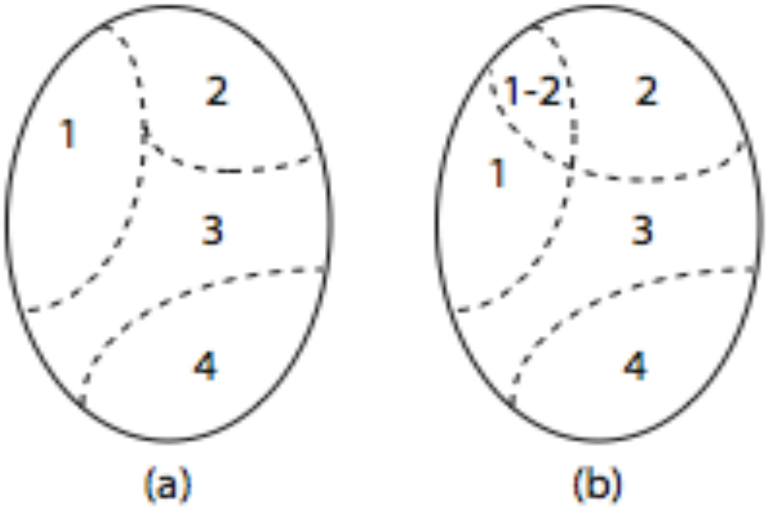
Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown (next slide (a)).

In strict mathematical terms, the sub-domains by definition are disjoint.   The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an equivalence class.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Subdomains



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Program behavior and equivalence classes

The equivalence classes are created assuming that the program under test exhibits the same behavior on all elements, i.e. tests, within a class.

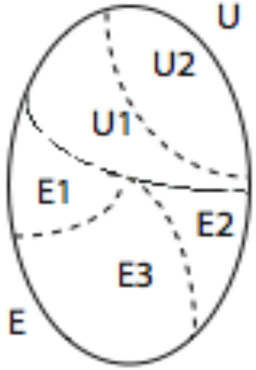
This assumption allow the tester to select exactly one test from each equivalence class resulting in a test suite of exactly N tests.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Faults targeted

The entire set of inputs to any application can be divided into at least two subsets: one containing all the expected, or legal, inputs (E) and the other containing all unexpected, or illegal, inputs (U).

Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g. E1, E2, E3, and U1, U2).

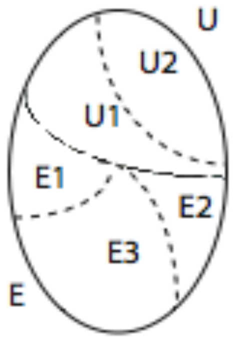


Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Faults targeted (contd.)

Equivalence class partitioning selects tests that target any faults in the application that cause it to behave incorrectly when the input is in either of the two classes or their subsets.

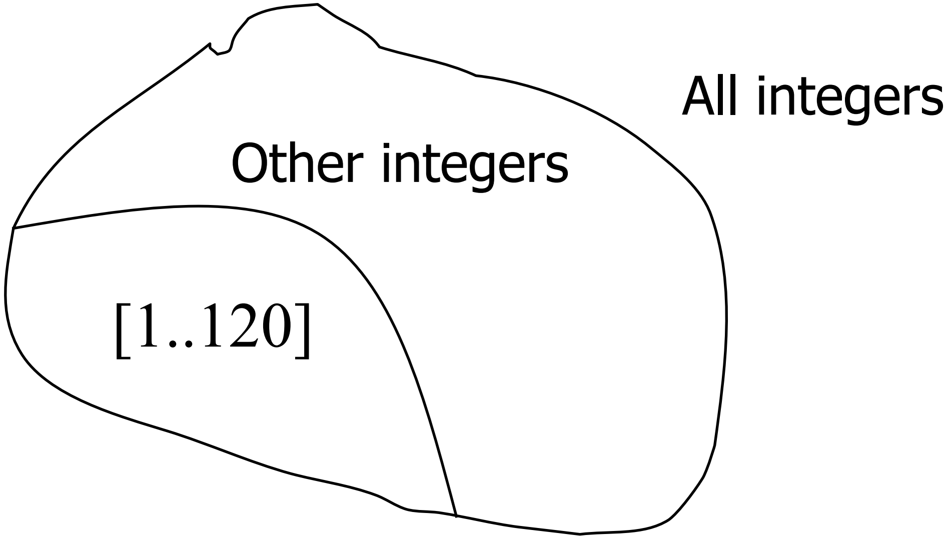


Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example 1

Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example 1 (contd.)

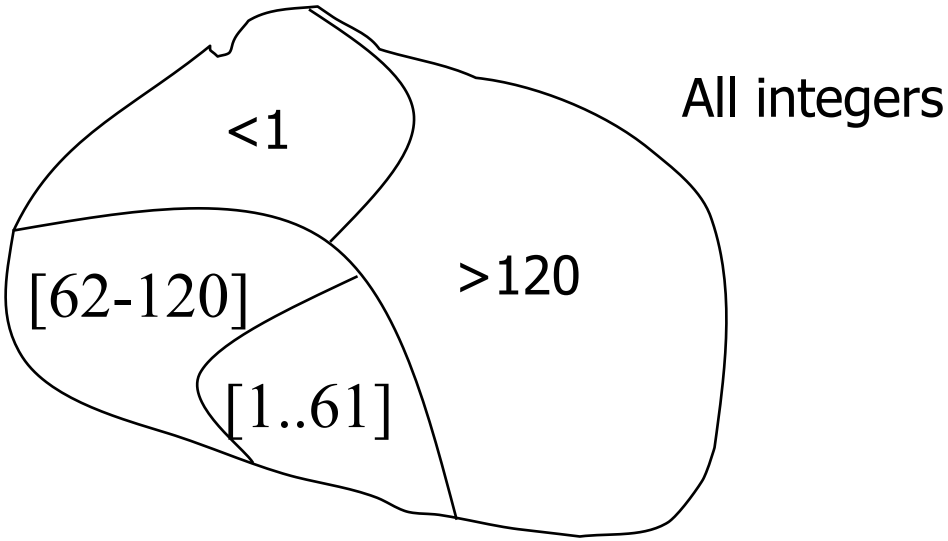
Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2.

Thus, E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example 1 (contd.)



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example 1 (contd.)

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e., two regions containing expected inputs and two regions containing the unexpected inputs.

It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Effectiveness

The effectiveness of tests generated using equivalence partitioning for testing application A, is judged by the ratio of the number of faults these tests are able to expose to the total faults lurking in A.

As is the case with any test selection technique in software testing, the effectiveness of tests selected using equivalence partitioning is less than 1 for most practical applications. The effectiveness can be improved through an unambiguous and complete specification of the requirements and carefully selected tests using the equivalence partitioning technique described in the following sections.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example 2

This example shows a few ways to define equivalence classes based on the knowledge of requirements and the program text.

Consider that wordCount method takes a word w and a filename f as input and returns the number of occurrences of w in the text contained in the file named f. An exception is raised if there is no file with name f.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example 2 (contd.)

```
begin
    String w, f
    Input w, f
    if (not exists(f) {raise exception; return(0);}
    if(length(w)==0)return(0);
    if(empty(f))return(0);
    return(getCount(w,f));
end
```

Using the partitioning method described in the examples above, we obtain the equivalence classes (next slide).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example 2 (contd.)

Equivalence class	w	f
E1	non-null	exists, not empty
E2	non-null	does not exist
E3	non-null	exists, empty
E4	null	exists, not empty
E5	null	does not exist
E6	null	exists, empty

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example 2 (contd.)

Note that the number of equivalence classes without any knowledge of the program code is 2, whereas the number of equivalence classes derived with the knowledge of partial code is 6.

Of course, an experienced tester will likely derive the six equivalence classes given above, and perhaps more, even before the code is available

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes based on program output

In some cases the equivalence classes are based on the output generated by the program. For example, suppose that a program outputs an integer.

It is worth asking: ``Does the program ever generate a 0? What are the maximum and minimum possible values of the output?"

These two questions lead to two the following equivalence classes based on outputs:

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Equivalence classes based on program output (contd.)

E1: Output value v is 0.

E2: Output value v is the maximum possible.

E3: Output value v is the minimum possible.

E4: All other output values.

Based on the output equivalence classes one may now derive equivalence classes for the inputs. Thus, each of the four classes given above might lead to one equivalence class consisting of inputs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	speed ∈[60..90]	{50}, {75}, {92}
	area: float area≥0.0	{{-1.0}, {15.52}}
	age: int	{{-1}, {56}, {132}}
	letter:bool	{{J}, {3}}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes for variables: strings

Equivalence Classes	Example	
	Constraints	Classes
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Loooong Name}}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes for variables: enumeration

Equivalence Classes	Example	
	Constraints	Classes
Each value in a separate class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	up:boolean	{{true}, {false}}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes for variables: arrays

Equivalence Classes	Example	
	Constraints	Classes
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [ ] aName: new int[3];	{[ ]}, {[-10, 20]}, {[-9, 0, 12, 15]}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Equivalence classes for variables: compound data type

Arrays in Java and records, or structures, in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object.

While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure. The next example illustrates the derivation of equivalence classes for an input variable that has a compound type.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Equivalence classes for variables: compound data type: Example

```
struct transcript
        {
                string fName; // First name.
                string lName; //  Last name.
                string cTitle [200]; // Course titles.
                char grades [200]; // Letter grades corresponding
                        to course titles.
}
```

In-class exercise: Derive equivalence classes for each component of R and combine them!

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

uni-dimensional partitioning

One way to partition the input domain is to consider one input variable at a time. Thus, each input variable leads to a partition of the input domain. We refer to this style of partitioning as uni-dimensional equivalence partitioning or simply uni-dimensional partitioning.

This type of partitioning is used commonly.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multidimensional partitioning

Another way is to consider the input domain I as the set product of the input variables and define a relation on I. This procedure creates one partition consisting of several equivalence classes. We refer to this method as multidimensional equivalence partitioning or simply multidimensional partitioning.

Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Partitioning Example

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: $3\leq x\leq 7$ and $5\leq y\leq 9$.

For uni-dimensional partitioning we apply the partitioning guidelines to x and y individually. This leads to the following six equivalence classes.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Partitioning Example (contd.)

E1: x<3 E2: 3≤x≤7 E3: x>7 ← y ignored.

E4: y<5 E5: 5≤y≤9 E6: y>9 ← x ignored.

For multidimensional partitioning we consider the input domain to be the set product X x Y. This leads to 9 equivalence classes.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Partitioning Example (contd.)

E1: $x<3$, $y<5$	E2: $x<3$, $5 \leq y \leq 9$	E3: $x<3$, $y>9$
E4: $3 \leq x \leq 7$, $y<5$	E5: $3 \leq x \leq 7$, $5 \leq y \leq 9$	E6: $3 \leq x \leq 7$, $y>9$
E7: >7, $y<5$	E8: $x>7$, $5 \leq y \leq 9$	E9: $x>7$, $y>9$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Partitioning Example (contd.)

6 equivalence classes:

E1: x<3, y<5

E3: x<3, y>9

E2: x<3, 5≤y≤9

E4: 3≤x≤7, y<5

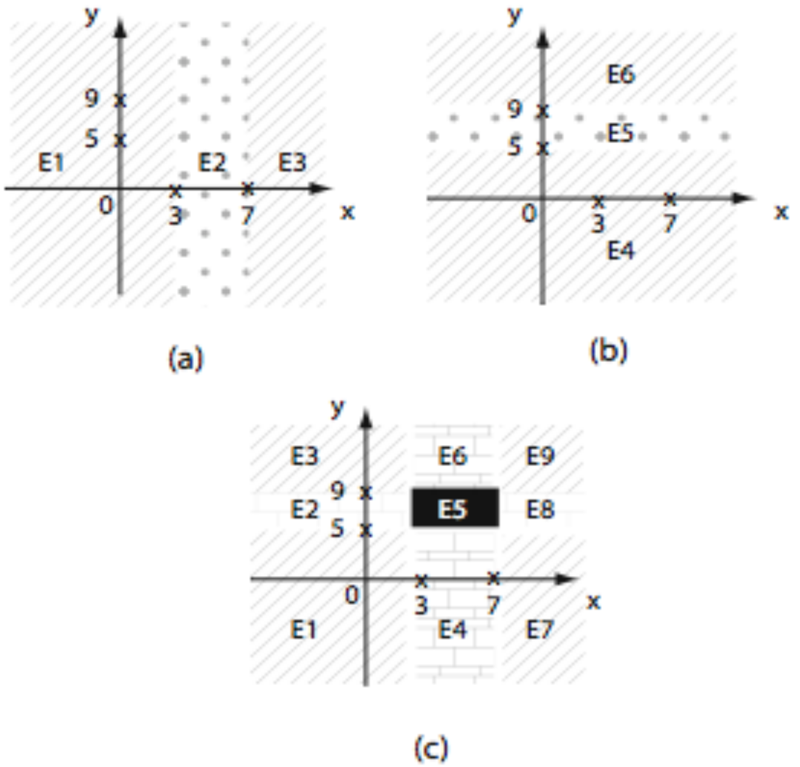
E5: 3≤x≤7, 5≤y≤9

E6: 3≤x≤7, y>9

E7: >7, y<5

E8: x>7, 5≤y≤9

E9: x>7, y>9



(a)

(b)

(c)

9 equivalence classes:

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Systematic procedure for equivalence partitioning

1. Identify the input domain: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.

Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Systematic procedure for equivalence partitioning (contd.)

2. Equivalence classing: Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. partitioning the input domain using values of one variable, is done based on the the expected behavior of the program.

Values for which the program is expected to behave in the ``same way'' are grouped together. Note that ``same way'' needs to be defined by the tester.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Systematic procedure for equivalence partitioning (contd.)

3. Combine equivalence classes: This step is usually omitted and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.

The equivalence classes are combined using the multidimensional partitioning approach described earlier.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Systematic procedure for equivalence partitioning (contd.)

4. Identify infeasible equivalence classes: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test.   Such an equivalence class might arise due to several reasons.

For example, suppose that an application is tested via its GUI, i.e. data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boiler control example (BCS)

The control software of BCS, abbreviated as CS, is required to offer several options. One of the options, C (for control), is used by a human operator to give one of four commands (cmd): change the boiler temperature (temp), shut down the boiler (shut), and cancel the request (cancel).

Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (tempch).

Values of tempch are in the range -10..10 in increments of 5 degrees Fahrenheit. An temperature change of 0 is not an option.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BCS: example (contd.)

Selection of option C forces the BCS to examine variable V. If V is set to GUI, the operator is asked to enter one of the three commands via a GUI. However, if V is set to file, BCS obtains the command from a command file.

The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is temp. The file name is obtained from variable F.

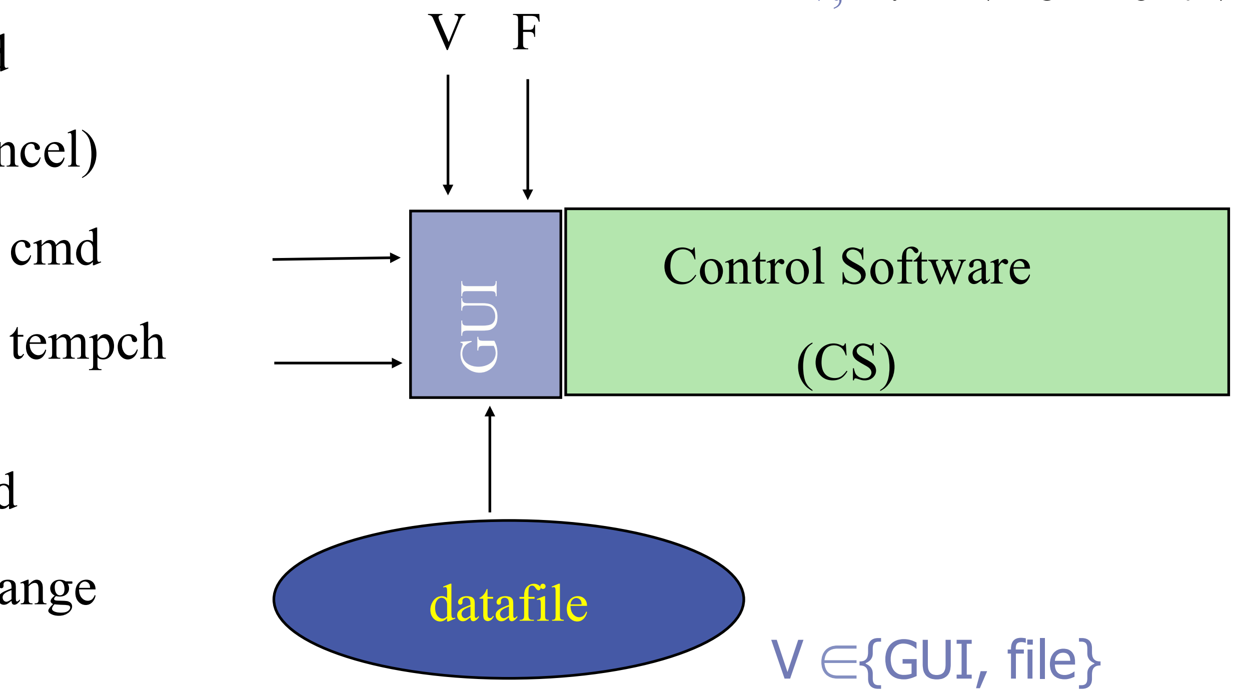
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: example (contd.)

V, F: Environment variables

cmd: command

(temp, shut, cancel)



tempch: desired

temperature change

(-10..10)

V ∈{GUI, file}

F: file name if V is set to "file."

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: example (contd.)

Values of V and F can be altered by a different module in BCS.

In response to temp and shut commands, the control software is required to generate appropriate signals to be sent to the boiler heating system.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: example (contd.)

We assume that the control software is to be tested in a simulated environment. The tester takes on the role of an operator and interacts with the CS via a GUI.

The GUI forces the tester to select from a limited set of values as specified in the requirements. For example, the only options available for the value of tempch are -10, -5, 5, and 10. We refer to these four values of tempch as tvalid while all other values as tinvalid.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BCS: 1. Identify input domain

The first step in generating equivalence partitions is to identify the (approximate) input domain. Recall that the domain identified in this step will likely be a superset of the complete input domain of the control software.

First we examine the requirements, identify input variables, their types, and values. These are listed in the following table.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: Variables, types, values

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	File, GUI
F	Environment	String	A file name
cmd	Input via GUI/File	Enumerated	{temp, cancel, shut}
tempch	Input via GUI/File	Enumerated	{-10, -5, 5, 10}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BCS: Input domain

Input domain⊆S=V×F×cmd×tempch

Sample values in the input domain (--: don't care):

(GUI, --, shut, --), (file, cmdfile, shut, --)

(file, cmdfile, temp, 0) ⟵ *Does this belong to the input domain?*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: 2. Equivalence classing

Variable	Partition
V	{{GUI}, {file}, {undefined}}
F	{{fvalid}, {finvalid}}
cmd	{{temp}, {cancel}, {shut}, {cinvalid}}
tempch	{{tvalid}, {tinvalid}}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: 3. Combine equivalence classes (contd.)

Note that tinvalid, tvalid, finvalid, and fvalid denote sets of values. “undefined” denotes one value.

There are a total of 3×4×2×5=120 equivalence classes.

Sample equivalence class: {(GUI, fvalid, temp, -10)}

Note that each of the classes listed above represents an infinite number of input values for the control software. For example, {(GUI}}, fvalid, temp, -10)} denotes an infinite set of values obtained by replacing fvalid by a string that corresponds to the name of an existing file. Each value is a potential input to the BCS.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: 4. Discard infeasible equivalence classes

Note that the GUI requests for the amount by which the boiler temperature is to be changed only when the operator selects temp for cmd. Thus, all equivalence classes that match the following template are infeasible.

{(V, F, {cancel, shut, cinvalid}, tvalid∪ tinvalid)}

This parent-child relationship between cmd and tempch renders infeasible a total of 3×2×3×5=90 equivalence classes.

Exercise: How many additional equivalence classes are infeasible?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BCS: 4. Discard infeasible equivalence classes (contd.)

After having discarded all infeasible equivalence classes, we are left with a total of 18 testable (or feasible) equivalence classes.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Selecting test data

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values.

In the most general case, a tester simply selects one test that serves as a representative of each equivalence class.

Exercise: Generate sample tests for BCS from the remaining feasible equivalence classes.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

GUI design and equivalence classes

While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry. For example, the requirement for an application.

The application places a constraint on an input variable X such that it can assume integral values in the range 0..4. However, testing must account for the possibility that a user may inadvertently enter a value for X that is out of range.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

GUI design and equivalence classes (contd.)

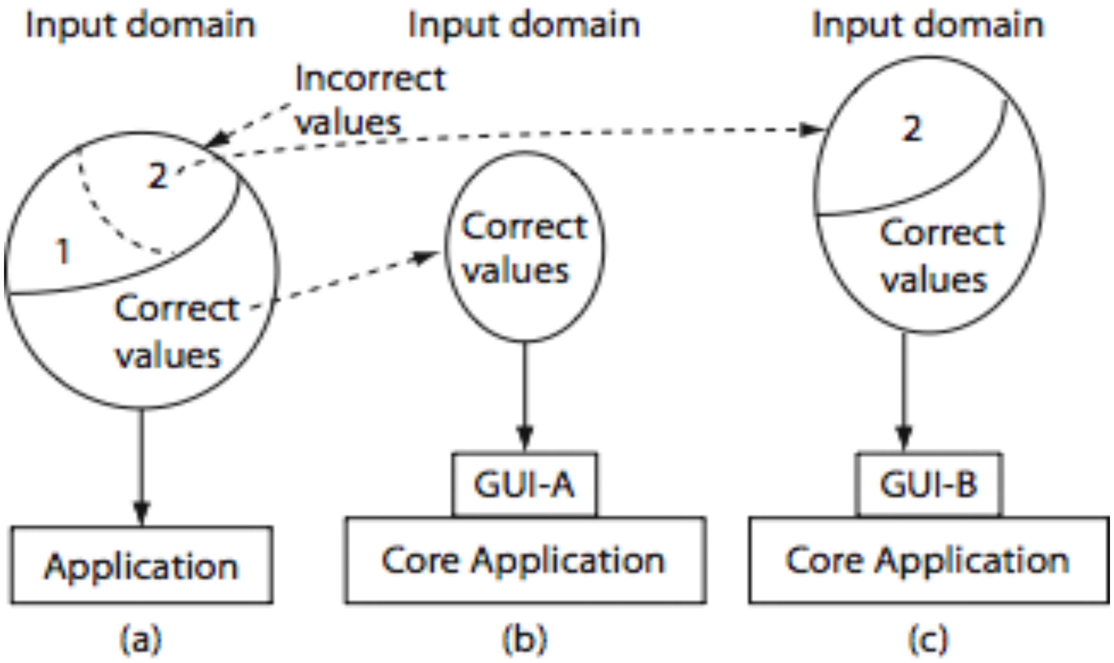
Suppose that all data entry to the application is via a GUI front end. Suppose also that the GUI offers exactly five correct choices to the user for X.

In such a situation it is impossible to test the application with a value of X that is out of range. Hence only the correct values of X will be input. See figure on the next slide.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

GUI design and equivalence classes (contd.)



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

3.4 Boundary value analysis

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Errors at the boundaries

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes.

For example, suppose that method M is required to compute a function f1 when x≤ 0 is true and function f2 otherwise. However, M has an error due to which it computes f1 for x<0 and f2 otherwise.

Obviously, this fault is revealed, though not necessarily, when M is tested against x=0 but not if the input test set is, for example, {-4, 7} derived using equivalence partitioning. In this example, the value x=0, lies at the boundary of the equivalence classes x≤0 and x>0.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boundary value analysis (BVA)

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may overlap.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BVA: Procedure

1 Partition the input domain using uni-dimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.

2 Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.

3 Select test data such that each boundary value occurs in at least one test input.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BVA: Example: 1. Create equivalence classes

Assuming that an item code must be in the range 99..999 and quantity in the range 1..100,

Equivalence classes for code:

E1: Values less than 99.

E2: Values in the range.

E3: Values greater than 999.

Equivalence classes for qty:

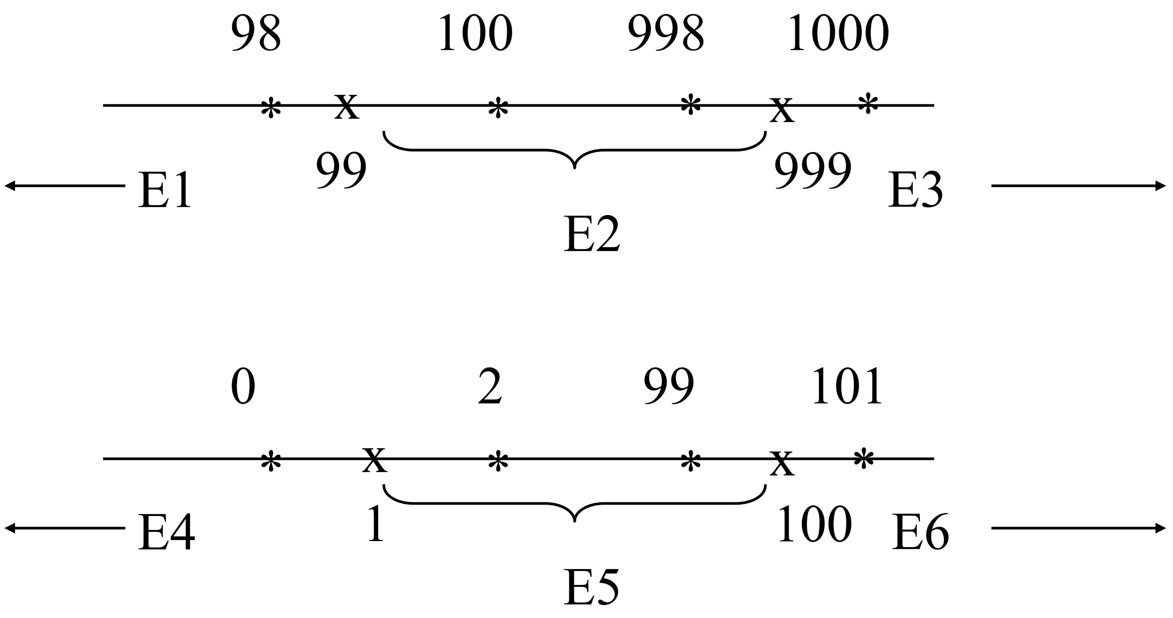
E4: Values less than 1.

E5: Values in the range.

E6: Values greater than 100.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BVA: Example: 2. Identify boundaries



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Equivalence classes and boundaries for findPrice. Boundaries are indicated with an x. Points near the boundary are marked *.

Contents

BVA: Example: 3. Construct test set

Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Consider the following test set:

```
T={     t1: (code=98, qty=0),
        t2: (code=99, qty=1),
        t3: (code=100, qty=2),
        t4: (code=998, qty=99),
        t5: (code=999, qty=100),
        t6: (code=1000, qty=101)
}
```

Illegal values of code and qty included.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BVA: In-class exercise

Is T the best possible test set for findPrice? Answer this question based on T’s ability to detect missing code for checking the validity of age.

Is there an advantage of separating the invalid values of code and age into different test cases?

Answer: Refer to Example 3.11.

Highly recommended: Go through Example 3.12.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BVA: Recommendations

Relationships amongst the input variables must be examined carefully while identifying boundaries along the input domain. This examination may lead to boundaries that are not evident from equivalence classes obtained from the input and output variables.

Additional tests may be obtained when using a partition of the input domain obtained by taking the product of equivalence classes created using individual variables.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

4.4. Tests using predicate syntax

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Where do predicates arise?

Predicates arise from requirements in a variety of applications. Here is an example from Paradkar, Tai, and Vouk, "Specification based testing using cause-effect graphs," Annals of Software Engineering," V 4, pp 133-157, 1997.

A boiler needs to be shut down when the following conditions hold:

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boiler shutdown conditions

1. The water level in the boiler is below X lbs. (a)
2. The water level in the boiler is above Y lbs. (b)
3. A water pump has failed. (c)
4. A pump monitor has failed. (d)
5. Steam meter has failed. (e)

} Boiler in degraded mode when either is true.

The boiler is to be shut down when a or b is true or the boiler is in degraded mode and the steam meter fails. We combine these five conditions to form a compound condition (predicate) for boiler shutdown.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boiler shutdown conditions

Denoting the five conditions above as a through e, we obtain the following Boolean expression E that when true must force a boiler shutdown:

$$E=a+b+(c+d)e$$

where the + sign indicates “OR” and a multiplication indicates “AND.”

The goal of predicate-based test generation is to generate tests from a predicate p that guarantee the detection of any error that belongs to a class of errors in the coding of p.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Another example

A condition is represented formally as a predicate, also known as a Boolean expression. For example, consider the requirement

``*if the printer is ON and has paper then send document to printer*."

This statement consists of a condition part and an action part. The following predicate represents the condition part of the statement.

p_r: (printer_status=ON) $\wedge$ (printer_tray!= empty)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary

Equivalence partitioning and boundary value analysis are the most commonly used methods for test generation while doing functional testing.

Given a function *f* to be tested in an application, one can apply these techniques to generate tests for *f*.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 4

Predicate Analysis

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 12, 2013

Contents

Learning Objectives

- Domain testing
- Cause-effect graphing
- Test generation from predicates

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

4.4 Tests using predicate syntax

4.4.1: A fault model

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Fault model for predicate testing

What faults are we targeting when testing for the correct implementation of predicates?

Boolean operator fault: Suppose that the specification of a software module requires that an action be performed when the condition $(a<b) \vee (c>d) \wedge e$ is true.

Here a, b, c, and d are integer variables and e is a Boolean variable.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Boolean operator faults

Correct predicate: $(a<b) \lor (c>d) \land e$

$(a<b) \land (c>d) \land e$	Incorrect Boolean operator
$(a<b) \lor\ !\ (c>d) \land e$	Incorrect negation operator
$(a<b) \land (c>d) \lor e$	Incorrect Boolean operators
$(a<b) \lor (e>d) \land c$	Incorrect Boolean variable.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Relational operator faults

Correct predicate: $(a<b) \vee (c>d) \wedge e$

$(a==b) \vee (c>d) \wedge e$	Incorrect relational operator
$(a==b) \vee (c\leq d) \wedge e$	Two relational operator faults
$(a==b) \vee (c>d) \vee e$	Incorrect Boolean operators

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Arithmetic expression faults

Correct predicate: Ec: e1 relop1 e2. Incorrect predicate: Ei: : e3 relop2 e4. Assume that Ec and Ei use the same set of variables.

Ei has an off-by-ε fault if |e3-e4|= ε for any test case for which e1=e2.

Ei has an off-by-ε* fault if |e3-e4|≥ ε for any test case for which e1=e2.

Ei has an off-by-ε+ fault if |e3-e4|> ε for any test case for which e1=e2.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Arithmetic expression faults: Examples

Correct predicate: Ec: $a<(b+c)$. Assume $\varepsilon=1$.

Ei: $a<b$. Given $c=1$, Ei has an off-by-1 fault as $|a-b|= 1$ for a test case for which $a=b+c$, e.g. $<a=2, b=1, c=1>$.

Ei: $a<b+1$. Given $c=2$, Ei has an off-by-1* fault as $|a-(b+1)|\geq 1$ for any test case for which $a=b+c$; $<a=4, b=2, c=2>$

Ei: $a<b-1$. Given $c>0$, Ei has an off-by-1+ fault as $|a-(b-1)|>1$ for any test case for which $a=b+c$; $<a=3, b=2, c=1>$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Arithmetic expression faults: In class exercise

Given the correct predicate: Ec: $2*X+Y>2$. Assume $\varepsilon=1$.

Find an incorrect version of Ec that has off-by-1 fault.

Find an incorrect version of Ec that has off-by-1* fault.

Find an incorrect version of Ec that has off-by-1+ fault.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Goal of predicate testing

Given a correct predicate p_c, the goal of predicate testing is to generate a test set T such that there is at least one test case $t \in$ T for which p_c and its faulty version p_i, evaluate to different truth values.

Such a test set is said to guarantee the detection of any fault of the kind in the fault model introduced above.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Goal of predicate testing (contd.)

As an example, suppose that p_c: a<b+c and p_i: a>b+c. Consider a test set T={t1, t2} where t1: <a=0, b=0, c=0> and t2: <a=0, b=1, c=1>.

The fault in p_i is not revealed by t1 as both p_c and p_i evaluate to false when evaluated against t1.

However, the fault is revealed by t2 as p_c evaluates to true and p_i to false when evaluated against t2.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Missing or extra Boolean variable faults

Correct predicate: $a \vee b$

Missing Boolean variable fault: a

Extra Boolean variable fault: $a \vee b \wedge c$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

4.4 Tests using predicate syntax

4.4.1: Predicate constraints

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Predicate constraints: BR symbols

Consider the following Boolean-Relational set of BR-symbols:

BR={**t**, **f**, <, =, >, +ε, **-**ε}

A BR symbol is a constraint on a Boolean variable or a relational expression.

For example, consider the predicate E: a<b and the constraint ">" . A test case that satisfies this constraint for E must cause E to evaluate to false.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Infeasible constraints

A constraint C is considered infeasible for predicate p_r if there exists no input values for the variables in p_r that satisfy c.

For example, the constraint **t** is infeasible for the predicate $a>b \wedge b>d$ if it is known that $d>a$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Predicate constraints

Let p_r denote a predicate with n, n>0, $\vee$ and $\wedge$ operators.

A predicate constraint C for predicate p_r is a sequence of (n+1) BR symbols, one for each Boolean variable or relational expression in p_r. When clear from context, we refer to ``predicate constraint'' as simply constraint.

Test case t satisfies C for predicate p_r, if each component of p_r satisfies the corresponding constraint in C when evaluated against t. Constraint C for predicate p_r guides the development of a test for p_r, i.e., it offers hints on what the values of the variables should be for p_r to satisfy C.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

True and false constraints

$p_r(C)$ denotes the value of predicate p_r evaluated using a test case that satisfies C.

C is referred to as a true constraint when $p_r(C)$ is true and a false constraint otherwise.

A set of constraints S is partitioned into subsets S^t and S^f, respectively, such that for each C in S^t, $p_r(C)$ =true, and for any C in S^f, $p_r(C)$ =false. $S= S^t \cup S^f$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Predicate constraints: Example

Consider the predicate p_r: $b \wedge (r<s) \vee (u \geq v)$ and a constraint C: (t, =, >). The following test case satisfies C for p_r.

<b=true, r=1, s=1, u=1, v=0>

The following test case does not satisfy C for p_r.

<b=true, r=1, s=2, u=1, v=2>

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

4.4 Tests using predicate syntax

4.4.3: Predicate testing criteria

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Predicate testing: criteria

Given a predicate p_r, we want to generate a test set T such that

- T is minimal and
- T guarantees the detection of any fault in the implementation of p_r; faults correspond to the fault model we discussed earlier.

We will discuss three such criteria named BOR, BRO, and BRE.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Predicate testing: BOR testing criterion

A test set T that satisfies the BOR testing criterion for a compound predicate p_r, guarantees the detection of single or multiple Boolean operator faults in the implementation of p_r.

T is referred to as a BOR-adequate test set and sometimes written as T_{BOR}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Predicate testing: BRO testing criterion

A test set T that satisfies the BRO testing criterion for a compound predicate p_r, guarantees the detection of single Boolean operator and relational operator faults in the implementation of p_r.

T is referred to as a BRO-adequate test set and sometimes written as T_{BRO}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Predicate testing: BRE testing criterion

A test set T that satisfies the BRE testing criterion for a compound predicate p_r, guarantees the detection of single Boolean operator, relational expression, and arithmetic expression faults in the implementation of p_r.

T is referred to as a BRE-adequate test set and sometimes written as T_{BRE}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Predicate testing: guaranteeing fault detection

Let T_x, $x \in \{BOR, BRO, BRE\}$, be a test set derived from predicate p_r. Let p_f be another predicate obtained from p_r by injecting single or multiple faults of one of three kinds: Boolean operator fault, relational operator fault, and arithmetic expression fault.

T_x is said to guarantee the detection of faults in p_f if for some $t \in T_x$, $p(t) \neq p_f(t)$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Guaranteeing fault detection: example

Let p_r=a<b ∧ c>d

Constraint set S={(t, t), (t,f), (f, t)}

Let $T_{BOR=}${t1, t2, t3} is a BOR adequate test set that satisfies S.

t1: <a=1, b=2, c=1, d=0 >; Satisfies (t, t), i.e. a<b is true and c<d is also true.

t2: <a=1, b=2, c=1, d=2 >; Satisfies (t, f)

t3: <a=1, b=0, c=1, d=0 >; Satisfies (f, t)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Guaranteeing fault detection: In class exercise

Generate single Boolean operator faults in

p_r: $a<b \wedge c>d$

and show that T guarantees the detection of each fault.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

4.4 Tests using predicate syntax

4.4.1: BOR, BRO, and BRE adequate tests

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Algorithms for generating BOR, BRO, and BRE adequate tests

Define the cross product of two sets A and B as:

$$A \times B = \{(a,b) | a \in A \text{ and } b \in B\}$$

The onto product of two sets A and B is defined as:

$$A \otimes B = \{(u,v) | u \in A,\ v \in B, \text{ such that each element of A appears at least once as u and each element of B appears once as v.}\}$$

Note that $A \otimes B$ is a minimal set.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Set products: Example

Let A={t, =, >} and B={f, <}

$A\times B=\{(t, f), (t, <), (=, f), (=, <), (>,f), (>,<)\}$

$A\otimes B =\{(t, f), (=,<), (>,<)\}$

Any other possibilities for $A\otimes B$?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

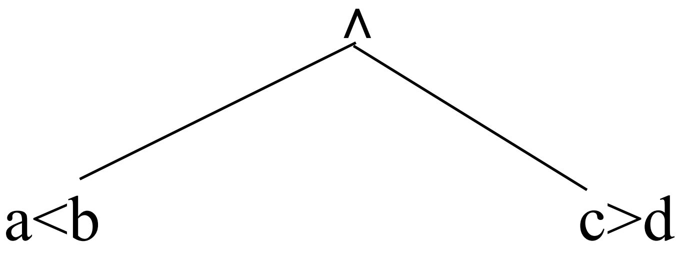
Contents

Generation of BOR constraint set

See page 184 for a formal algorithm. An illustration follows.

We want to generate T_{BOR} for p_r: $a<b \wedge c>d$

First, generate syntax tree of p_r.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Generation of the BOR constraint set

Given node N in the syntax tree for predicate $p_{r,}$ we use the following notation:

$S_N = S_N^t \cup S_N^f$ is the constraint set, where

S_N^t is the true constraint set, and

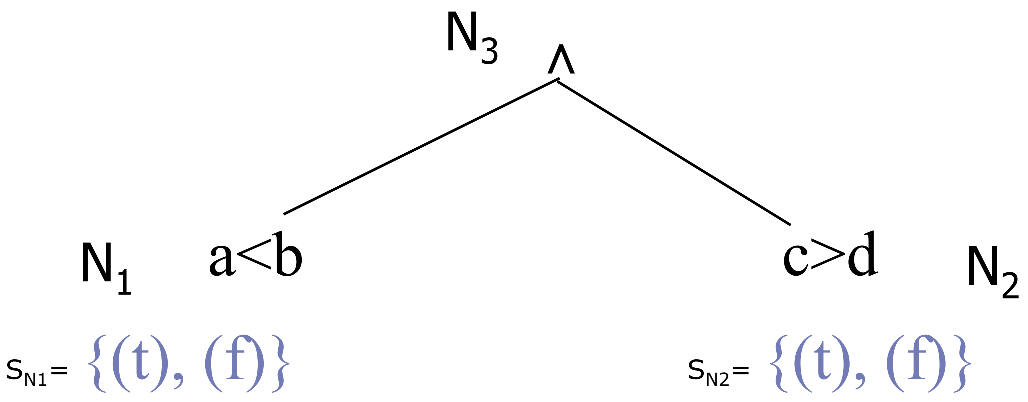
S_N^f is the false constraint.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Generation of the BOR constraint set (contd.)

Second, label each leaf node with the constraint set {(t), (f)}.

We label the nodes as $N_{1,}$ N_2, and so on for convenience.



Notice that N_1 and N_2 are direct descendants of N_3 which is an AND-node.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Generation of the BOR constraint set (contd.)

Third, compute the constraint set for the next higher node in the syntax tree, in this case N_3. For an AND node, the formulae used are the following.

$$S_{N3}^{t} = S_{N1}^{t} \otimes S_{N2}^{t} = \{(t)\} \otimes \{(t)\} = \{(t, t)\}$$

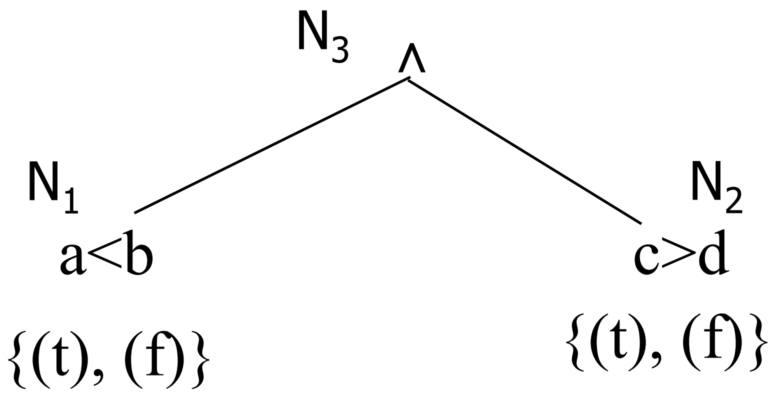
$$S_{N3}^{f} = (S_{N1}^{f} \times \{t_2\}) \cup (\{t_1\} \times S_{N2}^{f})$$

$$= (\{(f)\} \times \{(t)\}) \cup (\{(t)\} \times \{(f)\})$$

$$= \{(f, t)\} \cup \{(t, f)\}$$

$$= \{(f, t), \{(t, f)\}$$

$S_{N3}=\{(t,t), (f, t), (t, f)\}$



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

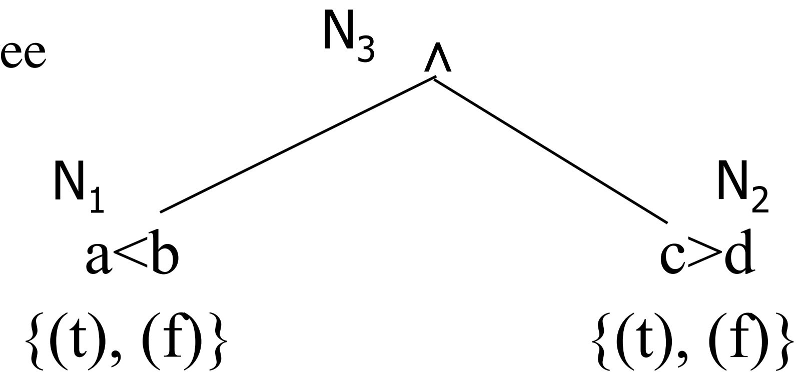
Contents

Generation of T_{BOR}

As per our objective, we have computed the BOR constraint set for the root node of the AST(p_r). We can now generate a test set using the BOR constraint set associated with the root node.

S_{N3} contains a sequence of three constraints and hence we get a minimal test set consisting of three test cases. Here is one possible test set.

S_{N3}={(t,t), (f, t), (t, f)}



T_{BOR} ={t1, t2, t3}

t1=<a=1, b=2, c=6, d=5> (t, t)

t2=<a=1, b=0, c=6, d=5> (f, t)

t3=<a=1, b=2, c=1, d=2> (t, f)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Generation of BRO constraint set

See pages 187-188 for a formal algorithm. An illustration follows.

Recall that a test set adequate with respect to a BRO constraint set for predicate p_r, guarantees the detection of all combinations of single or multiple Boolean operator and relational operator faults.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BRO constraint set

The BRO constraint set S for relational expression *e1 relop e2*:

$S=\{(>), (=), (<)\}$

Separation of S into its true (S^t) and false (S^f)components:

relop: >	$S^t=\{(>)\}$	$S^f=\{(=), (<)\}$
relop: ≥	$S^t=\{(>), (=)\}$	$S^f=\{(<)\}$
relop: =	$S^t=\{(=)\}$	$S^f=\{(<), (>)\}$
relop: <	$S^t=\{(<)\}$	$S^f=\{(=), (>)\}$
relop: ≤	$S^t=\{(<), (=)\}$	$S^f=\{(>)\}$

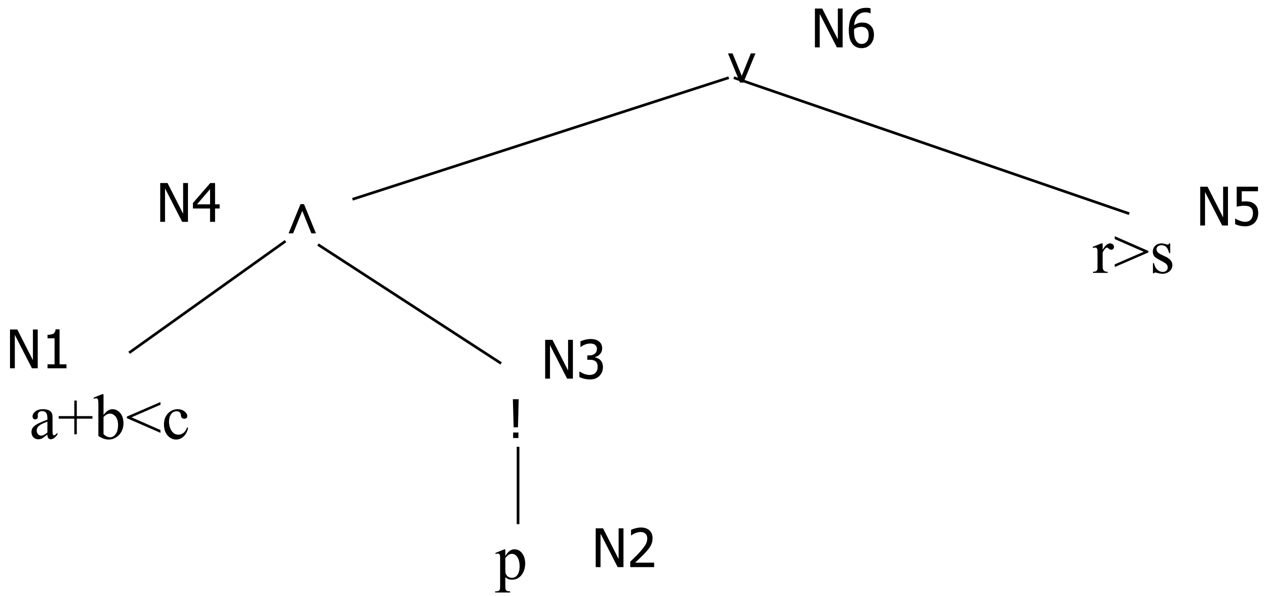
Note: t_N denotes an element of S^t_N and f_N denotes an element of S^f_N

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BRO constraint set: Example

p_r: (a+b<c)∧!p ∨ (r>s)

Step 1: Construct the AST for the given predicate.

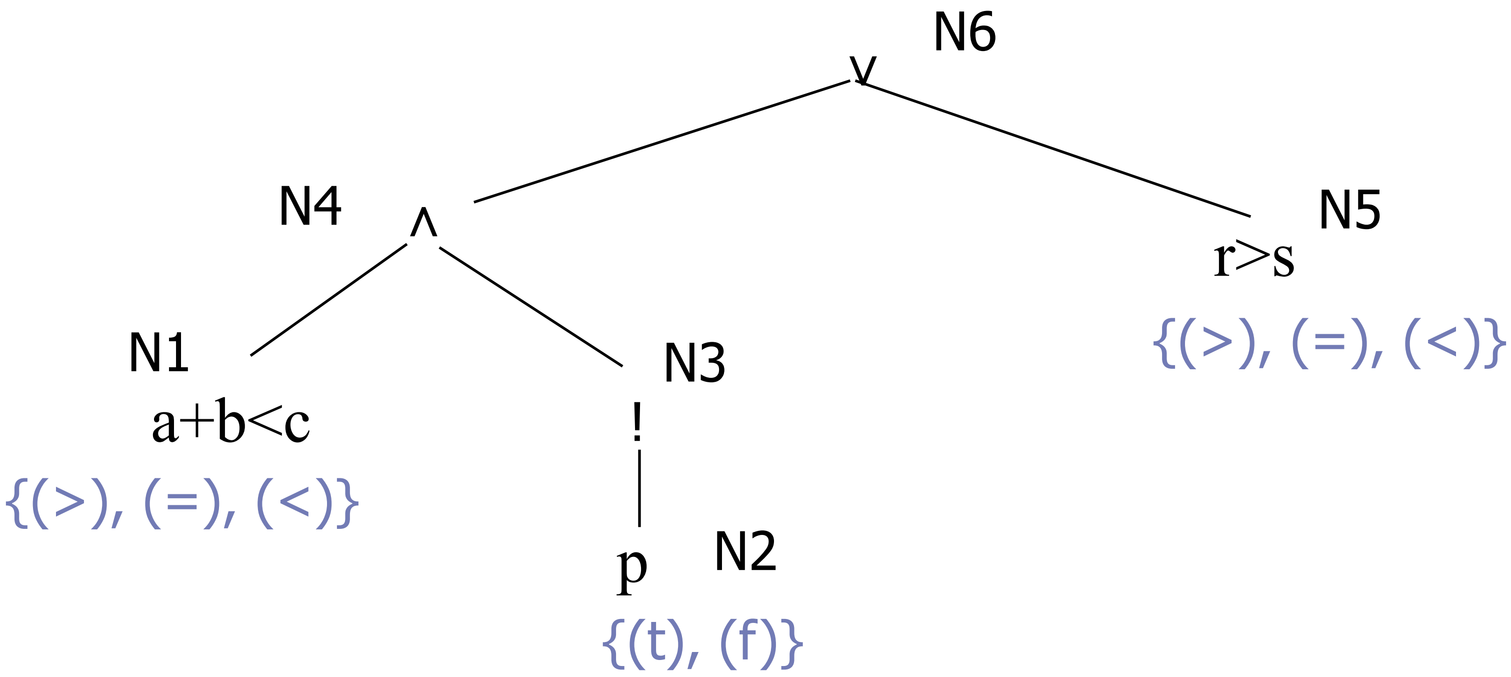


Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: Example (contd.)

Step 2: Label each leaf node with its constraint set S.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: Example (contd.)

Step 2: Traverse the tree and compute constraint set for each internal node.

$S^t_{N3}=S_{N2}{}^f=\{(f)\}$ $\qquad$ $S^f_{N3}=S_{N2}{}^t=\{(t)\}$

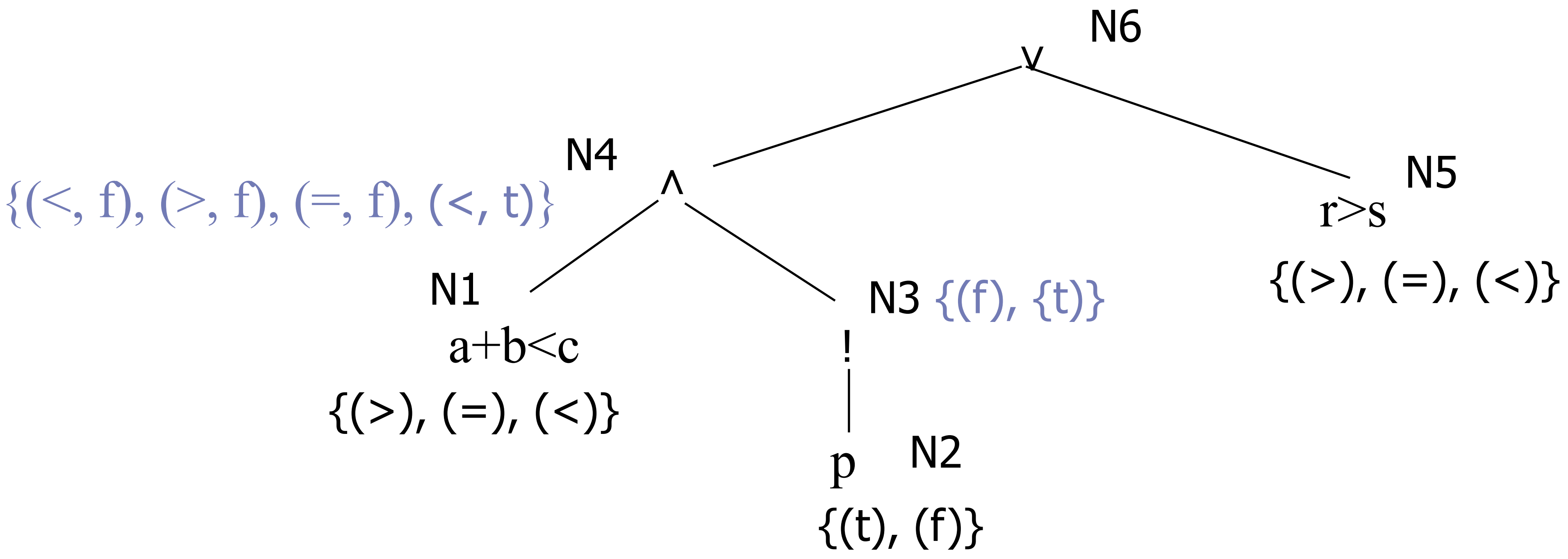
$S^t_{N4}=S_{N1}{}^t \otimes S_{N3}{}^t=\{(<)\} \otimes \{(f)\}=\{(<, f)\}$

$$\begin{aligned} S^f_{N4}= \quad & (S^f_{N1} \times \{(t_{N3})\}) \cup (\{(t_{N1})\} \times S^f_{N3}) \\ & =(\{(>,=)\} \times \{(f)\}) \cup \{(<)\} \times \{(t)\}) \\ & =\{(>, f), (=, f)\} \cup \{(<, t)\} \\ & =\{(>, f), (=, f), (<, t)\} \end{aligned}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: Example (contd.)



N6 ∨

N4 ∧

{(<, f), (>, f), (=, f), (<, t)}

N5 r>s

{(>), (=), (<)}

N1 a+b<c

{(>), (=), (<)}

N3 {(f), {t)}

!

p N2

{(t), (f)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: Example (contd.)

Next compute the constraint set for the rot node (this is an OR-node).

$$S^f_{N6}=S^f_{N4} \otimes S^f_{N5}$$

$$=\{(>,f),(=,f),(<,t)\} \otimes\{(=),(<)\}=\{(<, f)\}$$

$$=\{(>,f,=), (=,f,<),(<,t,=)\}$$

$$S^t_{N6}= \quad (S^t_{N4} \times \{(f_{N5})\})\cup (\{(f_{N4})\} \times S^t_{N5})$$

$$=(\{(<,f)\} \times\{(=)\}) \cup \{(>,f)\} \times\{(>)\})$$

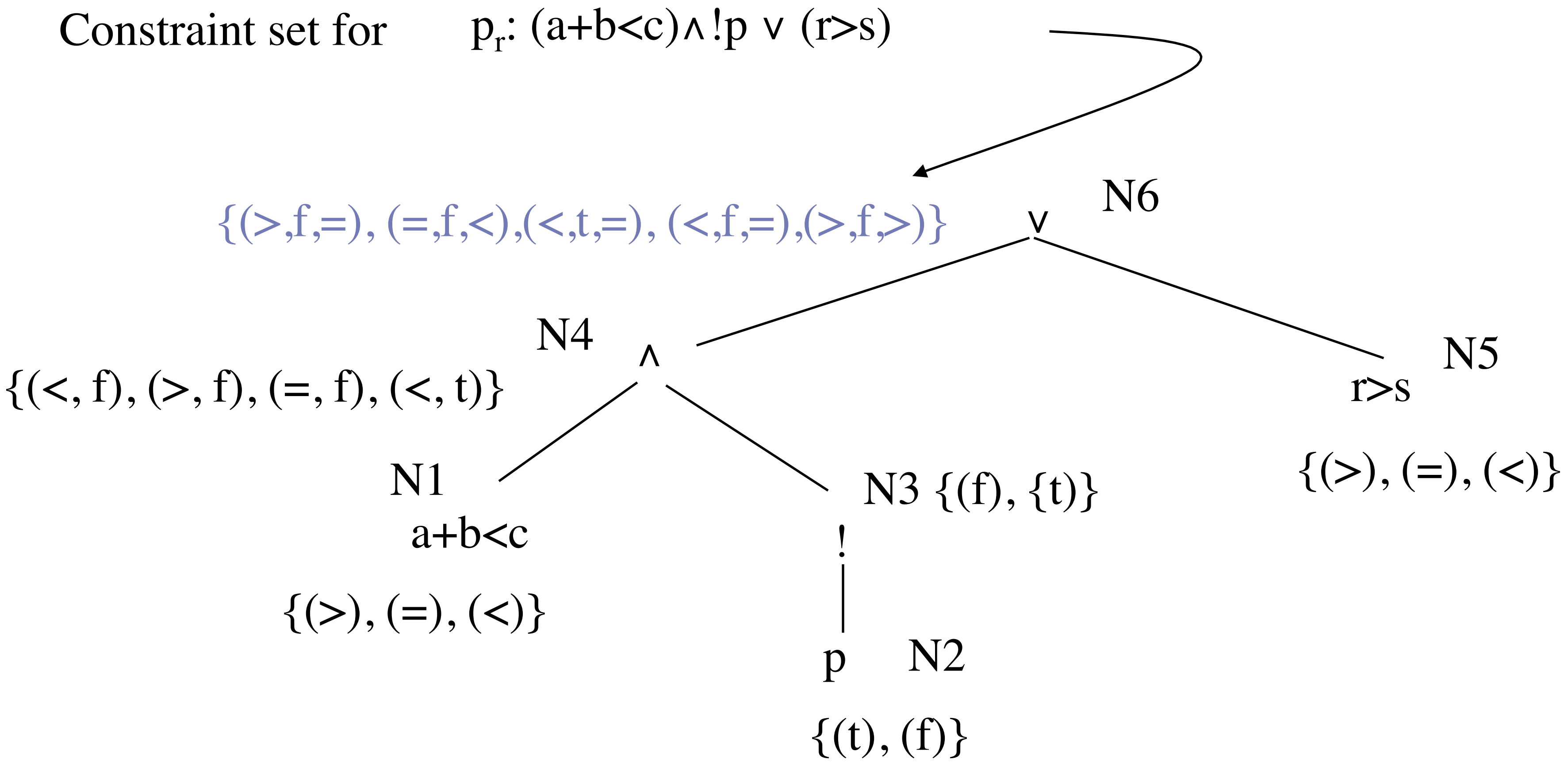
$$=\{(<,f,=)\} \cup \{(>,f,>)\}$$

$$=\{(<,f,=),(>,f,>)\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: Example (contd.)



Constraint set for p_r: $(a+b<c) \wedge !p \vee (r>s)$

{(>,f,=), (=,f,<),(<,t,=), (<,f,=),(>,f,>)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BRO constraint set: In-class exercise

Given the constraint set for p_r: $(a+b<c) \wedge !p \vee (r>s)$, construct T_{BRO} .

{(>,f,=), (=,f,<),(<,t,=), (<,f,=),(>,f,>)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

4.4 Tests using predicate syntax

4.4.5: BOR constraints for non-singular expressions

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR constraints for non-singular expressions

Test generation procedures described so far are for singular predicates. Recall that a singular predicate contains only one occurrence of each variable.

We will now learn how to generate BOR constraints for non-singular predicates.

First, let us look at some non-singular expressions, their respective disjunctive normal forms (DNF), and their mutually singular components.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Non-singular expressions and DNF: Examples

Predicate (p_r)	DNF	Mutually singular components in p_r
ab(b+c)	abb+abc	a; b(b+c)
a(bc+ bd)	abc+abd	a; (bc+bd)
a(!b+!c)+cde	a!ba +a!c+cde	a; !b+!c+ cde
a(bc+!b+de)	abc+a!b+ade	a; bc+!b; de

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Generating BOR constraints for non-singular expressions

We proceed in two steps.

First we examine the Meaning Impact (MI) procedure for generating a minimal set of constraints from a possibly non-singular predicate.

Next, we examine the procedure to generate BOR constraint set for a non-singular predicate.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Meaning Impact (MI) procedure

Given Boolean expression E in DNF, the MI procedure produces a set of constraints S_E that guarantees the detection of missing or extra NOT (!) operator faults in the implementation of E.

The MI procedure is on page 193 and is illustrated next.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MI procedure: An Example

Consider the non-singular predicate: a(bc+!bd). Its DNF equivalent is:

E=abc+a!bd.

Note that a, b, c, and d are Boolean variables and also referred to as literals. Each literal represents a condition. For example, a could represent r<s.

Recall that + is the Boolean OR operator, ! is the Boolean NOT operator, and as per common convention we have omitted the Boolean AND operator. For example bc is the same as b∧c.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MI procedure: Example (contd.)

Step 0: Express E in DNF notation. Clearly, we can write E=e1+e2, where e1=abc and e2=a!bd.

Step 1: Construct a constraint set T_{e1} for e1 that makes e1 true. Similarly construct T_{e2} for e2 that makes e2 true.

T_{e1} ={(t,t,t,t), (t,t,t,f)} T_{e2} ={(t,f,t,t), (t,f,f,t)}

Note that the four t's in the first element of T_{e1} denote the values of the Boolean variables a, b,c, and d, respectively. The second element, and others, are to be interpreted similarly.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MI procedure: Example (contd.)

Step 2: From each T_{ei} , remove the constraints that are in any other T_{ej}. This gives us TS_{ei} and TS_{ej}. Note that this step will lead $TS_{ei} \cap TS_{ej} = \emptyset$.

There are no common constraints between T_{e1} and T_{e2} in our example. Hence we get:

$TS_{e1} = \{(t,t,t,t), (t,t,t,f)\}$ $TS_{e2} = \{(t,f,t,t), (t,f,f,t)\}$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MI procedure: Example (contd.)

Step 3: Construct S^t_E by selecting one element from each T_e.

$$S^t_E = \{(t,t,t,t), (t,f,f,f)\}$$

Note that for each constraint x in S^t_E we get E(x)=true. Also, S^t_E is minimal. *Check it out!*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MI procedure: Example (contd.)

Step 4: For each term in E, obtain terms by complementing each literal, one at a time.

e^1_1= !abc e^2_1= a!bc e^3_1= ab!c

e^1_2= !a!bd e^2_2= abd e^3_2= a!b!d

From each term e above, derive constraints F_e that make e true. We get the following six sets.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MI procedure: Example (contd.)

$Fe^1_1 = \{(f,t,t,t), (f,t,t,f)\}$

$Fe^2_1 = \{(t,f,t,t), (t,f,t,f)\}$

$Fe^3_1 = \{(t,t,f,t), (t,t,f,f)\}$

$Fe^1_2 = \{(f,f,t,t), (f,f,f,t)\}$

$Fe^2_2 = \{(t,t,t,t), (t,t,f,t)\}$

$Fe^3_2 = \{(t,f,t,f), (t,f,f,f)\}$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MI procedure: Example (contd.)

Step 5: Now construct FS_e by removing from F_e any constraint that appeared in any of the two sets T_e constructed earlier.

$FSe^1_1 = FSe^1_1$

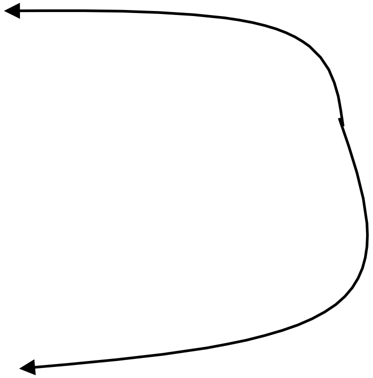
$FSe^2_1 = \{(t,f,t,f)\}$

$FSe^3_1 = FSe^1_3$

$FSe^1_2 = FSe^1_2$

$FSe^2_2 = \{(t,t,f,t)\}$

$FSe^3_2 = FSe^1_3$



Constraints common to T_{e1} and T_{e2} are removed.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MI procedure: Example (contd.)

Step 6: Now construct S^f_E by selecting one constraint from each F_e

S^f_E ={(f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)}

Step 7: Now construct $S_E = S^t_E \cup S^f_E$

S_E={{(t,t,t,t), (t,f,f,f), (f,t,t,f), (t,f,t,f), (t,t,f,t), (f,f,t,t)}

Note: Each constraint in S^t_E makes E true and each constraint in S^f_E makes E false.

Check it out!

We are now done with the MI procedure.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR-MI-CSET procedure

The BOR-MI-CSET procedure takes a non-singular expression E as input and generates a constraint set that guarantees the detection of Boolean operator faults in the implementation of E.

The BOR-MI-CSET procedure using the MI procedure described earlier.

The entire procedure is described on page 195. We illustrate it with an example.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BOR-MI-CSET: Example

Consider a non-singular Boolean expression: E= a(bc+!bd)

Mutually non-singular components of E:

e1=a

e2=bc+!bd

We use the BOR-CSET procedure to generate the constraint set for e1 (singular component) and MI-CSET procedure for e2 (non-singular component).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR-MI-CSET: Example (contd.)

For component e1 we get:

$$S^t_{e1}=\{t\}.\ S^f_{e1}=\{f\}$$

Recall that S^t_{e1} is true constraint set for e1 and S^f_{e1} is false constraint set for e1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR-MI-CSET: Example (contd.)

Component e2 is a DNF expression. We can write e2=u+v where u=bc and v=!bd.

Let us now apply the MI-CSET procedure to obtain the BOR constraint set for e2.

As per Step 1 of the MI-CSET procedure we obtain:

$$T_u=\{(t,t,t), (t,t,f)\} \quad T_v=\{(f,t,t), (f,f,t)\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR-MI-CSET: Example (contd.)

Applying Steps 2 and 3 to T_u and T_v we obtain:

$TS_u=T_u$ $TS_v=T_v$

$S^t_{e2}=\{(t,t,f), (f, t, t)\}$

Next we apply Step 4 to u and v. We obtain the following complemented expressions from u and v:

One possible alternative. Can you think of other alternatives?

u1=!bc u2=b!c

v1=bd v2=!b!d

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BOR-MI-CSET: Example (contd.)

Continuing with Step 4 we obtain:

F_{u1}={(f,t,t), (f,t,f)} F_{u2}=(t,f,t), (t,f,f)}

F_{v1}={(t,t,t), (t,f,t)} F_{v2}={(f,t,f), (f,f,f)}

Next we apply Step 5 to the F constraint sets to obtain:

FS_{u1}={(f,t,f)} FS_{u2}=(t,f,t), (t,f,f)}

FS_{v1}={(t,f,t)} FS_{v2}={(f,t,f), (f,f,f)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BOR-MI-CSET: Example (contd.)

Applying Step 6 to the FS sets leads to the following

S^{f}_{e2}={(f,t,f), (t,f,t)}.

Combing the true and false constraint sets for e2 we get:

S_{e2}={(t,t,f), (f, t, t), {(f,t,f), (t,f,t)}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

BOR-MI-CSET: Example (contd.)

Summary:

$S^t_{e1}=\{(t)\}$ $S^f_{e1}=\{(f)\}$ from BOR-CSET procedure.

$S^t_{e2}=\{(t,t,f), (f, t, t)\}$ $S^f_{e2}=\{(f,t,f), (t,f,t)\}$ from MI-CSET procedure.

We now apply Step 2 of the BOR-CSET procedure to obtain the constraint set for the entire expression E.

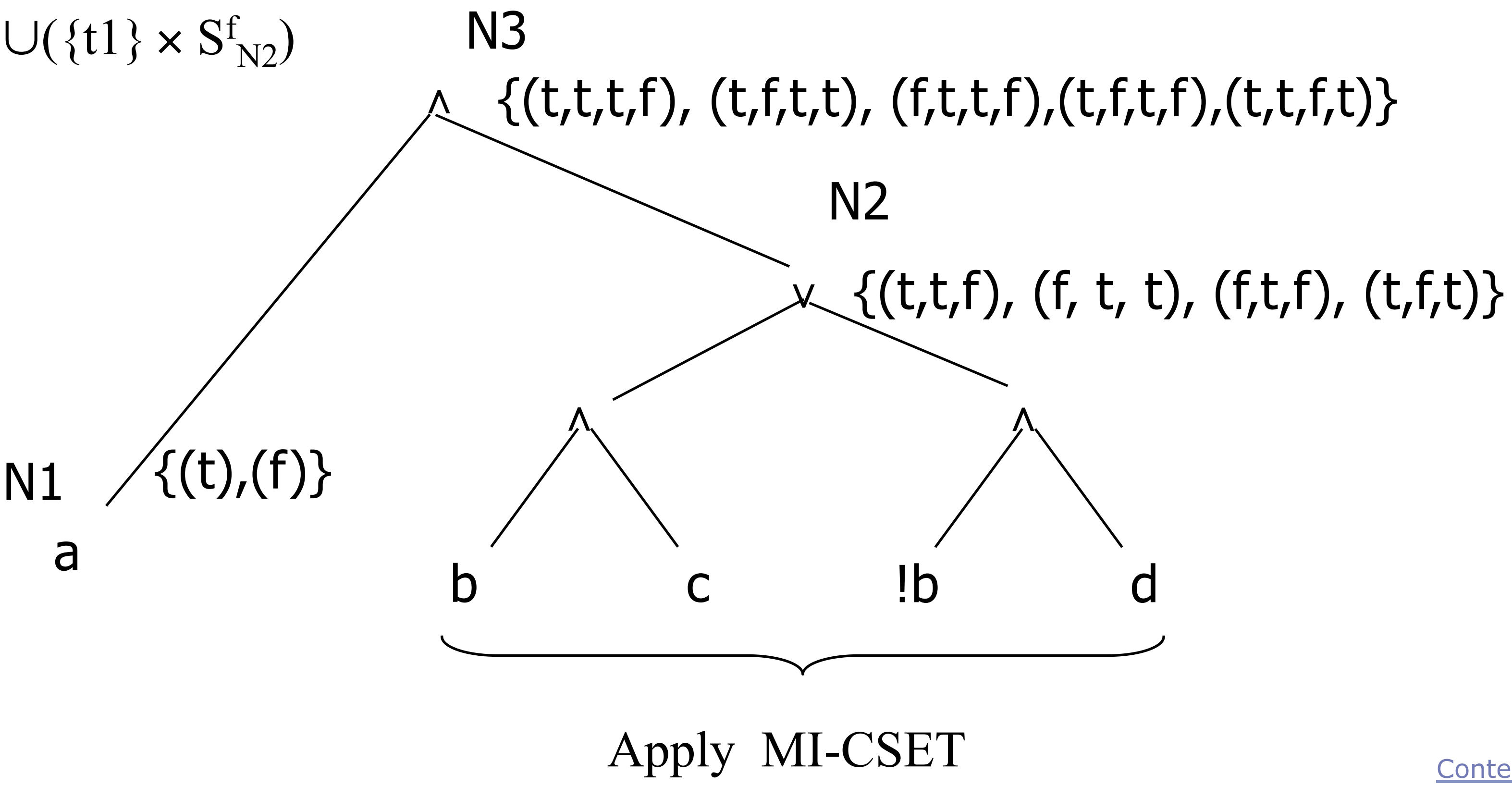
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

BOR-MI-CSET: Example (contd.)

$S^{t}_{N3}=S^{t}_{N1} \otimes S^{t}_{N22}$

$S^{f}_{N3}=(S^{f}_{N1} \times \{t2\})\cup(\{t1\} \times S^{f}_{N2})$

Obtained by applying Step 2 of BOR-CSET to an AND node.


N3
∧ {(t,t,t,f), (t,f,t,t), (f,t,t,f),(t,f,t,f),(t,t,f,t)}
N2
∨ {(t,t,f), (f, t, t), (f,t,f), (t,f,t)}
∧
∧
N1
{(t),(f)}
a
b
c
!b
d
Apply MI-CSET

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary

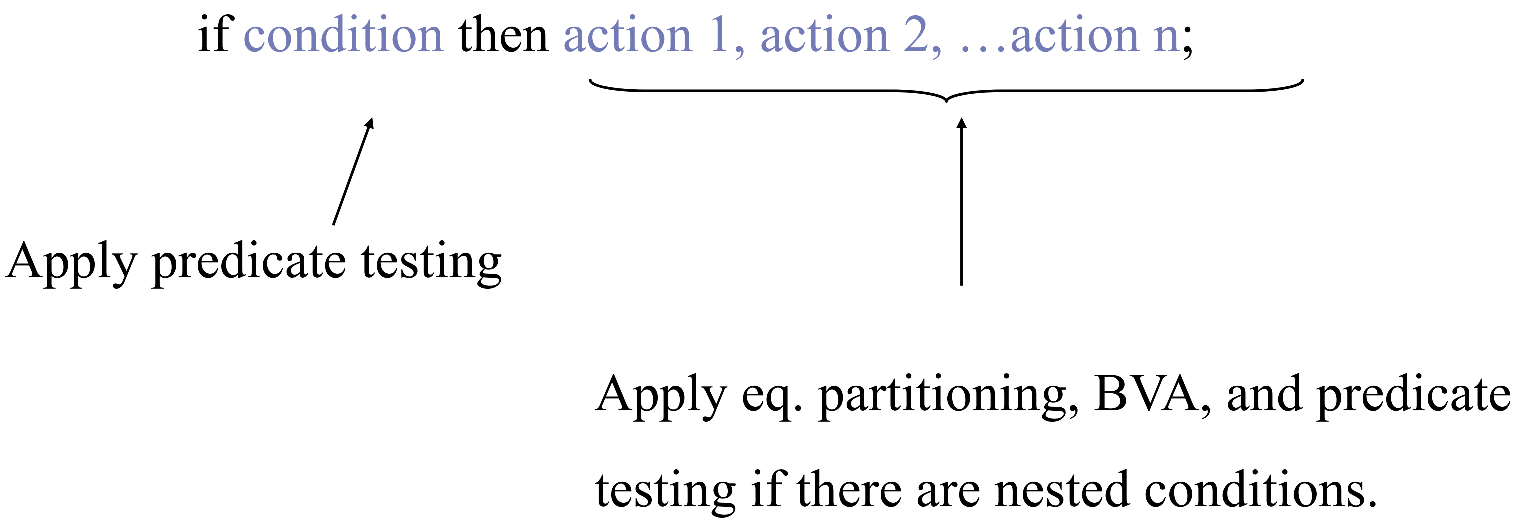
Most requirements contain conditions under which functions are to be executed. Predicate testing procedures covered are excellent means to generate tests to ensure that each condition is tested adequately.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary (contd.)

Usually one would combine equivalence partitioning, boundary value analysis, and predicate testing procedures to generate tests for a requirement of the following type:



if condition then action 1, action 2, …action n;

Apply predicate testing

Apply eq. partitioning, BVA, and predicate testing if there are nested conditions.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Chapter 5

Test Generation from Finite State Models

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 16, 2013

Contents

Learning Objectives

- What are Finite State Models?
- The W method for test generation
- The Wp method for test generation
- Automata theoretic versus control-flow based test generation

UIO method is not covered in these slides. It is left for the students to read on their own (Section 5.8).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Where are these methods used?

- Conformance testing of communications protocols--this is where it all started.
- Testing of any system/subsystem modeled as a finite state machine, e.g. elevator designs, automobile components (locks, transmission, stepper motors, etc), nuclear plant protection systems, steam boiler control, etc.)
- Finite state machines are widely used in modeling of all kinds of systems. Generation of tests from FSM specifications assists in testing the conformance of implementations to the corresponding FSM model.

Alert: It will be a mistake to assume that the test generation methods described here are applicable only to protocol testing!

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

5.2 Finite State Machines

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

What is a Finite State Machine?

A finite state machine, abbreviated as FSM, is an abstract representation of behavior exhibited by some systems.

An FSM is derived from application requirements. For example, a network protocol could be modeled using an FSM.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

What is a Finite State Machine?

Not all aspects of an application's requirements are specified by an FSM. Real time requirements, performance requirements, and several types of computational requirements cannot be specified by an FSM.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Requirements or design specification?

An FSM could serve any of two roles: as a specification of the required behavior and/or as a design artifact according to which an application is to be implemented.

The role assigned to an FSM depends on whether it is a part of the requirements specification or of the design specification.

Note that FSMs are a part of UML 2.0 design notation.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Where are FSMs used?

Modeling GUIs, network protocols, pacemakers, Teller machines, WEB applications, safety software modeling in nuclear plants, and many more.

While the FSM' s considered in examples are abstract machines, they are abstractions of many real-life machines.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM and statcharts

Note that FSMs are different from statecharts. While FSMs can be modeled using statecharts, the reverse is not true.

Techniques for generating tests from FSMs are different from those for generating tests from statecharts.

The term "state diagram" is often used to denote a graphical representation of an FSM or a statechart.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

FSM (Mealy machine): Formal definition

An FSM (Mealy) is a 6-tuple: (X, Y, Q, q_0, δ, O), where:,

X is a finite set of input symbols also known as the input alphabet.

Y is a finite set of output symbols also known as the output alphabet,

Q is a finite set states,

q_0 in Q is the initial state,

δ: Q x X→ Q is a next-state or state transition function, and

O: Q x X→ Y is an output function

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM (Moore machine): Formal definition

An FSM (Moore) is a 7-tuple: $(X, Y, Q, q_{0,} \delta, O, F)$, where:$_,$

X , Y, Q, q_0, and δ are the same as in FSM (Mealy)

O: $Q \rightarrow Y$ is an output function

$F \in Q$ is the set of final or accepting or terminating states.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

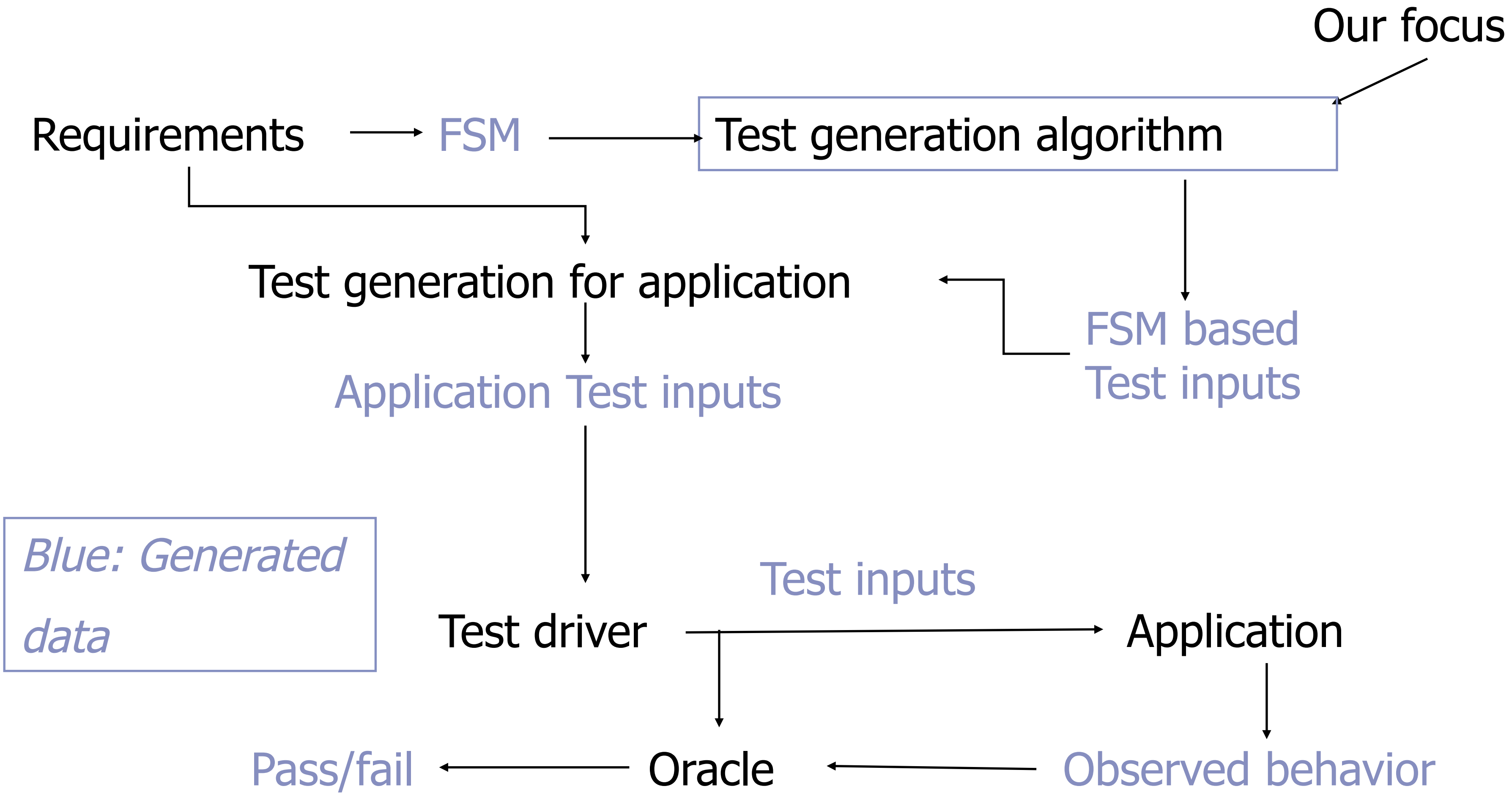
FSM: Formal definition (contd.)

Mealy machines are due to G. H. Mealy (1955 publication)

Moore machines are due to E. F. Moore (1956 publication)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test generation from FSMs



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Embedded systems

Many real-life devices have computers embedded in them. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses.

Such devices are also known as embedded systems. An embedded system can be as simple as a child's musical keyboard or as complex as the flight controller in an aircraft. In any case, an embedded system contains one or more computers for processing inputs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Specifying embedded systems

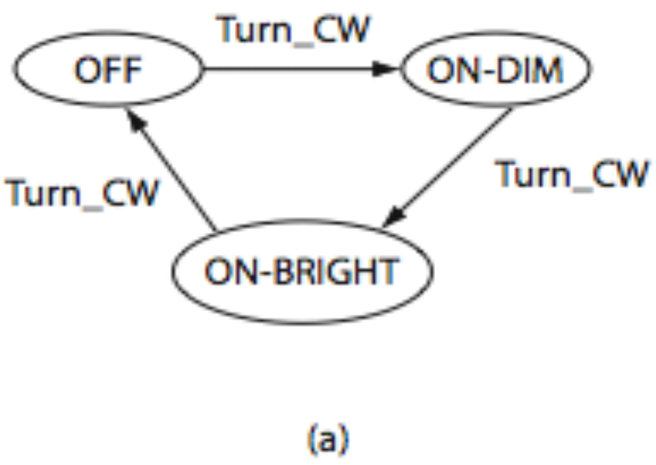
An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another.

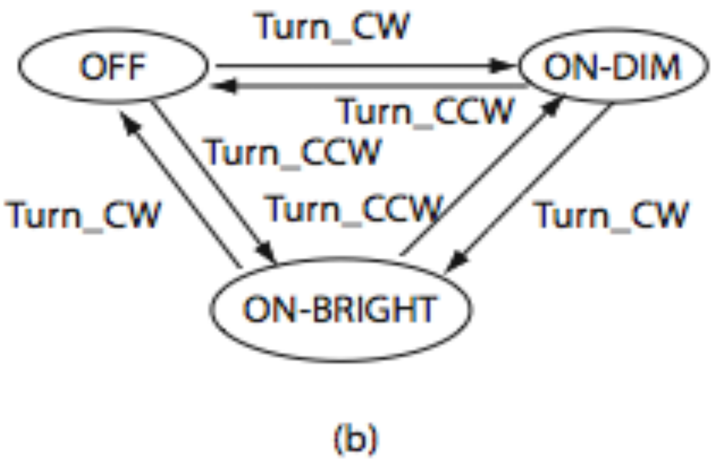
The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by a finite state machine (FSM).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM: lamp example

Simple three state lamp behavior:





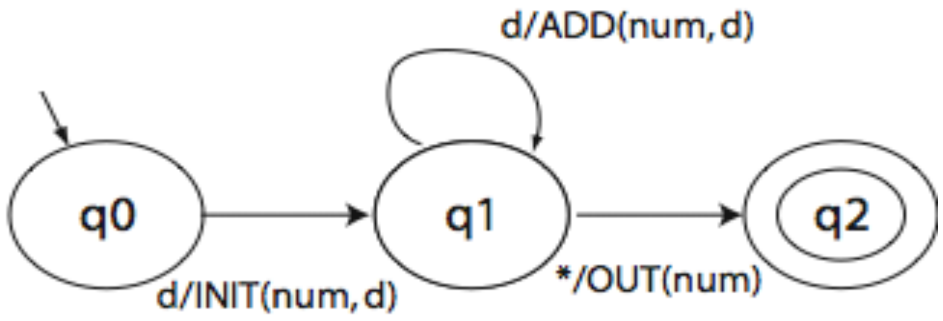
(a) Lamp switch can be turned clockwise.

(b) Lamp switch can be turned clockwise and counterclockwise.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM: Actions with state transitions

Machine to convert a sequence of decimal digits to an integer:



(a) Notice the ADD, INIT, ADD, and OUT actions.

(b) INIT: Initialize num. ADD: Add to num. OUT: Output num.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM: Formal definition

An FSM is a quintuple: (X, Y, Q, $q_{0,}$ δ, O), where:

X is a finite set of input symbols also known as the input alphabet.

Y is a finite set of output symbols also known as the output alphabet,

Q is a finite set states,

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

FSM: Formal definition (contd.)

q_0 in Q is the initial state,

δ: Q x X→ Q is a next-state or state transition function, and

O: Q x X→ Y is an output function.

In some variants of FSM more than one state could be specified as an initial state. Also, sometimes it is convenient to add $F \subseteq Q$ as a set of final or accepting states while specifying an FSM.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

State diagram representation of FSM

A state diagram is a directed graph that contains nodes representing states and edges representing state transitions and output functions.

Each node is labeled with the state it represents. Each directed edge in a state diagram connects two states. Each edge is labeled i/o where i denotes an input symbol that belongs to the input alphabet X and o denotes an output symbol that belongs to the output alphabet O. i is also known as the input portion of the edge and o its output portion.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

5.2.2 Tabular representation

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Tabular representation of FSM

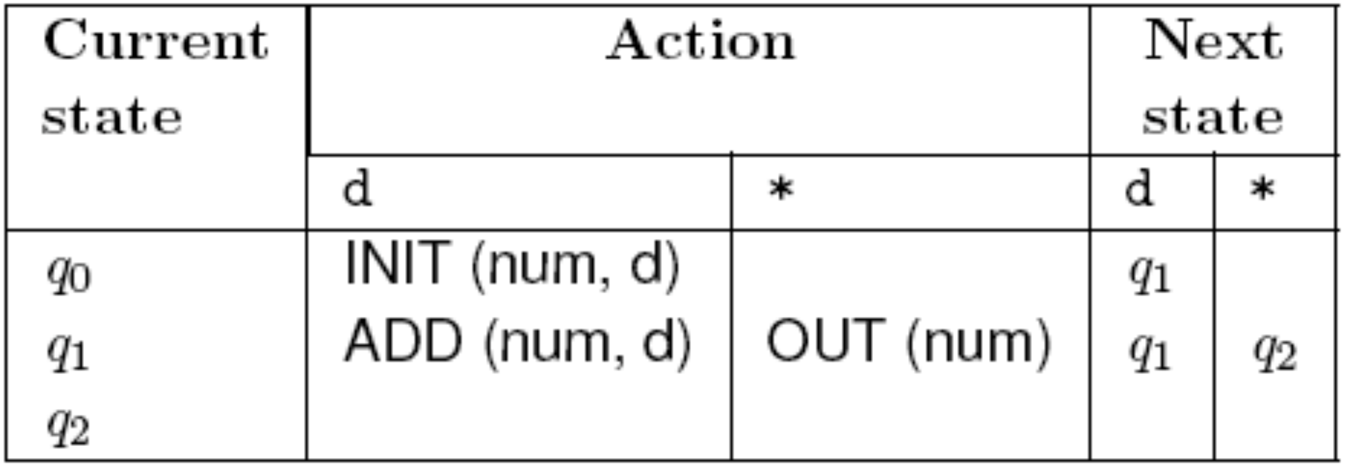
A table is often used as an alternative to the state diagram to represent the state transition function δ and the output function O.

The table consists of two sub-tables that consist of one or more columns each. The leftmost sub table is the output or the action sub-table. The rows are labeled by the states of the FSM. The rightmost sub-table is the next state sub-table.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Tabular representation of FSM: Example

The table given below shows how to represent functions δ and O for the DIGDEC machine.



Current state	Action		Next state	
	d	*	d	*
q_0	INIT (num, d)		q_1	
q_1	ADD (num, d)	OUT (num)	q_1	q_2
q_2				

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

5.2.3 Properties of FSM

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Properties of FSM

Completely specified: An FSM M is said to be completely specified if from each state in M there exists a transition for each input symbol.

Strongly connected: An FSM M is considered strongly connected if for each pair of states (q_i , q_j) there exists an input sequence that takes M from state q_i to state q_j.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Properties of FSM: Equivalence

V-equivalence: Let $M_1=(X, Y, Q_1, m^1_0, T_1, O_1)$ and $M_2=(X, Y, Q_2, m^2_0, T_2, O_2)$ be two FSMs. Let V denote a set of non-empty strings over the input alphabet X i.e. $V\subseteq X^+$.

Let q_i and q_j, $i\neq j$, be the states of machines M_1 and M_2, respectively. q_i and q_j are considered V-equivalent if $O_1(q_i, s)=O_2(q_j, s)$ for all s in V.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Properties of FSM: Distinguishable

Stated differently, states q_i and q_j are considered V-equivalent if M_1 and M_2 , when excited in states q_i and q_j, respectively, yield identical output sequences.

States q_i and q_j are said to be equivalent if $O_1(q_i, r)=O_2(q_j, r)$ for any set V. If q_i and q_j are not equivalent then they are said to be distinguishable. This definition of equivalence also applies to states within a machine. Thus, machines M_1 and M_2 could be the same machine.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Properties of FSM: Machine Equivalence

Machine equivalence: Machines M_1 and M_2 are said to be equivalent if (a) for each state σ in M1 there exists a state σ' in M_2 such that σ and σ' are equivalent and (b) for each state σ in M_2 there exists a state σ' in M_1 such that σ and σ' are equivalent.

Machines that are not equivalent are considered distinguishable.

Minimal machine: An FSM M is considered minimal if the number of states in M is less than or equal to any other FSM equivalent to M.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Properties of FSM: k-equivalence

k-equivalence: Let M_1=(X, Y, Q_1, $m^1{}_0$, T_1, O_1) and M_2=(X, Y, Q_2, $m^2{}_0$, T_2, O_2) be two FSMs.

States q_iε Q_1 and q_jε Q_2 are considered k-equivalent if, when excited by any input of length k, yield identical output sequences.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Properties of FSM: k-equivalence (contd.)

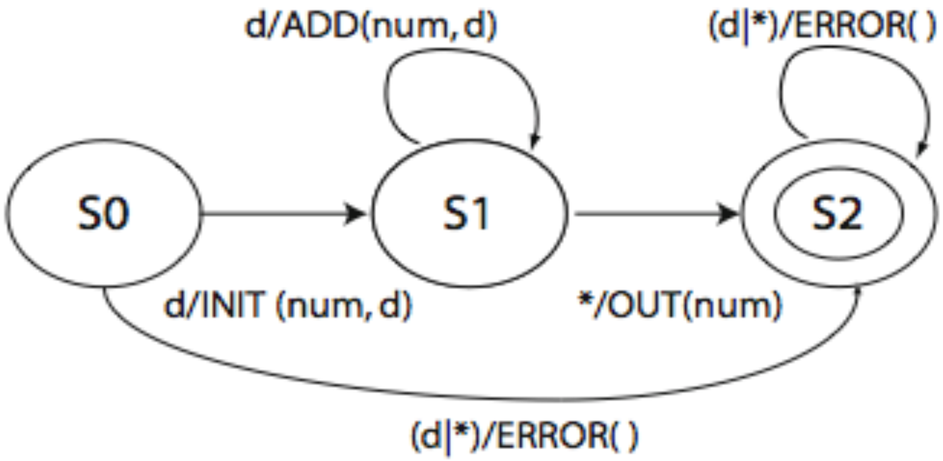
States that are not k-equivalent are considered k-distinguishable.

Once again, M_1 and M_2 may be the same machines implying that k-distinguishability applies to any pair of states of an FSM.

It is also easy to see that if two states are k-distinguishable for any k>0 then they are also distinguishable for any n≥ k. If M_1 and M_2 are not k-distinguishable then they are said to be k-equivalent.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: Completely specified machine



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

5.4 A fault model

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

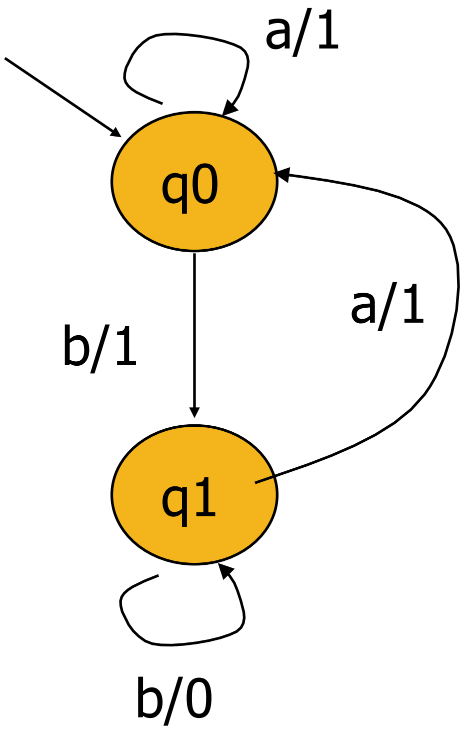
Faults in implementation

An FSM serves to specify the correct requirement or design of an application. Hence tests generated from an FSM target faults related to the FSM itself.

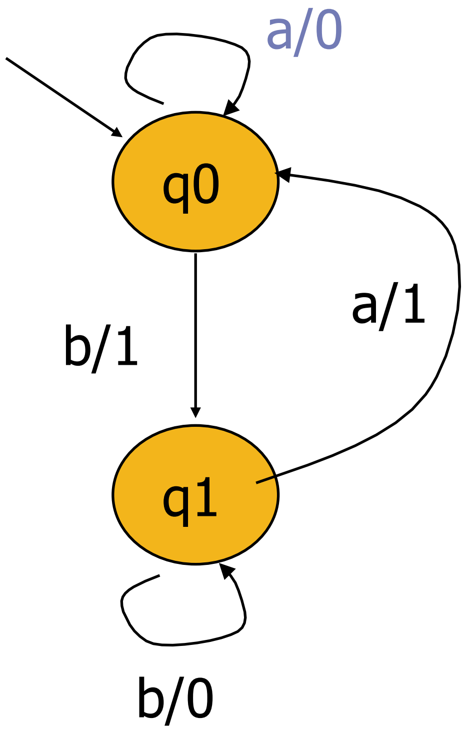
What faults are targeted by the tests generated using an FSM?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

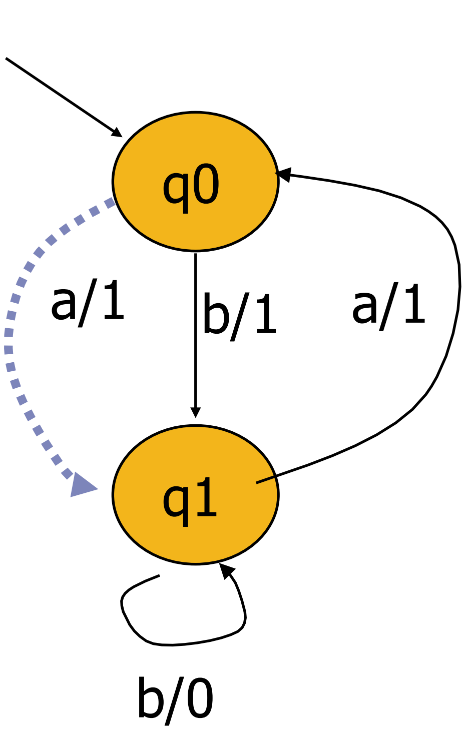
Fault model



Correct design



Operation error

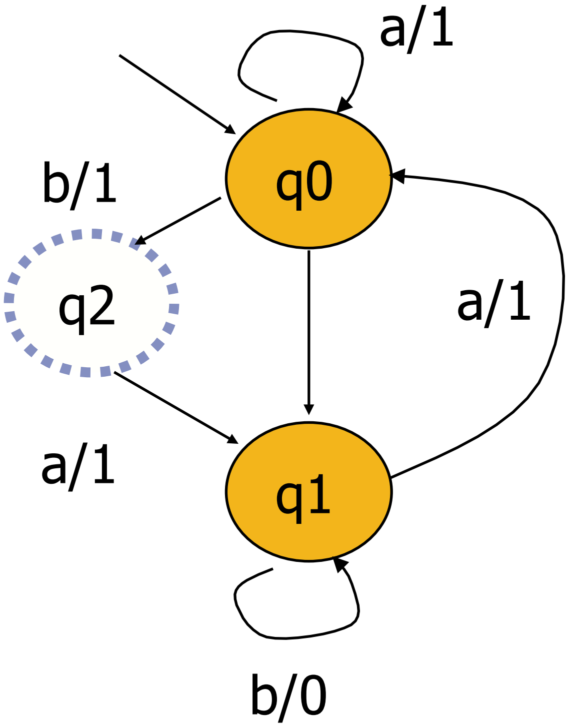


Transfer error

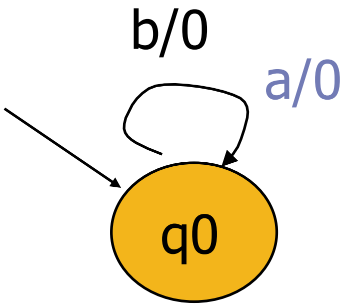
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Fault model (contd.)



Extra state error



Missing state error

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

5.5 Characterization set

5.6 The W-method

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Assumptions for test generation

Minimality: An FSM M is considered minimal if the number of states in M is less than or equal to any other FSM equivalent to M.

Completely specified: An FSM M is said to be completely specified if from each state in M there exists a transition for each input symbol.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Chow’s (W) method

Step 1: Estimate the maximum number of states (m) in the correct implementation of the given FSM M.

Step 2: Construct the characterization set W for M.

Step 3: (a) Construct the testing tree for M and (b) generate the transition cover set P from the testing tree.

Step 4: Construct set Z from W and m.

Step 5: Desired test set=P.Z

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Step 1: Estimation of m

This is based on a knowledge of the implementation. In the absence of any such knowledge, let m=|Q|.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Step 2: Construction of the W-set

Let $M=(X, Y, Q, q1, \delta, O)$ be a minimal and complete FSM.

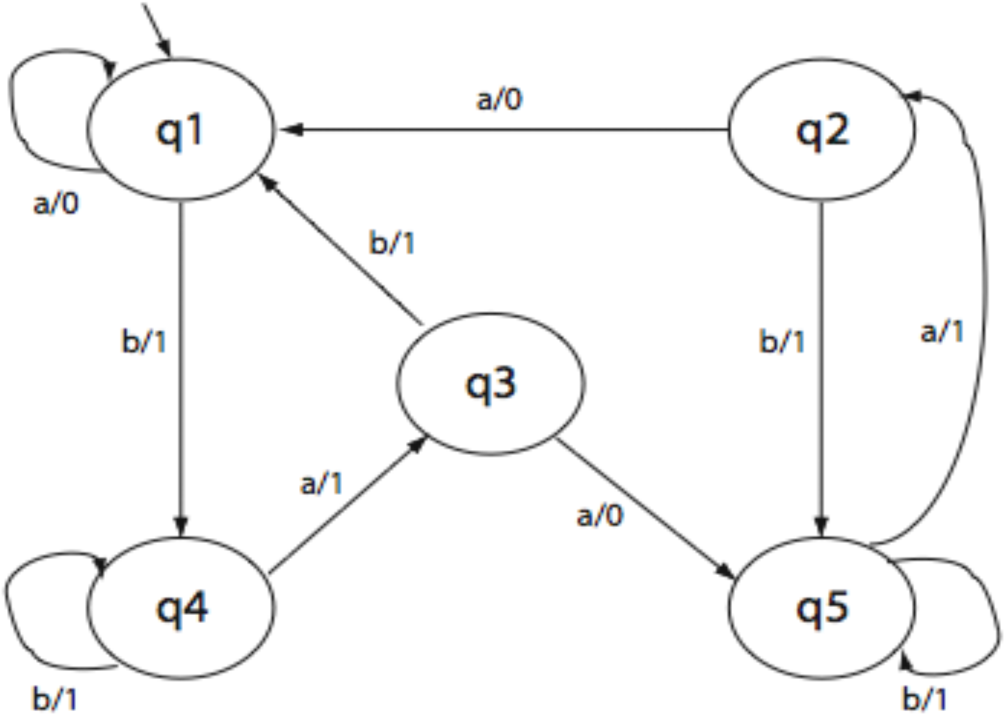
W is a finite set of input sequences that distinguish the behavior of any pair of states in M. Each input sequence in W is of finite length.

Given states qi and qj in Q, W contains a string s such that:

$$O(qi, s) \neq O(qj, s)$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example of a W-set



W={baaa,aa,aaa}

O(baaa,q1)=1101

O(baaa,q2)=1100

Thus, baaa distinguishes state q1 from q2 as O(baaa,q1) ≠ O(baaa,q2)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Steps in the construction of W-set

Step 1: Construct a sequence of k-equivalence partitions of Q denoted as P1, P2, …Pm, m>0.

Step 2: Traverse the k-equivalence partitions in reverse order to obtain distinguishing sequence for each pair of states.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

What is a k-equivalence partition of Q?

A k-equivalence partition of Q, denoted as P_k, is a collection of n finite sets Σ_{k1}, Σ_{k2} … Σ_{kn} such that

$\cup^n{}_{i=1}\ \Sigma_{ki} = Q$

States in Σ_{ki} are k-equivalent.

If state u is in Σ_{ki} and v in Σ_{kj} for $i \neq j$, then u and v are k-distinguishable.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

How to construct a k-equivalence partition?

Given an FSM M, construct a 1-equivalence partition, start with a tabular representation of M.

Current state	Output		Next state	
	a	b	a	b
q1	0	1	q1	q4
q2	0	1	q1	q5
q3	0	1	q5	q1
q4	1	1	q3	q4
q5	1	1	q2	q5

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construct 1-equivalence partition

Group states identical in their Output entries. This gives us 1-partition P_1 consisting of Σ_1={q1, q2, q3} and Σ_2 ={q4, q5}.

Σ	Current state	Output		Next state	
		a	b	a	b
1	q1	0	1	q1	q4
	q2	0	1	q1	q5
	q3	0	1	q5	q1
2	q4	1	1	q3	q4
	q5	1	1	q2	q5

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construct 2-equivalence partition: Rewrite P_1 table

Rewrite P_1 table. Remove the output columns. Replace a state entry q_i by q_{ij} where j is the group number in which lies state q_i.

Σ	Current state	Next state	
		a	b
1	q1	q11	q42
	q2	q11	q52
	q3	q52	q11
2	q4	q31	q42
	q5	q21	q52

P_1 Table

Group number

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construct 2-equivalence partition: Construct P_2 table

Group all entries with identical second subscripts under the next state column. This gives us the P_2 table. Note the change in second subscripts.

Σ	Current state	Next state	
		a	b
1	q1	q11	q43
	q2	q11	q53
2	q3	q53	q11
3	q4	q32	q43
	q5	q21	q53

P_2 Table

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construct 3-equivalence partition: Construct P_3 table

Group all entries with identical second subscripts under the next state column. This gives us the P_3 table. Note the change in second subscripts.

Σ	Current state	Next state	
		a	b
1	q1	q11	q43
	q2	q11	q54
2	q3	q54	q11
3	q4	q32	q43
4	q5	q21	q54

P_3 Table

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Construct 4-equivalence partition: Construct P_4 table

Continuing with regrouping and relabeling, we finally arrive at P_4 table.

P_4 Table

Σ	Current state	Next state	
		a	b
1	q1	q11	q44
2	q2	q11	q55
3	q3	q55	q11
4	q4	q33	q44
5	q5	q22	q55

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

k-equivalence partition: Convergence

The process is guaranteed to converge.

When the process converges, and the machine is minimal, each state will be in a separate group.

The next step is to obtain the distinguishing strings for each state.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Finding distinguishing sequences: Example

Let us find a distinguishing sequence for states q1 and q2.

Find tables P_i and P_{i+1} such that (q1, q2) are in the same group in P_i and different groups in P_{i+1}. We get P_3 and P_4.

Initialize z=ε. Find the input symbol that distinguishes q1 and q2 in table P3. This symbol is b. We update z to z.b. Hence z now becomes b.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Finding the distinguishing sequences: Example (contd.)

The next states for q1 and q2 on b are, respectively, q4 and q5.

We move to the P_2 table and find the input symbol that distinguishes q4 and q5. Let us select a as the distinguishing symbol. Update z which now becomes ba.

The next states for states q4 and q5 on symbol a are, respectively, q3 and q2. These two states are distinguished in P_1 by a and b. Let us select a. We update z to baa.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Finding the distinguishing sequences: Example (contd.)

The next states for q3 and q2 on a are, respectively, q1 and q5.

Moving to the original state transition table we obtain a as the distinguishing symbol for q1 and q5

We update z to baaa. This is the farthest we can go backwards through the various tables. baaa is the desired distinguishing sequence for states q1 and q2. Check that o(q1,baaa)≠o(q2,baaa).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Finding the distinguishing sequences: Example (contd.)

Using the procedure analogous to the one used for q1 and q2, we can find the distinguishing sequence for each pair of states. This leads us to the following characterization set for our FSM.

W={a, aa, aaa, baaa}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chow’ s method: where are we?

Step 1: Estimate the maximum number of states (m) in the correct implementation of the given FSM M. ← Done

Step 2: Construct the characterization set W for M.

Step 3: (a) Construct the testing tree for M and (b) generate the transition cover set P from the testing tree. ← Next (a)

Step 4: Construct set Z from W and m.

Step 5: Desired test set=P.Z

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Step 3: (a) Construct the testing tree for M

A testing tree of an FSM is a tree rooted at the initial state. It contains at least one path from the initial state to the remaining states in the FSM. Here is how we construct the testing tree.

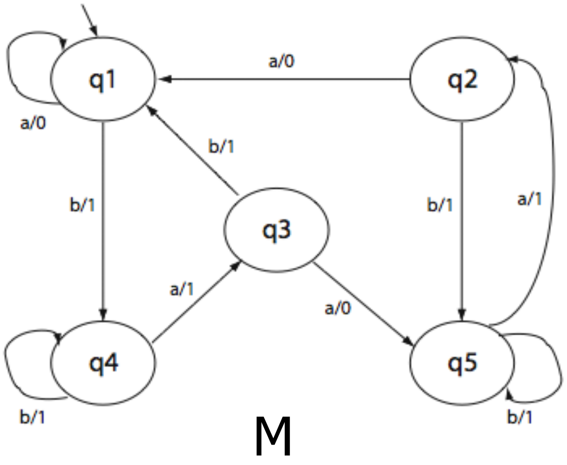
State q0, the initial state, is the root of the testing tree. Suppose that the testing tree has been constructed until level k . The (k+1)th level is built as follows.

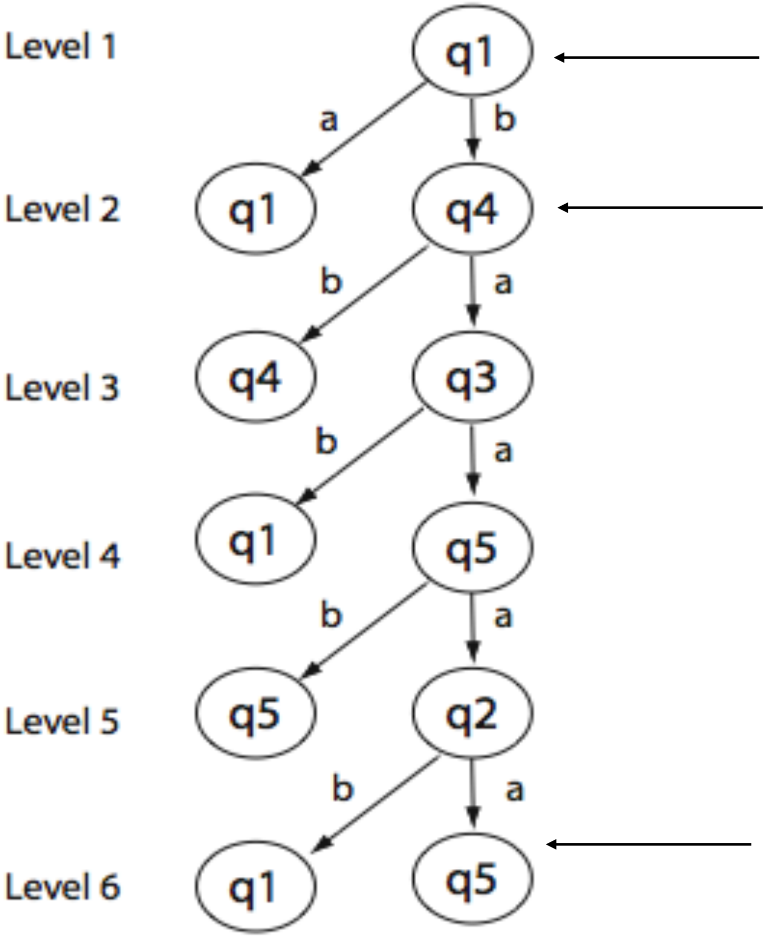
Select a node n at level k. If n appears at any level from 1 through k , then n is a leaf node and is not expanded any further. If n is not a leaf node then we expand it by adding a branch from node n to a new node m if $\delta(n, x)=m$ for $x \in X$. This branch is labeled as x. This step is repeated for all nodes at level k.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: Construct the testing tree for M





Start here, initial state is the root.

q1 becomes leaf, q4 can be expanded.

.

.

No further expansion possible

.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Chow's method: where are we?

Step 1: Estimate the maximum number of states (m) in the correct implementation of the given FSM M. ← Done

Step 2: Construct the characterization set W for M.

Step 3: (a) Construct the testing tree for M and (b) generate the transition cover set P from the testing tree. ← Next, (b)

Step 4: Construct set Z from W and m.

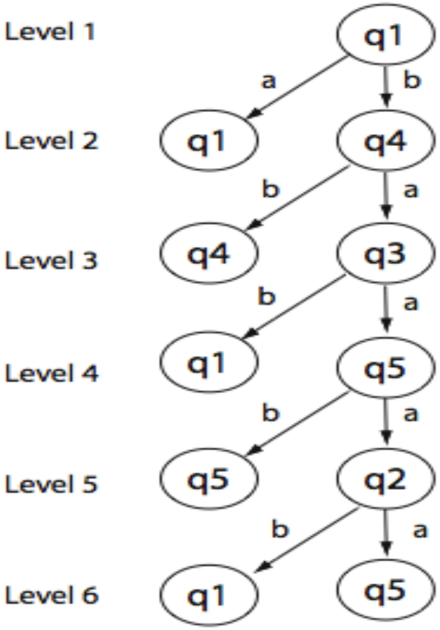
Step 5: Desired test set=P.Z

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Step 3: (b) Find the transition cover set from the testing tree

A transition cover set P is a set of all strings representing sub-paths, starting at the root, in the testing tree. Concatenation of the labels along the edges of a sub-path is a string that belongs to P. The empty string (ε) also belongs to P.



P={ε, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chow’ s method: where are we?

Step 1: Estimate the maximum number of states (m) in the correct implementation of the given FSM M. ← Done

Step 2: Construct the characterization set W for M.

Step 3: (a) Construct the testing tree for M and (b) generate the transition cover set P from the testing tree. ← Done

Step 4: Construct set Z from W and m. ← Next

Step 5: Desired test set=P.Z

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Step 4: Construct set Z from W and m

Given that X is the input alphabet and W the characterization set, we have:

$$Z = X^0.W \cup X^1.W \cup \ldots.. X^{m-1-n}.W \cup X^{m-n}.W$$

For m=n, we get

$$Z = X^0.W=W$$

For X={a, b}, W={a, aa, aaa, baaa}, m=6

$$Z = W \cup X^1.W =\{a, aa, aaa, baaa\} \cup \{a, b\}.\{a, aa, aaa, baaa\}$$

$$=\{a, aa, aaa, baaa, aa, aaa, aaaa, baaaa, ba, baa, baaa, bbaaa\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chow's method: where are we?

Step 1: Estimate the maximum number of states (m) in the correct implementation of the given FSM M. ← Done

Step 2: Construct the characterization set W for M.

Step 3: (a) Construct the testing tree for M and (b) generate the transition cover set P from the testing tree. ← Done

Step 4: Construct set Z from W and m. ← Done

Step 5: Desired test set=P.Z ← Next

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Step 5: Desired test set=P.Z

The test inputs based on the given FSM M can now be derived as:

T=P.Z

Do the following to test the implementation:

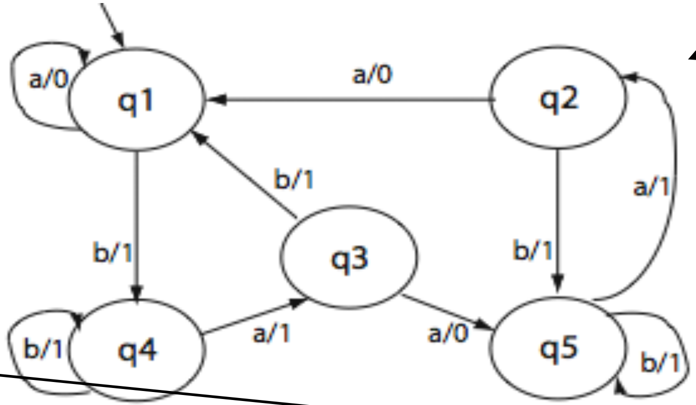
1. Find the expected response to each element of T.
2. Generate test cases for the application. Note that even though the application is modeled by M, there might be variables to be set before it can be exercised with elements of T.
3. Execute the application and check if the response matches. Reset the application to the initial state after each test.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example 1: Testing an erroneous application

Correct design

M



(a) Specification

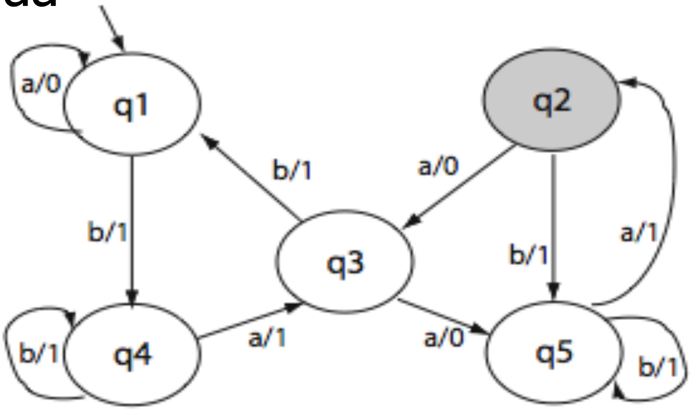
M(t1)=1101001

M(t2)=11011

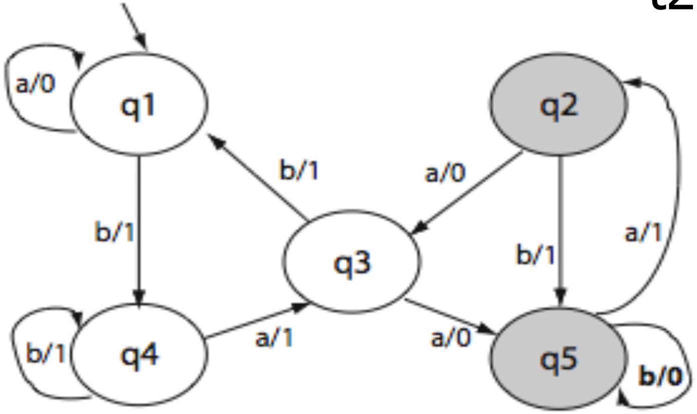
Error revealing test cases

t1=baaaaaa

t2=baaba



(b) Transfer error in state q2.



(c) Transfer error in state q2 and operation error in state q5.

M1(t1)=1101001

M2(t2)=11001

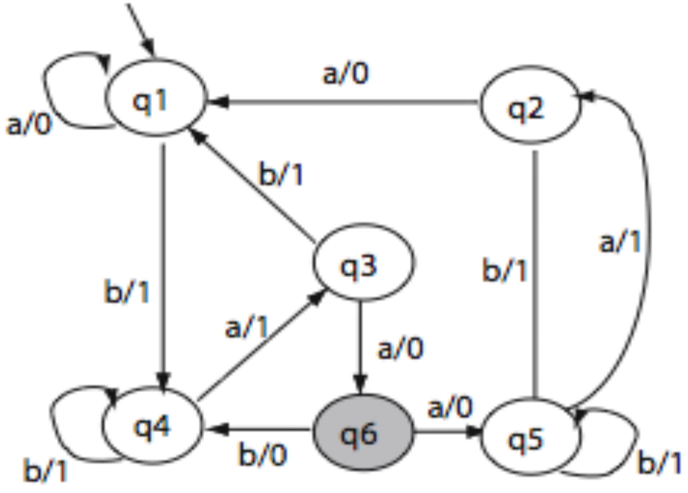
M1

M2

Contents

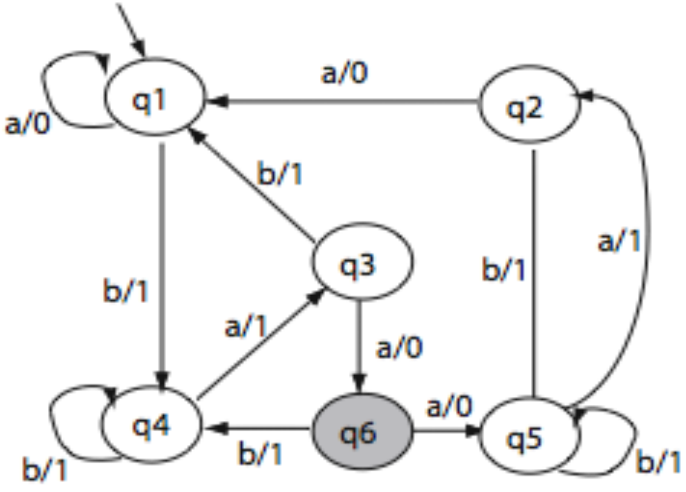
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example 2: Extra state. N=5, m=6.



(a)

M1



(b)

M2

t1=baaba M(t1)=11011 M1(t1)=11001

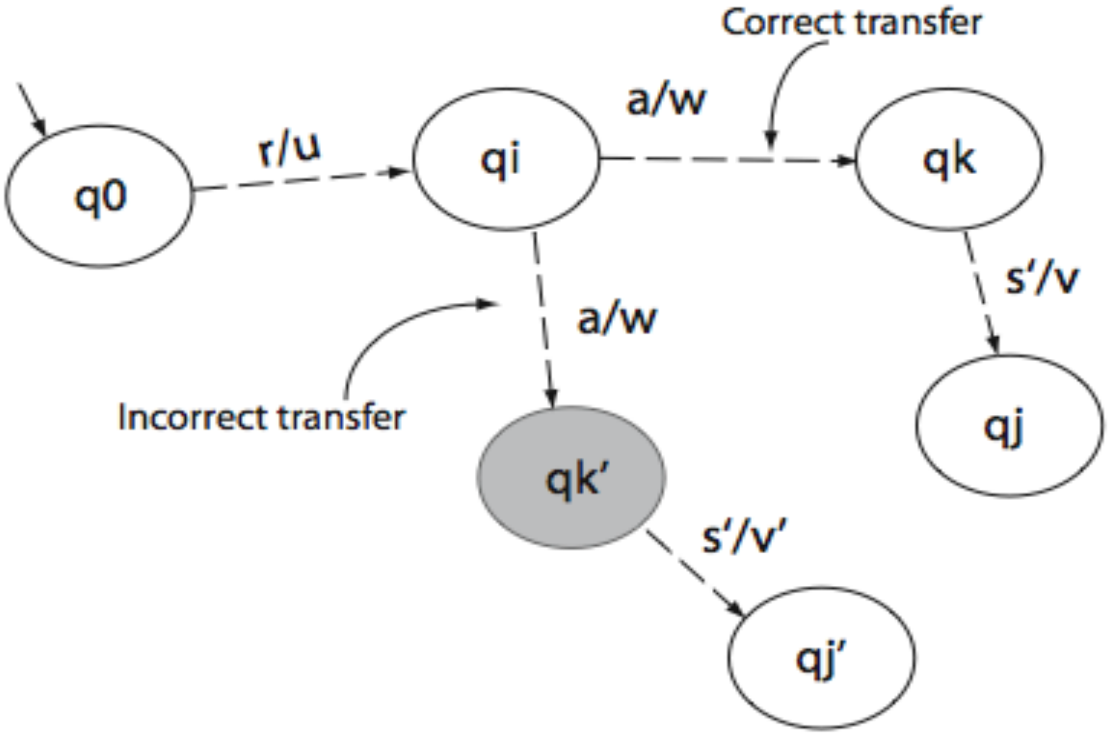
t2=baaa M(t2)=1101 M2(t2)=1100

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Error detection process: in-class discussion

Given m=n, each test case t is of the form r.s where r is in P and s in W. r moves the application from initial state q0 to state qj. Then, s=as' takes it from qi to state qj or qj'.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

5.7 The Partial W method

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

The partial W (Wp) method

Tests are generated from minimal, complete, and connected FSM.

Size of tests generated is generally smaller than that generated using the W-method.

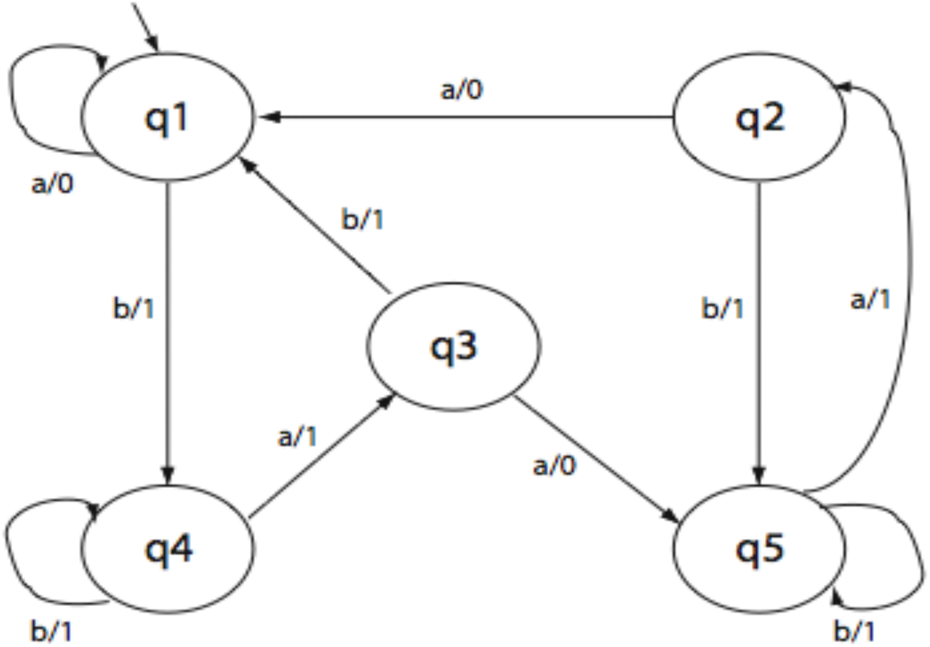
Test generation process is divided into two phases: Phase 1: Generate a test set using the state cover set (S) and the characterization set (W). Phase 2: Generate additional tests using a subset of the transition cover set and state identification sets.

What is a state cover set? A state identification set?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

State cover set

Given FSM M with input alphabet X, a state cover set S is a finite non-empty set of strings over X* such that for each state qi in Q, there is a string in S that takes M from its initial state to qi.



S={ε, b, ba, baa, baaa}

S is always a subset of the transition cover set P. Also, S is not necessarily unique.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

State identification set

Given an FSM M with Q as the set of states, an identification set for state qi∈Q is denoted by W_i and has the following properties:

(a) $W_i \subseteq W$, $1 \leq i \leq n$ [Identification set is a subset of W.]

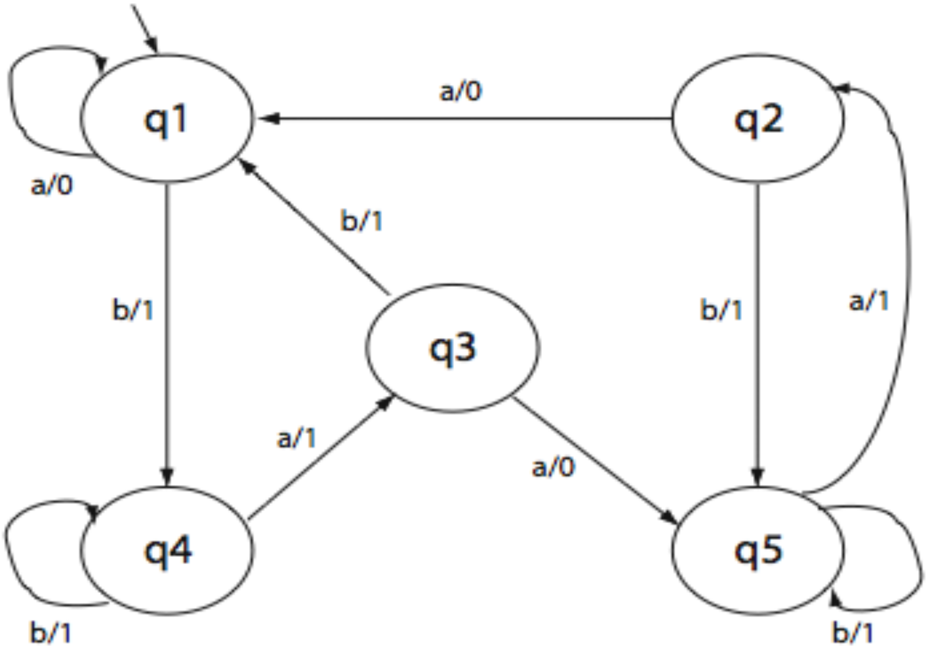
(b) $O(qi, s) \neq O(qj, s)$, for $1 \leq j \leq n$, $j \neq i$, $s \in W_i$ [For each state other than qi, there is a string in Wi that distinguishes qi from qj.]

(c) No subset of W_i satisfies property (b). [W_i is minimal.]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

State identification set: Example

Last element of the output string



Si	Sj	X	o(Si,x)	o(Sj,x)
1	2	baaa	1	0
	3	aa	0	1
	4	a	0	1
	5	a	0	1
2	3	aa	0	1
	4	a	0	1
	5	a	0	1
3	4	a	0	1
	5	a	0	1
4	5	aaa	1	0

$W_1=W_2=\{\text{baaa, aa, a}\}$

$W_3=\{\text{a aa}\}$ $W_4=W_5=\{\text{a, aaa}\}$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Wp method: Example: Step 1: Compute S, P, W, Wi, $\mathcal{W}$

S={ε, b, ba, baa, baaa}

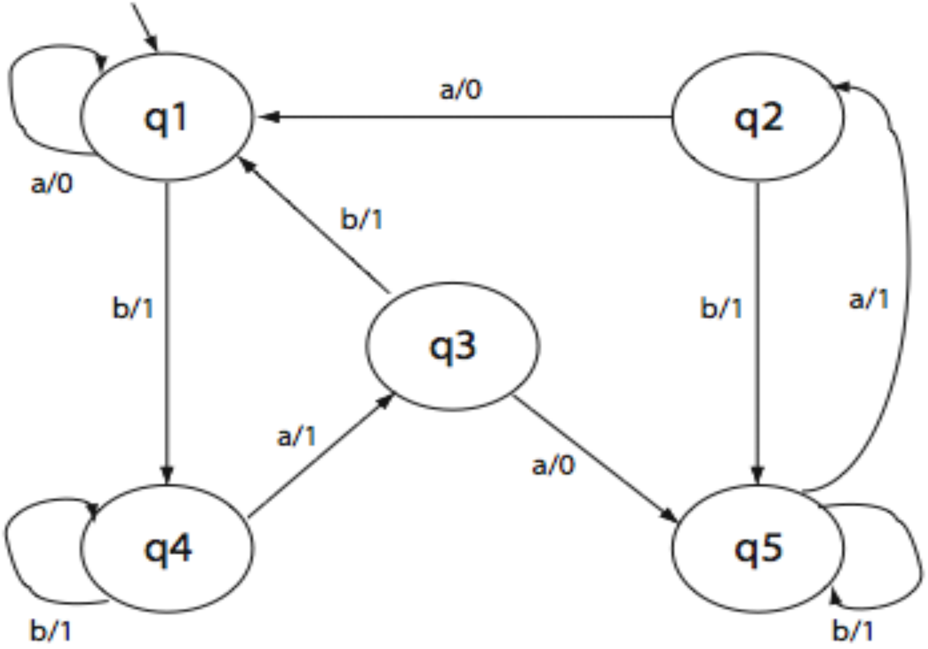
P={ε, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa}

$W_1=W_2$={baaa, aa, a}

W_3={a aa} $W_4=W_5$={a, aaa}

W={a, aa, aaa, baaa}

$\mathcal{W}$={W1, W2, W3, W4, W5}



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Wp method: Example: Step 2: Compute T1 [m=n]

T1=S. W={ε, b, ba, baa, baaa}.{a, aa, aaa, baaa}

Elements of T1 ensure that the each state of the FSM is covered and distinguished from the remaining states.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Wp method: Example: Step 3: Compute R and δ [m=n]

$R=P-S=\{\varepsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}-\{\varepsilon, b, ba, baa, baaa\}$

$=\{a, bb, bab, baab, baaab, baaaa\}$

Let each element of R be denoted as $r_{i1}, r_{i2},\ldots r_{ik}$.

$\delta(r_{ik}, m)=q_{ij}$, where $m \in X$ (the alphabet)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Wp method: Example: Step 4: Compute T2 [m=n]

$T2=R\otimes W=\cup^{k}_{(j=1)}(r_{ij}\}.\ W_{ij}$, where W_{ij} is the identification set for state q_{ij}.

$\delta(q1, a)=q1$ $\delta(q1, bb)=q4$ $\delta(q1, bab)=q5$

$\delta(q1, baab)=q5$ $\delta(q1, baaab)=q5$ $\delta(q1, baaaa)=q1$

$T2=(\{a\}.\ W_1)\cup(\{bb\}.W_4)\cup(\{bab\}.W_5)\cup(\{baab\}.W_5)\cup$
$\{baaab\}.W_5)\cup(\{baaaa\}.\ W_1)$

$=\{abaaa, aaa, aa\}\cup\{bba, bbaaa\}\cup\{baba, babaaa\}\cup$

$\{baaba, baabaaa\}\cup\{baaaba, baaabaaa\}\cup\{baaaabaaa, baaaaaa, baaaaa\}$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Wp method: Example: Savings

Test set size using the W method= 44

Test set size using the Wp method= 34 (20 from T1+14 from T2)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Testing using the Wp method

Testing proceeds in two phases.

Tests from T1 are applied in phase 1. Tests from T2 are applied in phase 2.

While tests from phase 1 ensure state coverage, they do not ensure all transition coverage. Also, even when tests from phase cover all transitions, they do not apply the state identification sets and hence not all transfer errors are guaranteed to be revealed by these tests.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Wp method:

Both sets T1 and T2 are computed a bit differently, as follows:

T1=S. X[m-n], where X[m-n] is the set union of X^i , $1 \leq i \leq (m-n)$

T2= T2=R. X[m-n] $\otimes$W

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

5.8 The UIO sequence method [See the text]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

5.9 Automata theoretic versus control flow based techniques

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Automata-theoretic vs. Control theoretic techniques

The W and the Wp methods are considered automata-theoretic methods for test generation.

In contrast, many books on software testing mention control-theoretic techniques for test generation. Let us understand the difference between the two types of techniques and their fault detection abilities.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Control theoretic techniques

State cover: A test set T is considered adequate with respect to the state cover criterion for an FSM M if the execution of M against each element of T causes each state in M to be visited at least once.

Transition cover: A test set T is considered adequate with respect to the branch/ transition cover criterion for an FSM M if the execution of M against each element of T causes each transition in M to be taken at least once

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Control theoretic techniques (contd.)

Switch cover: A test set T is considered adequate with respect to the 1-switch cover criterion for an FSM M if the execution of M against each element of T causes each pair of transitions (tr1, tr2) in M to be taken at least once, where for some input substring ab tr1: qi=δ(qj, a) and tr_2: qk= δ(qi, b) and qi, qj, qk are states in M.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Control theoretic techniques (contd.)

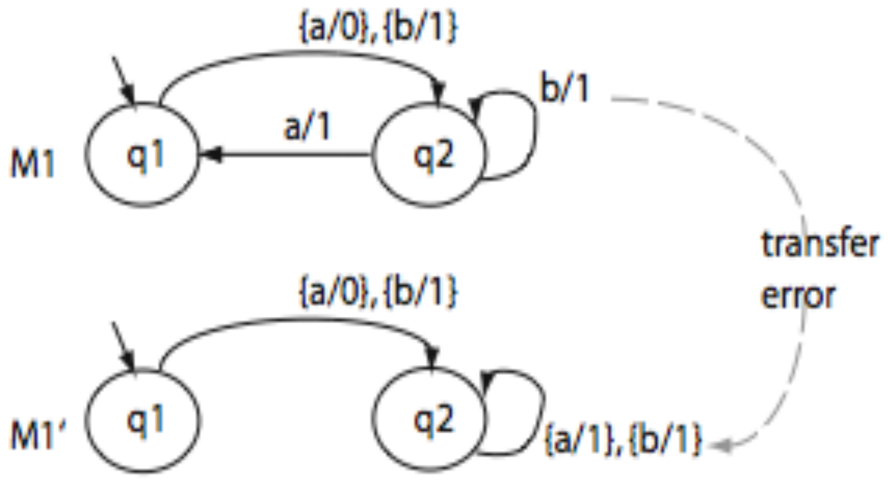
Boundary interior cover: A test set T is considered adequate with respect to the boundary-interior cover criterion for an FSM M if the execution of M against each element of T causes each loop (a self-transition) across states to be traversed zero times and at least once. Exiting the loop upon arrival covers the ``boundary'' condition and entering it and traversing the loop at least once covers the ``interior'' condition.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Control theoretic technique: Example 1

Consider the following machines, a correct one (M1) and one with a transfer error (M1').



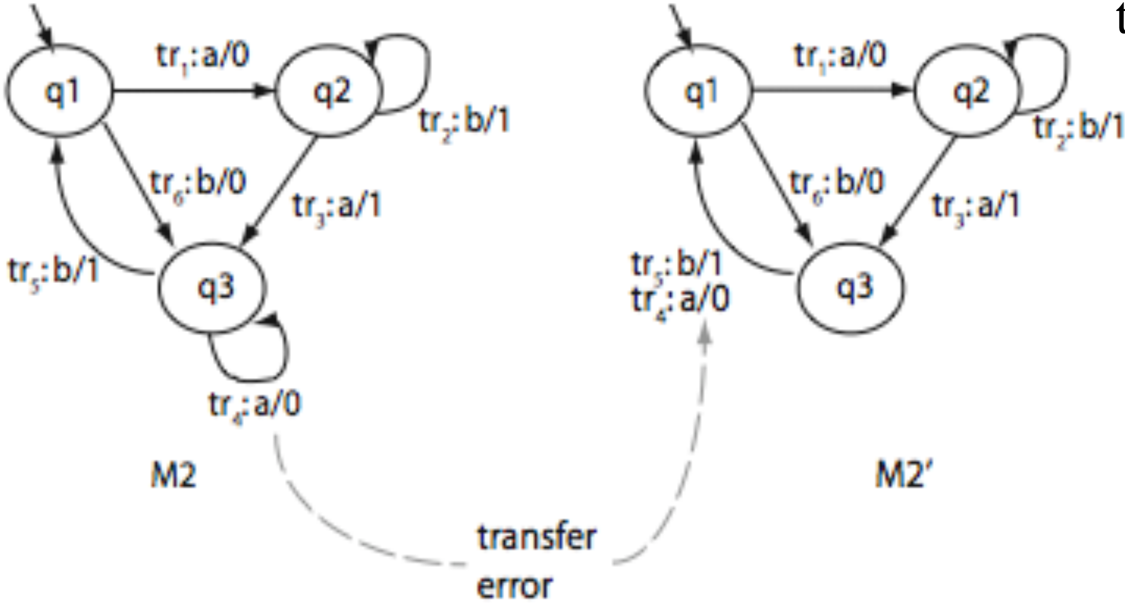
t=abba covers all states but does not not reveal the error. Both machines generate the same output which is 0111.

Will the tests generated by the W method reveal this error? Check it out!

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Control theoretic technique: Example 2

Consider the following machines, a correct one (M2) and one with a transfer error (M2').



There are 12 branch pairs, such as (tr1, tr2), (tr1, tr3), tr6, tr5).

Consider the test set: {bb, baab, aabb, aaba, abbaab}. Does it cover all branches? Does it reveal the error?

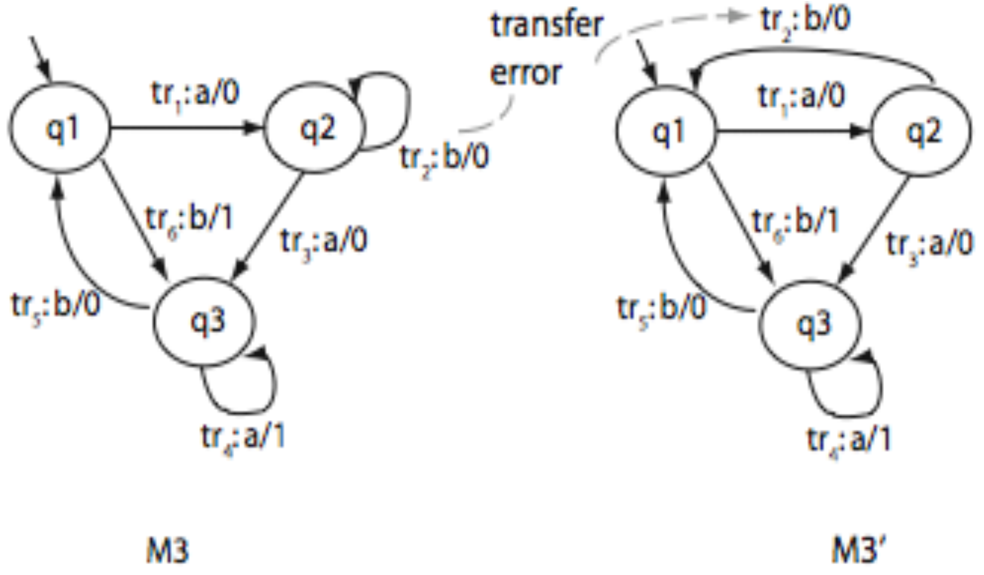
Are the states in M2 1-distinguishable?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Control theoretic technique: Example 3

Consider the following machines, a correct one (M3) and one with a transfer error (M3’).



Consider T={t1: aab, t2: abaab}. T1 causes each state to be entered but loop not traversed. T2 causes each loop to be traversed once.

Is the error revealed by T?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary

Behavior of a large variety of applications can be modeled using finite state machines (FSM). GUIs can also be modeled using FSMs

The W and the Wp methods are automata theoretic methods to generate tests from a given FSM model.

Tests so generated are guaranteed to detect all operation errors, transfer errors, and missing/extra state errors in the implementation given that the FSM representing the implementation is complete, connected, and minimal. *What happens if it is not?*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary (contd.)

Automata theoretic techniques generate tests superior in their fault detection ability than their control-theoretic counterparts.

Control-theoretic techniques, that are often described in books on software testing, include branch cover, state cover, boundary-interior, and n-switch cover.

The size of tests sets generated by the W method is larger than generated by the Wp method while their fault detection effectiveness are the same.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 6

Test Generation: Combinatorial Designs

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 16, 2013

Contents

Learning Objectives

- What are test configurations? How do they differ from test sets?
- Why combinatorial design?
- What are Latin squares and mutually orthogonal Latin squares (MOLS)?
- How does one generate test configurations from MOLS?
- What are orthogonal arrays, covering arrays and mixed-level covering arrays?
- How to generate mixed-level covering arrays and test configurations from them?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.1.1. Test configuration and test set

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test configuration

- Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments.

- An environment is characterized by combination of hardware and software.

- Each environment corresponds to a given set of values for each factor, known as a test configuration.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test configuration: Example

- Windows XP, Dial-up connection, and a PC with 512MB of main memory, is one possible configuration.
- Different versions of operating systems and printer drivers, can be combined to create several test configurations for a printer.
- To ensure high reliability across the intended environments, the application must be tested under as many test configurations, or environments, as possible.

The number of such test configurations could be exorbitantly large making it impossible to test the application exhaustively.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test configuration and test set

- While a test configuration is a combination of factors corresponding to hardware and software within which an application is to operate, a test set is a collection of test cases. Each test case consists of input values and expected output.

- Techniques we shall learn are useful in deriving test configurations as well as test sets.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Motivation

- While testing a program with one or more input variables, each test run of a program often requires at least one value for each variable.

- For example, a program to find the greatest common divisor of two integers x and y requires two values, one corresponding to x and the other to y.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Motivation [2]

While equivalence partitioning discussed earlier offers a set of guidelines to design test cases, it suffers from two shortcomings:

(a) It raises the possibility of a large number of sub-domains in the partition.

(b) It lacks guidelines on how to select inputs from various sub-domains in the partition.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Motivation [3]

The number of sub-domains in a partition of the input domain increases in direct proportion to the number and type of input variables, and especially so when multidimensional partitioning is used.

Once a partition is determined, one selects at random a value from each of the sub-domains. Such a selection procedure, especially when using uni-dimensional equivalence partitioning, does not account for the possibility of faults in the program under test that arise due to specific interactions amongst values of different input variables.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Motivation [4]

While boundary values analysis leads to the selection of test cases that test a program at the boundaries of the input domain, other interactions in the input domain might remain untested.

We will learn several techniques for generating test configurations or test sets that are small even when the set of possible configurations or the input domain and the number of sub-domains in its partition, is large and complex.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.1.2. Modeling the input and configuration spaces

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Modeling: Input and configuration space [1]

The input space of a program P consists of k-tuples of values that could be input to P during execution.

The configuration space of P consists of all possible settings of the environment variables under which P could be used.

Consider program P that takes two integers x>0 and y>0 as inputs. The input space of P is the set of all pairs of positive non-zero integers.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Modeling: Input and configuration space [2]

Now suppose that this program is intended to be executed under the Windows and the MacOS operating system, through the Netscape or Safari browsers, and must be able to print to a local or a networked printer.

The configuration space of P consists of triples (X, Y, Z) where X represents an operating system, Y a browser, and Z a local or a networked printer.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Factors and levels

Consider a program P that takes n inputs corresponding to variables X_1, X_2, ..X_n. We refer to the inputs as factors. The inputs are also referred to as test parameters or as values.

Let us assume that each factor may be set at any one from a total of c_i, $1\leq i \leq n$ values. Each value assignable to a factor is known as a level. |F| refers to the number of levels for factor F.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Factor combinations

A set of values, one for each factor, is known as a factor combination.

For example, suppose that program P has two input variables X and Y. Let us say that during an execution of P, X and Y may each assume a value from the set {a, b, c} and {d, e, f}, respectively.

Thus, we have 2 factors and 3 levels for each factor. This leads to a total of $3^2=9$ factor combinations, namely (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e), and (c, f).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Factor combinations: Too large?

In general, for k factors with each factor assuming a value from a set of n values, the total number of factor combinations is n^k.

Suppose now that each factor combination yields one test case. For many programs, the number of tests generated for exhaustive testing could be exorbitantly large.

For example, if a program has 15 factors with 4 levels each, the total number of tests is $4^{15} \sim 10^9$. *Executing a billion tests might be impractical for many software applications.*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: Pizza Delivery Service (PDS) [1]

A PDS takes orders online, checks for their validity, and schedules Pizza for delivery.

A customer is required to specify the following four items as part of the online order: Pizza size, Toppings list, Delivery address and a home phone number. Let us denote these four factors by S, T, A, and P, respectively.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Pizza Delivery Service (PDS): Specs

Suppose now that there are three varieties for size: Large, Medium, and Small.

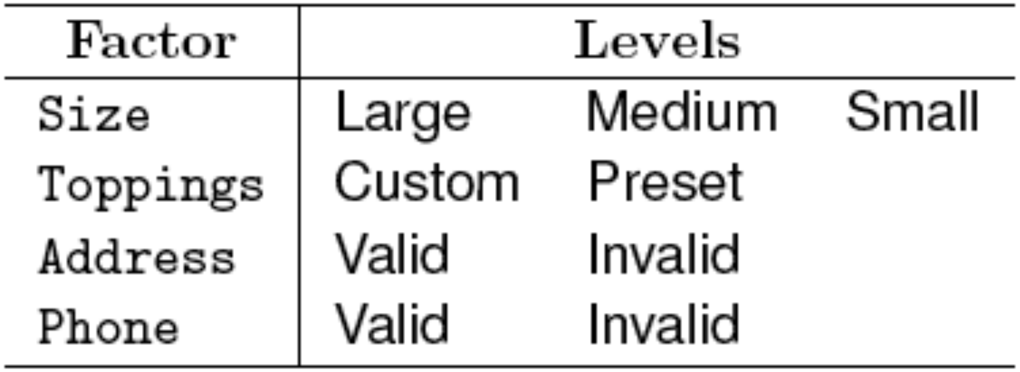
There is a list of 6 toppings from which to select. In addition, the customer can customize the toppings.

The delivery address consists of customer name, one line of address, city, and the zip code. The phone number is a numeric string possibly containing the dash (``--'') separator.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

PDS: Input space model



Factor	Levels		
Size	Large	Medium	Small
Toppings	Custom	Preset	
Address	Valid	Invalid	
Phone	Valid	Invalid	

The total number of factor combinations is $2^4+2^3=24$.

Suppose we consider 6+1=7 levels for Toppings. Number of combinations= $2^4+5x2^3+2^3+5x2^2=84$.

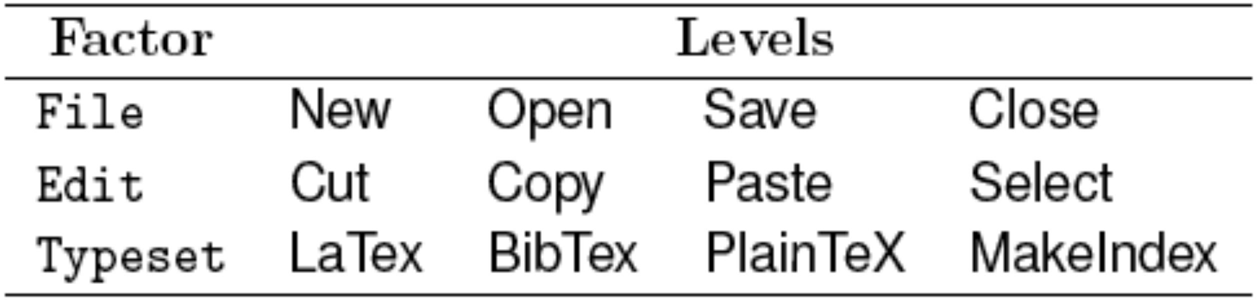
Different types of values for Address and Phone number will further increase the combinations

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: Testing a GUI

The Graphical User Interface of application T consists of three menus labeled File, Edit, and Format.



Factor	Levels			
File	New	Open	Save	Close
Edit	Cut	Copy	Paste	Select
Typeset	LaTex	BibTex	PlainTeX	MakeIndex

We have three factors in T. Each of these three factors can be set to any of four levels. Thus, we have a total 4^3=64 factor combinations.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: The UNIX sort utility

The sort utility has several options and makes an interesting example for the identification of factors and levels. The command line for sort is given below.

sort [-cmu] [-ooutput] [-Tdirectory] [-y [kmem]] [-zrecsz] [-dfiMnr] [-b] [tchar] [-kkeydef] [+pos1[-pos2]] [file...]

We have identified a total of 20 factors for the sort command. The levels listed in Table 11.1 of the book lead to a total of approximately 1.9x10^9 combinations.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: Compatibility testing

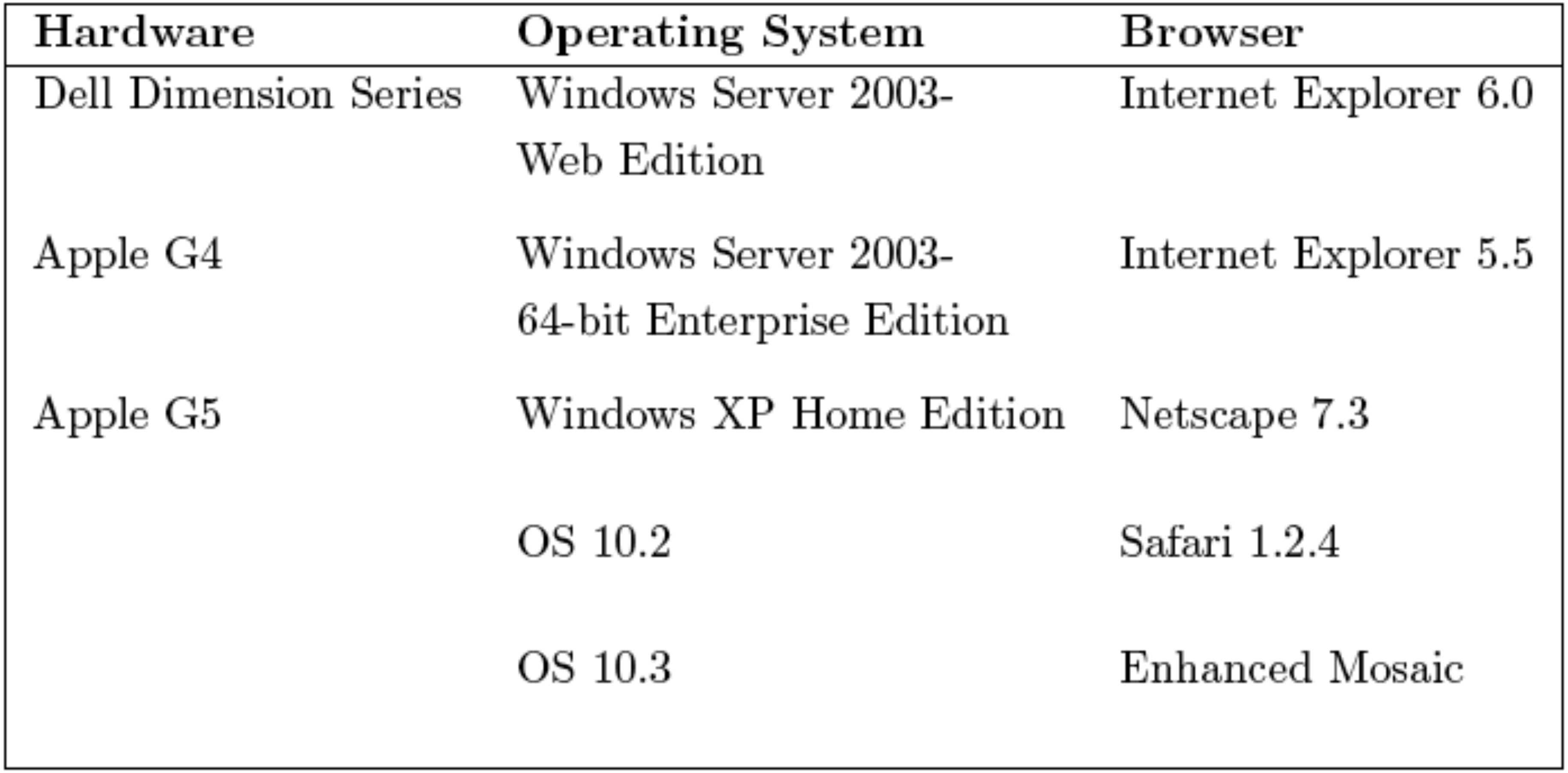
There is often a need to test a web application on different platforms to ensure that any claim such as ``Application X can be used under Windows and Mac OS X" are valid.

Here we consider a combination of hardware, operating system, and a browser as a platform. Let X denote a Web application to be tested for compatibility.

Given that we want X to work on a variety of hardware, OS, and browser combinations, it is easy to obtain three factors, i.e. hardware, OS, and browser.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Compatibility testing: Factor levels



Hardware	Operating System	Browser
Dell Dimension Series	Windows Server 2003- Web Edition	Internet Explorer 6.0
Apple G4	Windows Server 2003- 64-bit Enterprise Edition	Internet Explorer 5.5
Apple G5	Windows XP Home Edition	Netscape 7.3
	OS 10.2	Safari 1.2.4
	OS 10.3	Enhanced Mosaic

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Compatibility testing: Combinations

There are 75 factor combinations. However, some of these combinations are infeasible.

For example, Mac OS10.2 is an OS for the Apple computers and not for the Dell Dimension series PCs. Similarly, the Safari browser is used on Apple computers and not on the PC in the Dell Series.

While various editions of the Windows OS can be used on an Apple computer using an OS bridge such as the Virtual PC, we assume that this is not the case for testing application X.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Compatibility testing: Reduced combinations

The discussion above leads to a total of 40 infeasible factor combinations corresponding to the hardware-OS combination and the hardware-browser combination. Thus, in all we are left with 35 platforms on which to test X.

Note that there is a large number of hardware configurations under the Dell Dimension Series. These configurations are obtained by selecting from a variety of processor types, e.g. Pentium versus Athelon, processor speeds, memory sizes, and several others.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Compatibility testing: Reduced combinations-2

While testing against all configurations will lead to more thorough testing of application X, it will also increase the number of factor combinations, and hence the time to test.

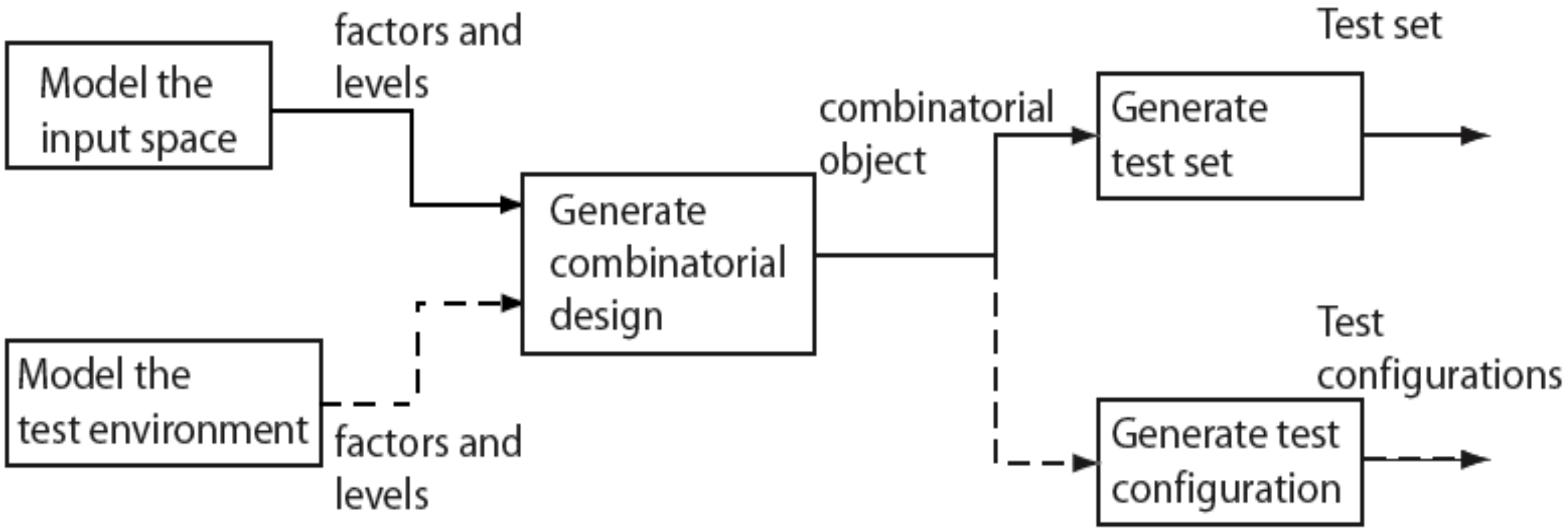
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.2. Combinatorial test design process

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Combinatorial test design process



Modeling of input space or the environment is not exclusive and one might apply either one or both depending on the application under test.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Combinatorial test design process: steps

Step 1: Model the input space and/or the configuration space. The model is expressed in terms of factors and their respective levels.

Step 2: The model is input to a combinatorial design procedure to generate a combinatorial object which is simply an array of factors and levels. Such an object is also known as a factor covering design.

Step 3: The combinatorial object generated is used to design a test set or a test configuration as the requirement might be.

Steps 2 and 3 can be automated.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Combinatorial test design process: test inputs

Each combination obtained from the levels listed in Table 6.1 can be used to generate many test inputs.

For example, consider the combination in which all factors are set to ``Unused" except the -o option which is set to ``Valid File" and the file option that is set to ``Exists.” Two sample test cases are:

t_1: sort -o afile bfile

t_2: sort -o cfile dfile

Is one of the above tests sufficient?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Combinatorial test design process: summary

Combination of factor levels is used to generate one or more test cases. For each test case, the sequence in which inputs are to be applied to the program under test must be determined by the tester.

Further, the factor combinations do not indicate in any way the sequence in which the generated tests are to be applied to the program under test. This sequence too must be determined by the tester.

The sequencing of tests generated by most test generation techniques must be determined by the tester and is not a unique characteristic of test generated in combinatorial testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.3. Fault model

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Fault model

Faults aimed at by the combinatorial design techniques are known as interaction faults.

We say that an interaction fault is triggered when a certain combination of $t \geq 1$ input values causes the program containing the fault to enter an invalid state.

Of course, this invalid state must propagate to a point in the program execution where it is observable and hence is said to reveal the fault.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

t-way interaction faults

Faults triggered by some value of an input variable, i.e. t=1, regardless of the values of other input variables, are known as simple faults.

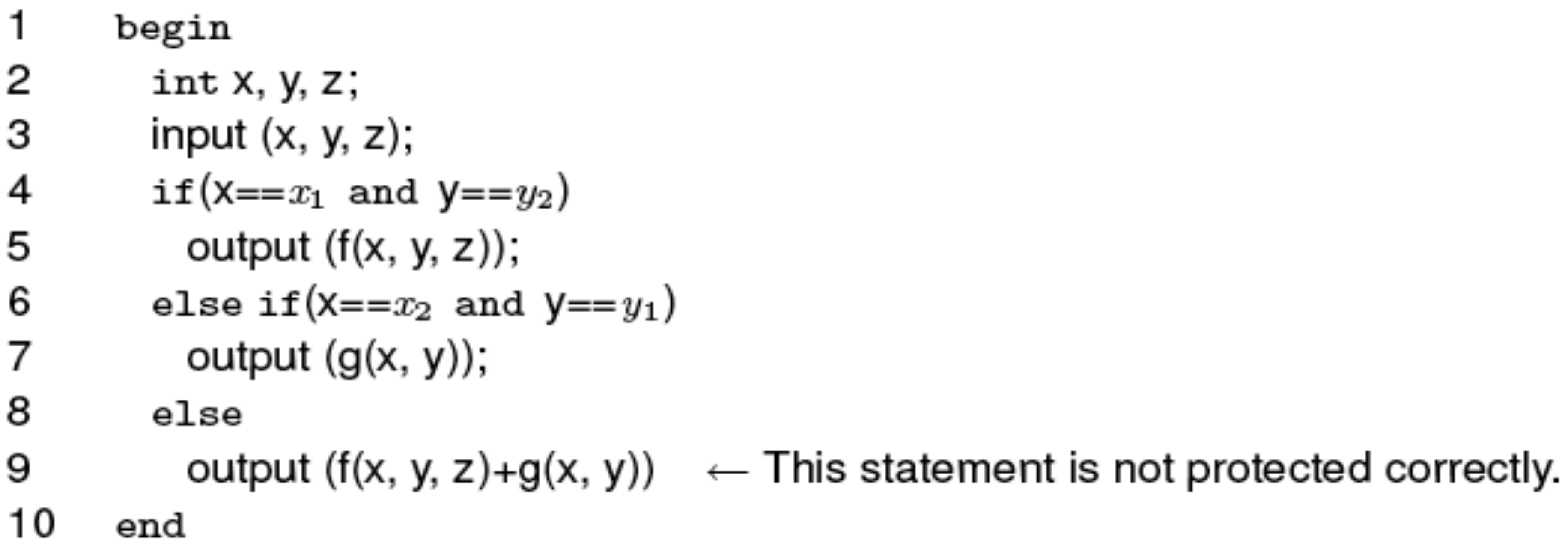
For t=2, the faults are known as pairwise interaction faults.

In general, for any arbitrary value of t, the faults are known as t--way interaction faults.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Pairwise interaction fault: Example



```
begin
  int x, y, z;
  input (x, y, z);
  if(x==$x_1$ and y==$y_2$)
    output (f(x, y, z));
  else if(x==$x_2$ and y==$y_1$)
    output (g(x, y));
  else
    output (f(x, y, z)+g(x, y))    ← This statement is not protected correctly.
end
```

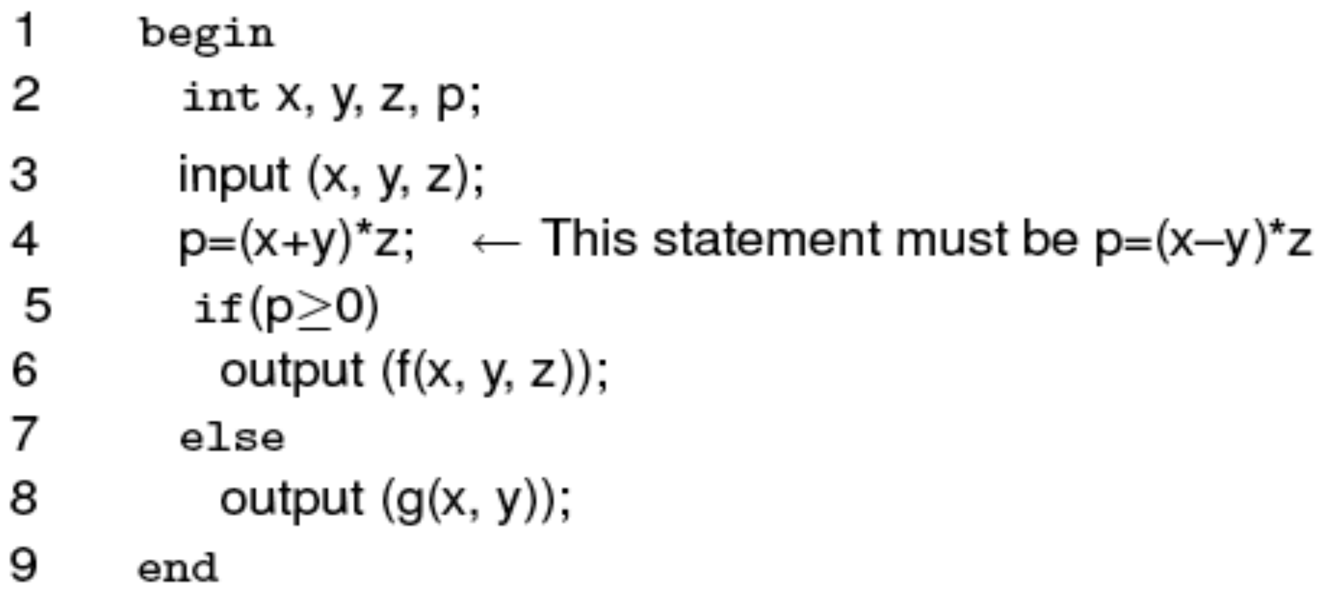
Correct output: f(x, y, z)-g(x, y) when X=x1 and Y=y1.

This is a pairwise interaction fault due to the interaction between factors X and Y.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

3-way interaction fault: Example



```
begin
  int x, y, z, p;
  input (x, y, z);
  p=(x+y)*z;    ← This statement must be p=(x–y)*z
   if(p≥0)
     output (f(x, y, z));
  else
     output (g(x, y));
end
```

This fault is triggered by all inputs such that x+y≠x-y and z ≠ 0. However, the fault is revealed only by the following two of the eight possible input combinations: x=-1, y=1, z=1 and x=-1, y=-1, z=1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

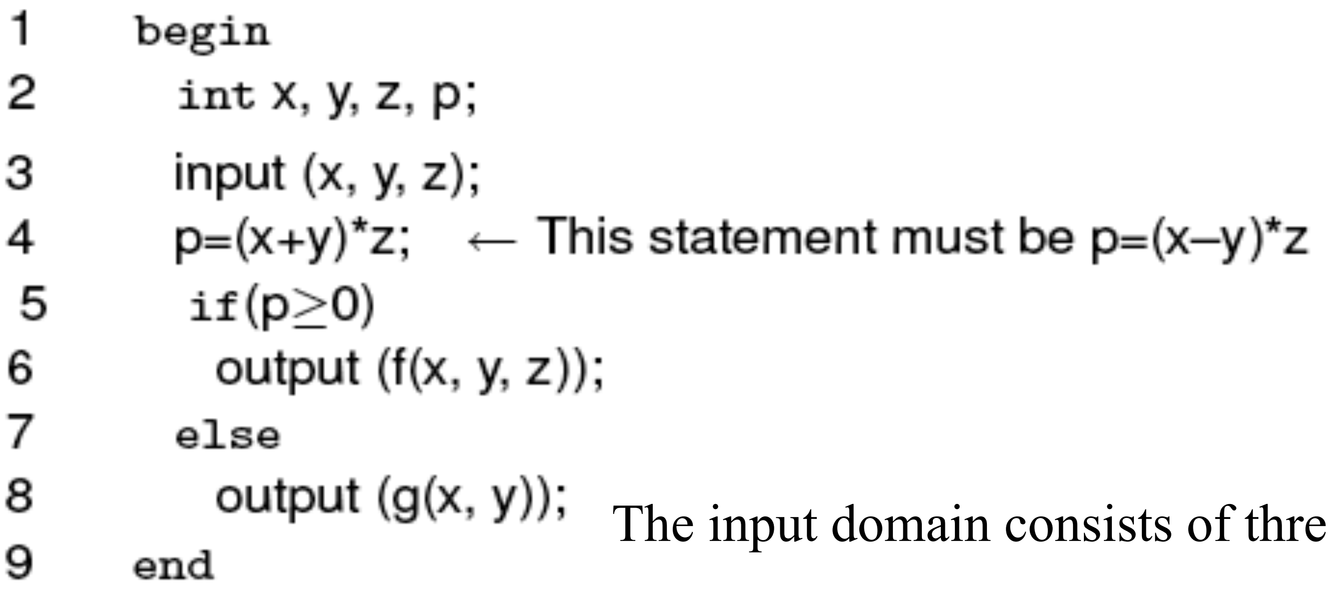
Fault vectors

Given a set of k factors f1, f2,.., fk, each at qi, 1≤ i ≤ k levels, a vector V of factor levels is (l1, l2,.., lk), where li, 1 ≤ i ≤ k is a specific level for the corresponding factor. V is also known as a run.

A run V is a fault vector for program P if the execution of P against a test case derived from V triggers a fault in P. V is considered as a t-fault vector if any t ≤ k elements in V are needed to trigger a fault in P. Note that a t-way fault vector for P triggers a t-way fault in P.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Fault vectors: Example



```
begin
  int x, y, z, p;
  input (x, y, z);
  p=(x+y)*z;    ← This statement must be p=(x–y)*z
  if(p≥0)
    output (f(x, y, z));
  else
    output (g(x, y));
end
```

The input domain consists of three factors x, y, and z each having two levels. There is a total of eight runs. For example, (1,1, 1) and (-1, -1, 0) are two runs.

Of these eight runs, (-1, 1, 1) and (-1, -1, 1) are three fault vectors that trigger the 3-way fault. (x1, y1, *) is a 2-way fault vector given that the values x1 and y1 trigger the two-way fault.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Goal reviewed

The goal of the test generation techniques described in this chapter is to generate a sufficient number of runs such that tests generated from these runs reveal all t-way faults in the program under test.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Goal reviewed

The number of such runs increases with the value of t. In many situations, t is set to 2 and hence the tests generated are expected to reveal pairwise interaction faults.

Of course, while generating t-way runs, one automatically generates some t+1, t+2, .., t +k-1, and k-way runs also. Hence, there is always a chance that runs generated with t=2 reveal some higher level interaction faults.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.4. Latin squares

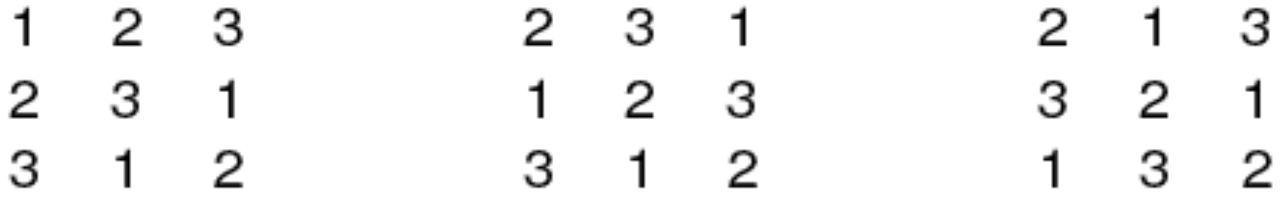
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Latin Squares

Let S be a finite set of n symbols. A Latin square of order n is an n x n matrix such that no symbol appears more than once in a row and column. The term ``Latin square" arises from the fact that the early versions used letters from the Latin alphabet A, B, C, etc. in a square arrangement.

$$\begin{matrix} A & B \\ B & A \end{matrix} \qquad \begin{matrix} B & A \\ A & B \end{matrix}$$

S={A, B}. Latin squares of order 2.



$$\begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{matrix} \qquad \begin{matrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{matrix} \qquad \begin{matrix} 2 & 1 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{matrix}$$

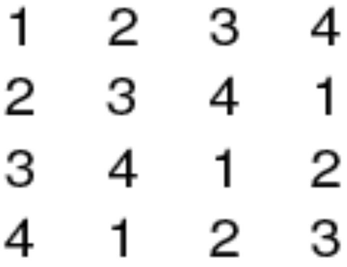
S={1, 2, 3}. Latin squares of order 3.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Larger Latin Squares

Larger Latin squares of order n can be constructed by creating a row of n distinct symbols. Additional rows can be created by permuting the first row.



1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

For example, here is a Latin square M of order 4 constructed by cyclically rotating the first row and placing successive rotations in subsequent rows.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Modulo arithmetic and Latin Squares

A Latin square of order n>2 can also be constructed easily by doing modulo arithmetic. For example, the Latin square M of order 4 given below is constructed such that M(i, j)=i+j (mod 4), 1≤ (i, j) ≤ 4.

	1	2	3	4
1	2	3	0	1
2	3	0	1	2
3	0	1	2	3
4	1	2	3	0

A Latin square based on integers 0, 1… n is said to be in standard form if the elements in the top row and the leftmost column are arranged in order.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.5. Mutually orthogonal Latin squares

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutually Orthogonal Latin Squares (MOLS)

Let M1 and M2 be two Latin squares, each of order n. Let M1(i, j) and M2(i, j) denote, respectively, the elements in the ith row and jth column of M1 and M2.

We now create an n x n matrix M from M1 and M2 such that the L(i, j) is M1(i, j)M2(i, j), i.e. we simply juxtapose the corresponding elements of M1 and M2.

If each element of M is unique, i.e. it appears exactly once in M, then M1 and M2 are said to be mutually orthogonal Latin squares of order n.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MOLS: Example

There are no MOLS of order 2. MOLS of order 3 follow.

$$M_1 = \begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array} \qquad M_2 = \begin{array}{ccc} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{array}$$

$$L = \begin{array}{ccc} 1\,2 & 2\,3 & 3\,1 \\ 2\,1 & 3\,2 & 1\,3 \\ 3\,3 & 1\,1 & 2\,2 \end{array}$$

Juxtaposing the corresponding elements gives us L. Its elements are unique and hence M1 and M2 are MOLS.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MOLS: How many of a given order?

MOLS(n) is the set of MOLS of order n. When n is prime, or a power of prime, MOLS(n) contains n-1 mutually orthogonal Latin squares. Such a set of MOLS is a complete set.

MOLS do not exist for n=2 and n=6 but they do exist for all other values of n>2. Numbers 2 and 6 are known as Eulerian numbers after the famous mathematician Leonhard Euler (1707-1783). The number of MOLS of order n is denoted by N(n). When n is prime or a power of prime, N(n)=n-1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MOLS: Construction [1]

Example: We begin by constructing a Latin square of order 5 given the symbol set S={1, 2, 3, 4, 5}.

$$M_1 = \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \\ 5 & 1 & 2 & 3 & 4 \end{array}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MOLS: Construction [2]

Next, we obtain M2 by rotating rows 2 through 5 of M1 by two positions to the left.

$$M_2 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 1 & 2 \\ 5 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 & 1 \\ 4 & 5 & 1 & 2 & 3 \end{matrix}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MOLS: Construction [3]

M3 and M4 are obtained similarly but by rotating the first row of M1 by 3 and 4 positions, respectively.

$$M_3 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 1 & 2 & 3 \\ 2 & 3 & 4 & 5 & 1 \\ 5 & 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 1 & 2 \end{matrix} \qquad M_4 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 4 \\ 4 & 5 & 1 & 2 & 3 \\ 3 & 4 & 5 & 1 & 2 \\ 2 & 3 & 4 & 5 & 1 \end{matrix}$$

Thus, we get MOLS(5)={M1, M2, M3, M4}. It is easy to check that indeed the elements of MOLS(5) are mutually orthogonal by superimposing them pairwise.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MOLS: Construction, limitation

The method illustrated in the previous example is guaranteed to work only when constructing MOLS(n) for n that is prime or a power of prime. For other values of n, the maximum size of MOLS(n) is n-1.

There is no general method available to construct the largest possible MOLS(n) for n that is not a prime or a power of prime. The CRC Handbook of Combinatorial Designs gives a large table of MOLS.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.6. Pairwise designs: Binary factors

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Pairwise designs

We will now look at a simple technique to generate a subset of factor combinations from the complete set. Each combination selected generates at least one test input or test configuration for the program under test.

Only 2-valued, or binary, factors are considered. Each factor can be at one of two levels. This assumption will be relaxed later.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Pairwise designs: Example

Suppose that a program to be tested requires 3 inputs, one corresponding to each input variable. Each variable can take only one of two distinct values.

Considering each input variable as a factor, the total number of factor combinations is 2^3. Let X, Y, and Z denote the three input variables and {X1, X2}, {Y1, Y2}, {Z1, Z2} their respective sets of values. All possible combinations of these three factors follow.

$$\begin{array}{ll}
(X_1, Y_1, Z_1) & (X_1, Y_1, Z_2) \\
(X_1, Y_2, Z_1) & (X_1, Y_2, Z_2) \\
(X_2, Y_1, Z_1) & (X_2, Y_1, Z_2) \\
(X_2, Y_2, Z_1) & (X_2, Y_2, Z_2)
\end{array}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Pairwise designs: Reducing the combinations

Now suppose we want to generate tests such that each pair appears in at least one test. There are 12 such pairs: (X1, Y1), (X1, Y2), (X1, Z1), (X1, Z2), (X2, Y1), (X2, Y2), (X2, Z1), (X2, Z2), (Y1, Z1), (Y1, Z2), (Y2, Z1), and (Y2, Z2). The following four combinations cover all pairs:

$$\begin{array}{cc} (X_1, Y_1, Z_2) & (X_1, Y_2, Z_1) \\ (X_2, Y_1, Z_1) & (X_2, Y_2, Z_2) \end{array}$$

The above design is also known as a pairwise design. It is a balanced design because each value occurs exactly the same number of times. *There are several sets of four combinations that cover all 12 pairs.*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

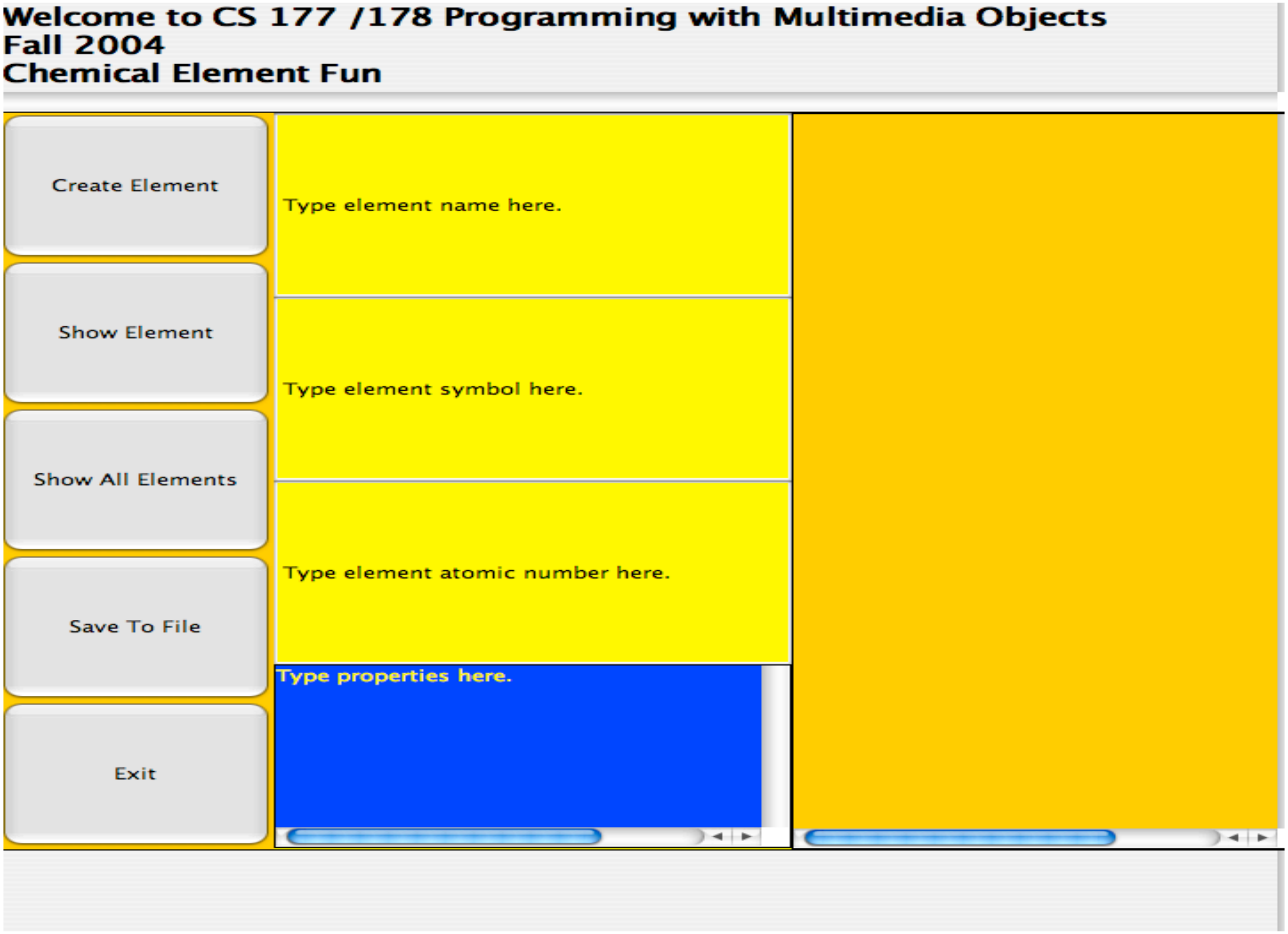
Example: ChemFun applet

A Java applet ChemFun allows its user to create an in-memory database of chemical elements and search for an element. The applet has 5 inputs listed after the next slide with their possible values.

We refer to the inputs as factors. For simplicity we assume that each input has exactly two possible values.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: ChemFun applet



Welcome to CS 177 /178 Programming with Multimedia Objects
Fall 2004
Chemical Element Fun

Type element name here.

Type element symbol here.

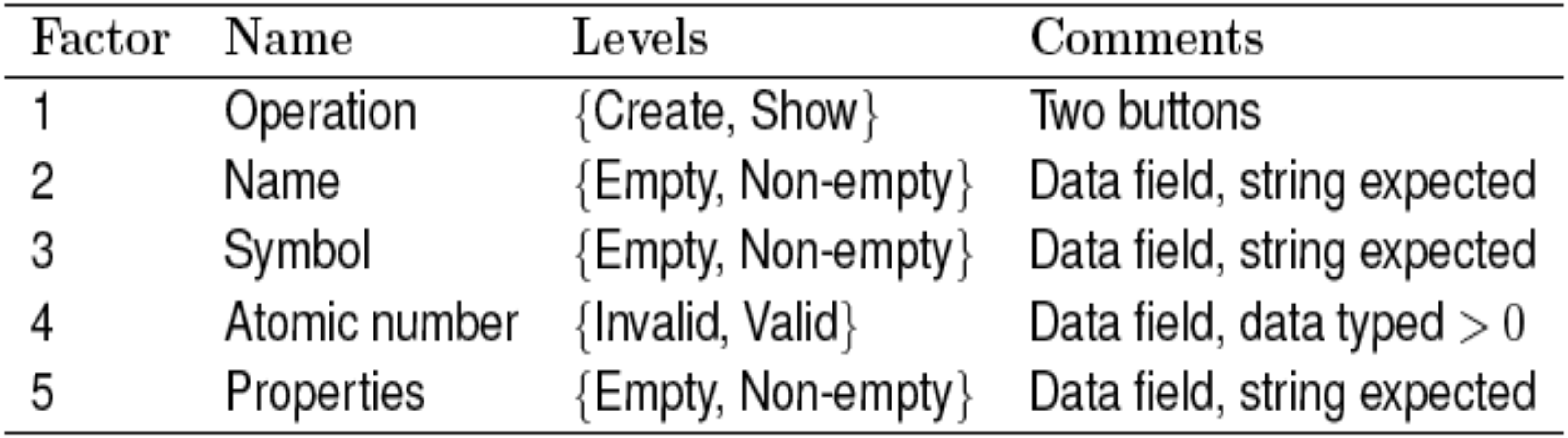
Type element atomic number here.

Type properties here.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: ChemFun applet: Factor identification



Factor	Name	Levels	Comments
1	Operation	{Create, Show}	Two buttons
2	Name	{Empty, Non-empty}	Data field, string expected
3	Symbol	{Empty, Non-empty}	Data field, string expected
4	Atomic number	{Invalid, Valid}	Data field, data typed > 0
5	Properties	{Empty, Non-empty}	Data field, string expected

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: Input/Output

Input: n=5 factors

Output: A set of factor combinations such that all pairs of input values are covered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

ChemFun applet: Step 1

Compute the smallest integer k such that n≤ $|S_{2k-1}|$

S_{2k-1}: Set of all binary strings of length 2k-1, k>0.

$$S_{2k-1}= \binom{2k-1}{k} \qquad \binom{n}{k} = \frac{!n}{!(n-k)\ !k}$$

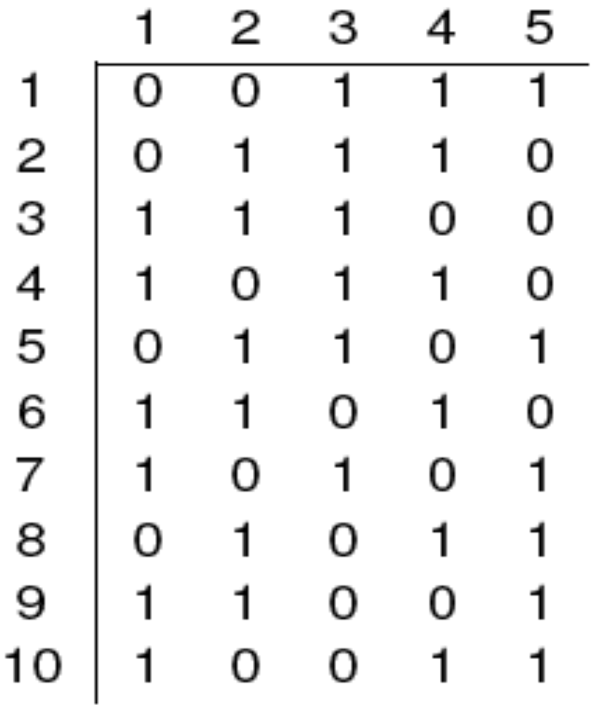
For k=3 we have S_5= 10 and for k=2, S_3= 3. Hence the desired integer k=3.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

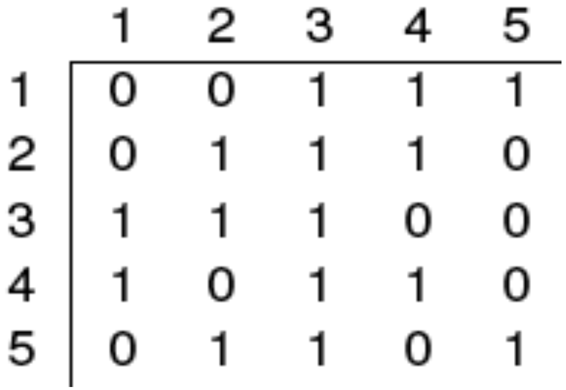
ChemFun applet: Step 2

Select any subset of n strings from S_{2k-1}. We have, k=3 and we have the following strings in the set S_5.



	1	2	3	4	5
1	0	0	1	1	1
2	0	1	1	1	0
3	1	1	1	0	0
4	1	0	1	1	0
5	0	1	1	0	1
6	1	1	0	1	0
7	1	0	1	0	1
8	0	1	0	1	1
9	1	1	0	0	1
10	1	0	0	1	1

We select first five of the 10 strings in S_5.



	1	2	3	4	5
1	0	0	1	1	1
2	0	1	1	1	0
3	1	1	1	0	0
4	1	0	1	1	0
5	0	1	1	0	1

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: Step 3

Append 0's to the end of each selected string. This will increase the size of each string from 2k-1 to 2k.

	1	2	3	4	5
1	0	0	1	1	1
2	0	1	1	1	0
3	1	1	1	0	0
4	1	0	1	1	0
5	0	1	1	0	1

→

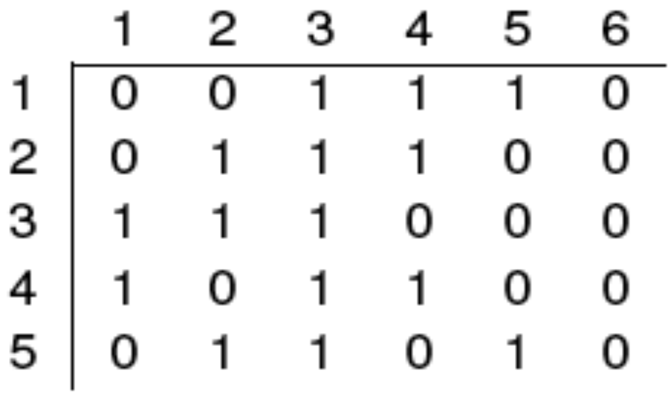
	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	1	1	1	0	0
3	1	1	1	0	0	0
4	1	0	1	1	0	0
5	0	1	1	0	1	0

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: Step 4

Each combination is of the kind (X_1, X_2,…, X_n), where the value of each variable is selected depending on whether the bit in column i, 1≤ i ≤ n, is a 0 or a 1.



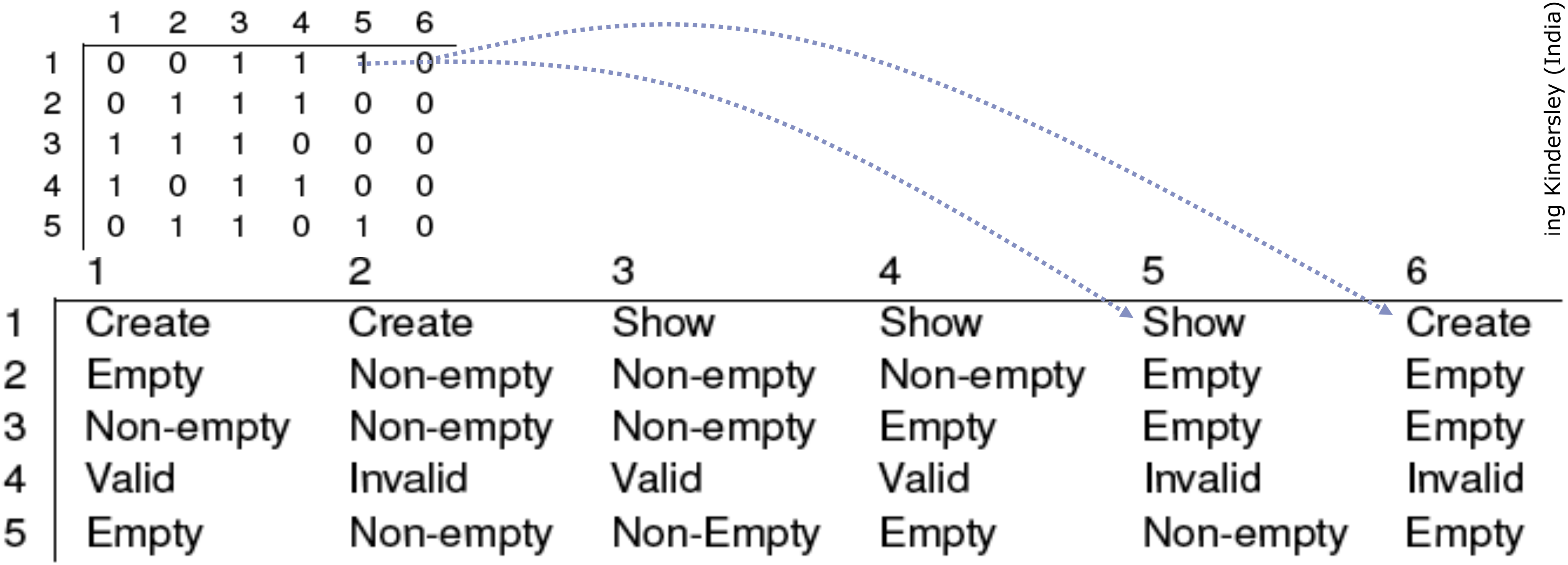
	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	1	1	1	0	0
3	1	1	1	0	0	0
4	1	0	1	1	0	0
5	0	1	1	0	1	0

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: Step 4 (contd.)

The following factor combinations by replacing the 0s and 1s in each column by the corresponding values of each factor.



	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	1	1	1	0	0
3	1	1	1	0	0	0
4	1	0	1	1	0	0
5	0	1	1	0	1	0

	1	2	3	4	5	6
1	Create	Create	Show	Show	Show	Create
2	Empty	Non-empty	Non-empty	Non-empty	Empty	Empty
3	Non-empty	Non-empty	Non-empty	Empty	Empty	Empty
4	Valid	Invalid	Valid	Valid	Invalid	Invalid
5	Empty	Non-empty	Non-Empty	Empty	Non-empty	Empty

ing Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: tests

	1	2	3	4	5	6
1	Create	Create	Show	Show	Show	Create
2	Empty	Non-empty	Non-empty	Non-empty	Empty	Empty
3	Non-empty	Non-empty	Non-empty	Empty	Empty	Empty
4	Valid	Invalid	Valid	Valid	Invalid	Invalid
5	Empty	Non-empty	Non-Empty	Empty	Non-empty	Empty

$$t_1 : \quad < \quad \text{Button} = \text{Create}, \text{Name} = \text{“”}, \text{Symbol} = \text{‘C”}, \\ \text{Atomic number} = 6, \text{Properties} = \text{“”} \quad >$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

ChemFun applet: All tests

$$
\begin{array}{llll}
T & = \{ & t_1: & < \text{Button} = \text{Create}, \text{Name} = \text{“”}, \text{Symbol} = \text{‘C”}, \\
 & & & \text{Atomic number} = 6, \text{Properties} = \text{“”} > \\
 & & t_2: & < \text{Button} = \text{Create}, \text{Name} = \text{“Carbon”}, \text{Symbol} = \text{‘C”}, \\
 & & & \text{Atomic number} = -6, \text{Properties} = \text{“Non-metal”} > \\
 & & t_3: & < \text{Button} = \text{Show}, \text{Name} = \text{“Hydrogen”}, \text{Symbol} = \text{‘C”}, \\
 & & & \text{Atomic number} = 1, \text{Properties} = \text{“Non-metal”} > \\
 & & t_4: & < \text{Button} = \text{Show}, \text{Name} = \text{“Carbon”}, \text{Symbol} = \text{‘C”}, \\
 & & & \text{Atomic number} = 6, \text{Properties} = \text{“”} > \\
 & & t_5: & < \text{Button} = \text{Show}, \text{Name} = \text{“”}, \text{Symbol} = \text{“”}, \\
 & & & \text{Atomic number} = -6, \text{Properties} = \text{“Non-metal”} > \\
 & & t_6: & < \text{Button} = \text{Create}, \text{Name} = \text{“”}, \text{Symbol} = \text{“”}, \\
 & & & \text{Atomic number} = -6, \text{Properties} = \text{“”} > \\
 & \} & &
\end{array}
$$

Recall that the total number of combinations is 32. Requiring only pairwise coverage reduces the tests to 6.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.7. Pairwise designs: Multi-valued factors

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Pairwise designs: Multi-valued factors

Next we will learn how to use MOLS to construct test configurations when:

- The number of factors is two or more,
- The number of levels for each factor is more than two,
- All factors have the same number of levels.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multi-valued factors: Sample problem

DNA sequencing is a common activity amongst biologists and other researchers. Several genomics facilities are available that allow a DNA sample to be submitted for sequencing.

One such facility is offered by The Applied Genomics Technology Center (AGTC) at the School of Medicine in Wayne State University.

The submission of the sample itself is done using a software application available from AGTC. We refer to this software as AGTCS.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Sample problem (contd.)

AGTCS is supposed to work on a variety of platforms that differ in their hardware and software configurations. Thus, the hardware platform and the operating system are two factors to be considered while developing a test plan for AGTCS.

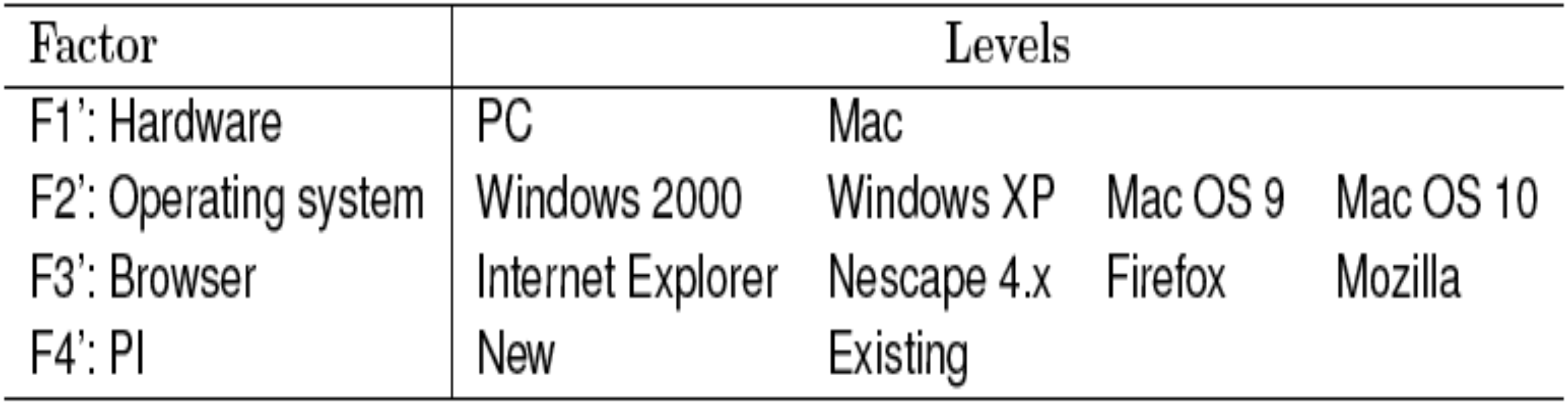
In addition, the user of AGTCS, referred to as PI, must either have a profile already created with AGTCS or create a new one prior to submitting a sample. AGTCS supports only a limited set of browsers.

For simplicity we consider a total of four factors with their respective levels given next.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

DNA sequencing: factors and levels



Factor	Levels			
F1’: Hardware	PC	Mac		
F2’: Operating system	Windows 2000	Windows XP	Mac OS 9	Mac OS 10
F3’: Browser	Internet Explorer	Nescape 4.x	Firefox	Mozilla
F4’: PI	New	Existing		

There are 64 combinations of the factors listed. As PCs and Macs run their dedicated operating systems, the number of combinations reduces to 32.

We want to test under enough configurations so that all possible pairs of factor levels are covered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

DNA sequencing: Approach to test design

We can now proceed to design test configurations in at least two ways. One way is to treat the testing on PC and Mac as two distinct problems and design the test configurations independently. Exercise 6.12 asks you to take this approach and explore its advantages over the second approach used in this example.

The approach used in this example is to arrive at a common set of test configurations that obey the constraint related to the operating systems.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

DNA sequencing: Test design algorithm

Input: n=4 factors. |F1’|=2, |F2’|=4, |F3’|=4, |F4’|=2, where F1’, F2’, F3’, and F4’ denote, respectively, hardware, OS, browser, and PI.

Output: A set of factor combinations such that all pairwise combinations are covered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test design algorithm: Step 1

Reliable the factors as F1, F2, F3, F4 such that |F1|≥|F2| ≥ |F3| ≥ |F4|.

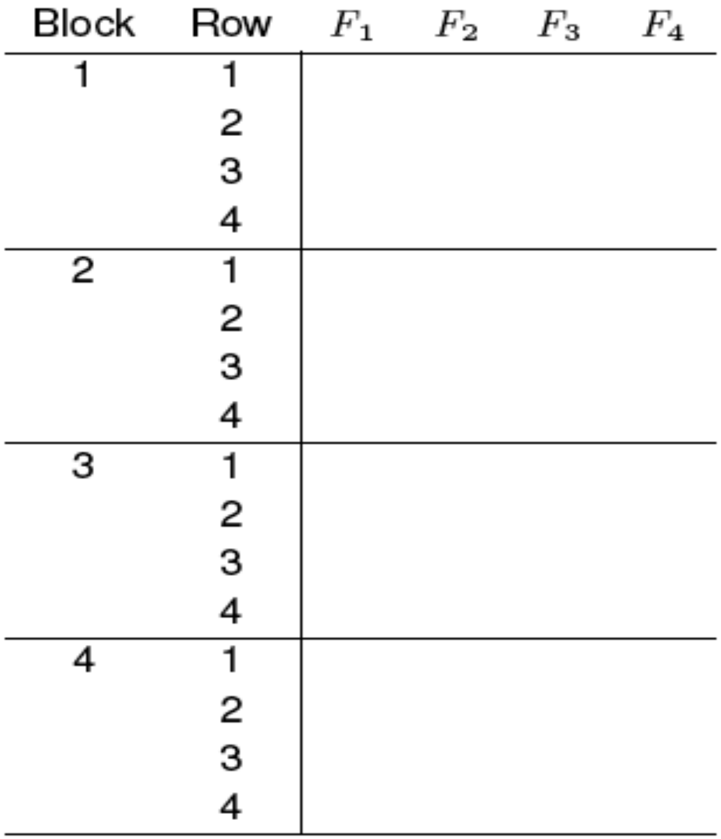
Doing so gives us F1=F2', F2=F3', F3=F1', F4=F4', b=k=4. Note that a different assignment is also possible because |F1|=|F4|and |F2|=|F3|.

Let b=|F1|=4 and k=|F2|=4

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test design algorithm: Step 2

Prepare a table containing 4 columns and b x k=16 rows divided into 4 blocks. Label the columns as F_1, F_2, … F_n. Each block contains k rows.



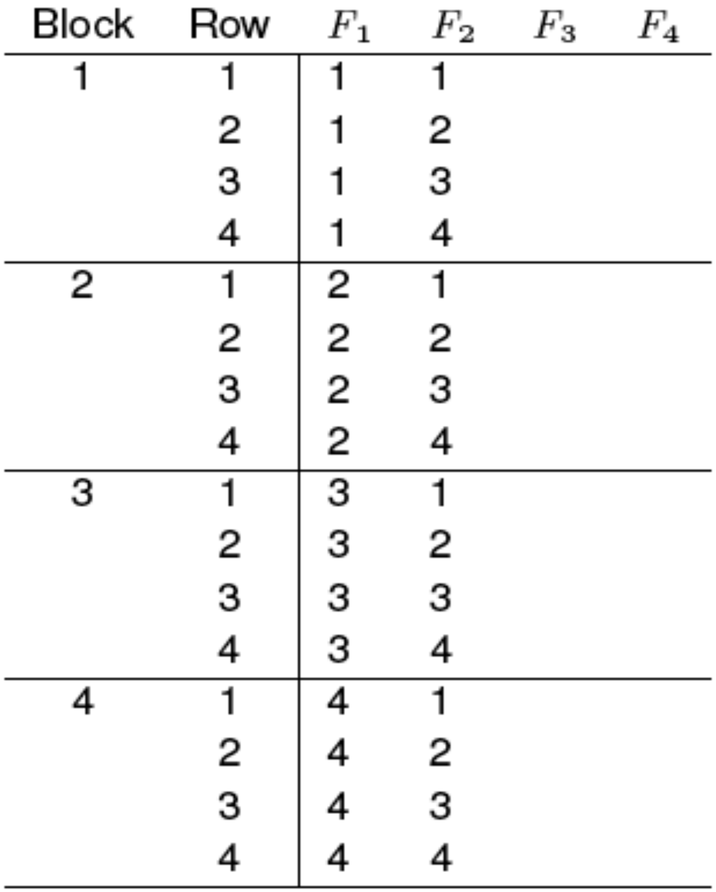
Block	Row	F_1	F_2	F_3	F_4
1	1				
	2				
	3				
	4				
2	1				
	2				
	3				
	4				
3	1				
	2				
	3				
	4				
4	1				
	2				
	3				
	4				

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Step 3 (contd.)

Fill column F1 with 1's in Block 1, 2's in Block 2, and so on. Fill Block 1 of column F2 with the sequence 1, 2,.., k in rows 1 through k (k=4).



Block	Row	F_1	F_2	F_3	F_4
1	1	1	1		
	2	1	2		
	3	1	3		
	4	1	4		
2	1	2	1		
	2	2	2		
	3	2	3		
	4	2	4		
3	1	3	1		
	2	3	2		
	3	3	3		
	4	3	4		
4	1	4	1		
	2	4	2		
	3	4	3		
	4	4	4		

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Step 4

Find MOLS of order 4. As 4 is a power of prime, we can use the procedure described earlier.

We choose the following set of MOLS of order 4.

$$M_1 = \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array} \qquad M_2 = \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{array} \qquad M_3 = \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \end{array}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

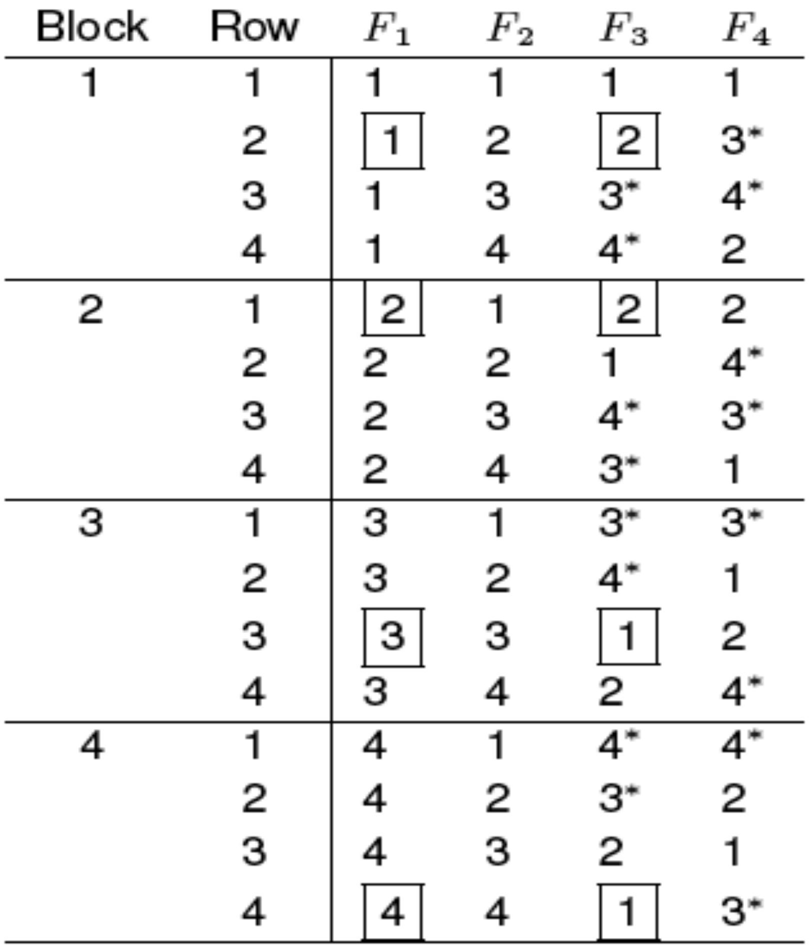
Test design algorithm: Step 5

Fill the remaining two columns of the table constructed earlier using columns of M1 for F_3 and M2 for F_4.

A boxed entry in each row indicates a pair that does not satisfy the operating system constraint. An entry marked with an asterisk (*) indicates an invalid level.

From M1

From M2



Block	Row	F_1	F_2	F_3	F_4
1	1	1	1	1	1
	2	$\boxed{1}$	2	$\boxed{2}$	3*
	3	1	3	3*	4*
	4	1	4	4*	2
2	1	$\boxed{2}$	1	$\boxed{2}$	2
	2	2	2	1	4*
	3	2	3	4*	3*
	4	2	4	3*	1
3	1	3	1	3*	3*
	2	3	2	4*	1
	3	$\boxed{3}$	3	$\boxed{1}$	2
	4	3	4	2	4*
4	1	4	1	4*	4*
	2	4	2	3*	2
	3	4	3	2	1
	4	$\boxed{4}$	4	$\boxed{1}$	3*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Step 6 [1]

Using the 16 entries in the table above, we can obtain 16 distinct test configurations for AGTCS. However, we need to resolve two problems before we get to the design of test configurations.

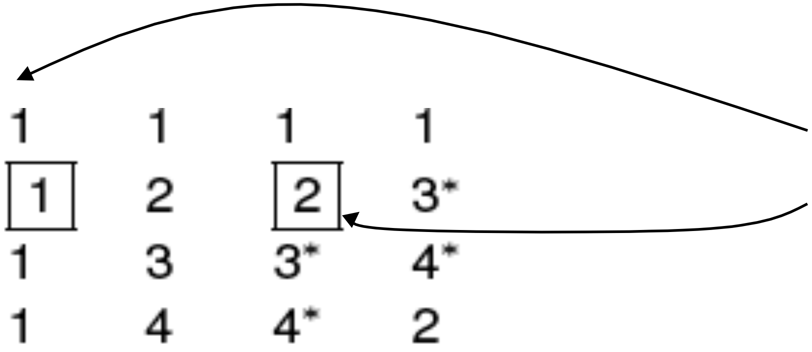
Problem 1: Factors F3 and F4 can only assume values 1 and 2 whereas the table above contains other infeasible values for these two factors. These infeasible values are marked with an asterisk.

Solution: One simple way to get rid of the infeasible values is to replace them by an arbitrarily selected feasible value for the corresponding factor..

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test design algorithm: Step 6 [2]

Problem 2: Some configurations do not satisfy the operating system constraint. Four such configurations are highlighted in the design by enclosing the corresponding numbers in rectangles. Here is an example:



1	1	1	1
1	2	2	3*
1	3	3*	4*
1	4	4*	2

F_1: Operating system=1(Win 2000) F3: Hardware=2 (Mac) is infeasible.

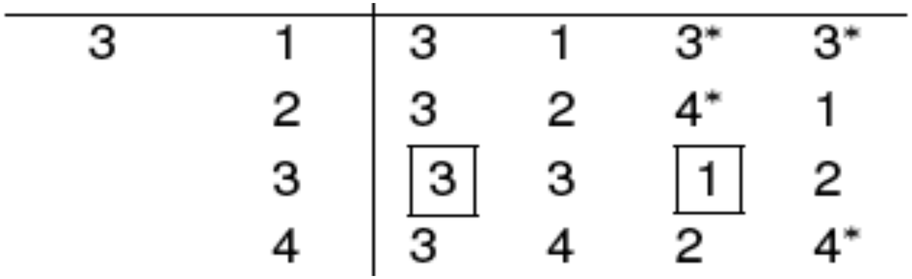
Here we are assume that one is not using Virtual PC on the Mac.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Step 6 [3]

Delete rows with conflicts?: Obviously we cannot delete these rows as that would leave some pairs uncovered. Consider block 3.



3	1	3	1	3*	3*
	2	3	2	4*	1
	3	$\boxed{3}$	3	$\boxed{1}$	2
	4	3	4	2	4*

Removing Row~3 will leave the following five pairs uncovered: (F_1=3, F_2=3), (F_1=3, F_4=2), (F_2=3, F_3=1), (F_2=3, F_4=2), and (F_3=1, F_4=2).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Step 6 [4]

Proposed solution: We follow a two step procedure to remove the highlighted configurations and retain complete pairwise coverage.

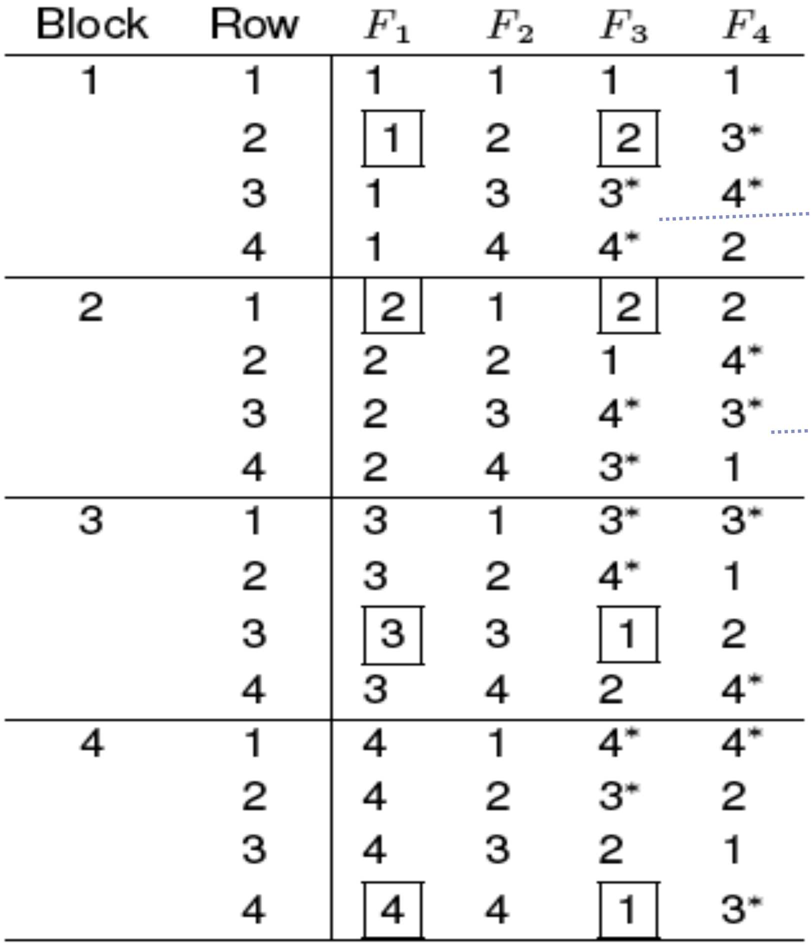
Step 1: Modify the four highlighted rows so they do not violate the constraint.

Step 2: Add new configurations that cover the pairs that are left uncovered when we replace the highlighted rows.

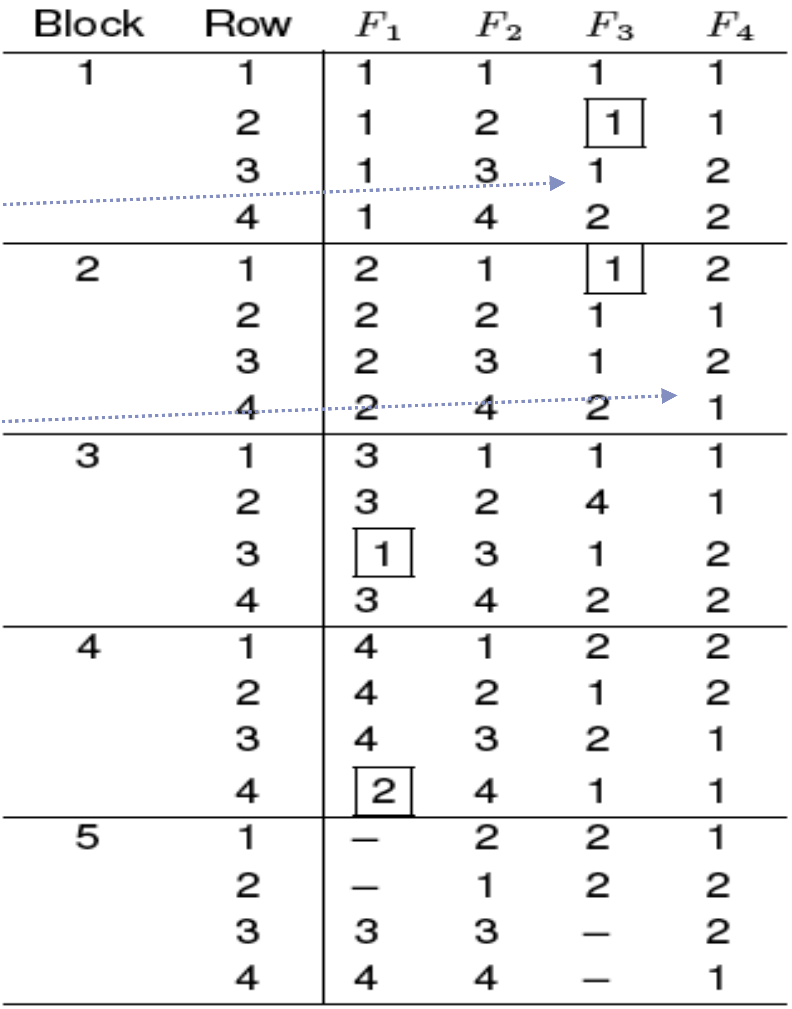
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test design algorithm: Step 6 [5]

F1: OS F2: Browser F3: Hardware F4: PI



Block	Row	F_1	F_2	F_3	F_4
1	1	1	1	1	1
	2	$\boxed{1}$	2	$\boxed{2}$	3*
	3	1	3	3*	4*
	4	1	4	4*	2
2	1	$\boxed{2}$	1	$\boxed{2}$	2
	2	2	2	1	4*
	3	2	3	4*	3*
	4	2	4	3*	1
3	1	3	1	3*	3*
	2	3	2	4*	1
	3	$\boxed{3}$	3	$\boxed{1}$	2
	4	3	4	2	4*
4	1	4	1	4*	4*
	2	4	2	3*	2
	3	4	3	2	1
	4	$\boxed{4}$	4	$\boxed{1}$	3*



Block	Row	F_1	F_2	F_3	F_4
1	1	1	1	1	1
	2	1	2	$\boxed{1}$	1
	3	1	3	1	2
	4	1	4	2	2
2	1	2	1	$\boxed{1}$	2
	2	2	2	1	1
	3	2	3	1	2
	4	2	4	2	1
3	1	3	1	1	1
	2	3	2	4	1
	3	$\boxed{1}$	3	1	2
	4	3	4	2	2
4	1	4	1	2	2
	2	4	2	1	2
	3	4	3	2	1
	4	$\boxed{2}$	4	1	1
5	1	—	2	2	1
	2	—	1	2	2
	3	3	3	—	2
	4	4	4	—	1

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test design algorithm: Design configurations

We can easily construct 20 test configurations from the design obtained. This is in contrast to 32 configurations obtained using a brute force method.

Can we remove some rows from the design without affecting pairwise coverage?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Shortcomings of using MOLS

A sufficient number of MOLS might not exist for the problem at hand.

While the MOLS approach assists with the generation of a balanced design in that all interaction pairs are covered an equal number of times, the number of test configurations is often larger than what can be achieved using other methods.

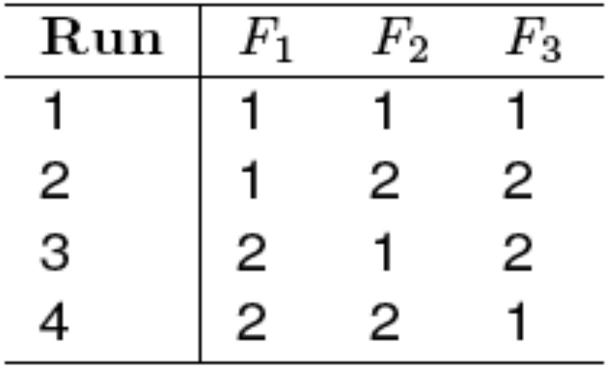
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

6.8. Orthogonal Arrays

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Orthogonal arrays

Examine this matrix and extract as many properties as you can:



Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

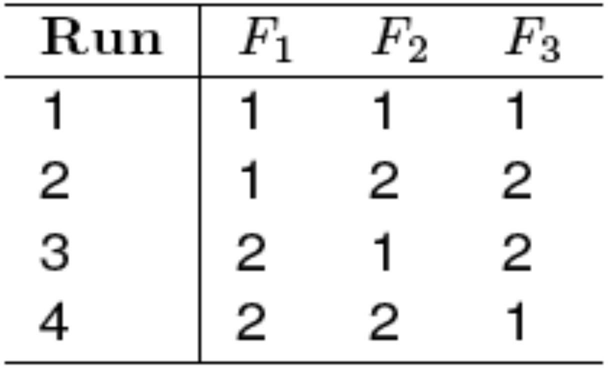
An orthogonal array, such as the one above, is an N x k matrix in which the entries are from a finite set S of s symbols such that any N x t sub array contains each t-tuple exactly the same number of times. Such an orthogonal array is denoted by OA(N, k, s, t).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Orthogonal arrays: Example

The following orthogonal array has 4 runs and has a strength of 2. It uses symbols from the set {1, 2}. This array is denoted as OA(4, 3, 2, 2). Note that the value of parameter k is 3 and hence we have labeled the columns as F1, F2, and F3 to indicate the three factors.



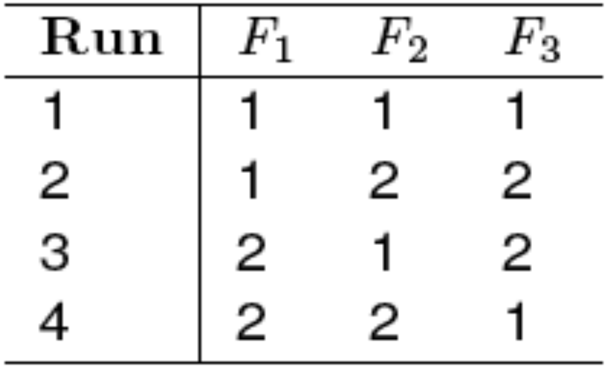
Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Orthogonal arrays: Index

The index of an orthogonal array is denoted by λ and is equal to N/s^t. N is referred to as the number of runs and t as the strength of the orthogonal array.



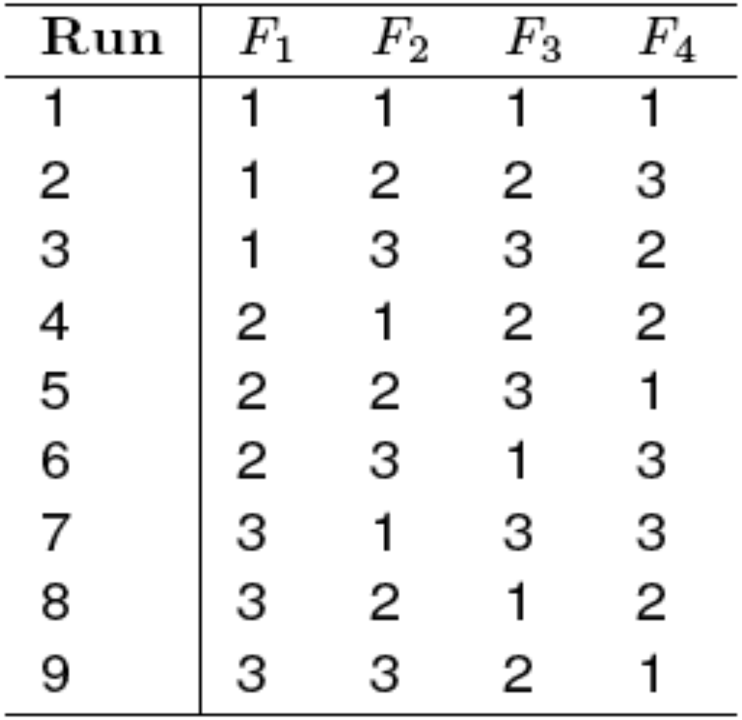
Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

λ $=4/2^2=1$ implying that each pair (t=2) appears exactly once (λ =1) in any 4 x 2 sub array. There is a total of $s^t=2^2=4$ pairs given as (1, 1), (1, 2), (2, 1), and (2, 2). It is easy to verify that each of the four pairs appears exactly once in each 4 x 2 sub array.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Orthogonal arrays: Another example



Run	F_1	F_2	F_3	F_4
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

What kind of an OA is this?

It has 9 runs and a strength of 2. Each of the four factors can be at any one of 3 levels. This array is denoted as OA(9, 4, 3, 2) and has an index of 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Orthogonal arrays: Alternate notations

$L_N(s^k)$ Orthogonal array of N runs where k factors take on any value from a set of s symbols.

Arrays shown earlier are $L_4(2^3)$ and $L_9(3^3)$

L_N denotes an orthogonal array of 9 runs. t, k, s are determined from the context, i.e. by examining the array itself.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.9. Mixed-level Orthogonal Arrays

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mixed level Orthogonal arrays

So far we have seen fixed level orthogonal arrays. This is because the design of such arrays assumes that all factors assume values from the same set of s values.

In many practical applications, one encounters more than one factor, each taking on a different set of values. Mixed orthogonal arrays are useful in designing test configurations for such applications.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mixed level Orthogonal arrays: Notation

$MA(N, s_1^{k_1} s_2^{k_2} \ldots s_p^{k_p}, t)$

Strength=t. Runs=N.

k1 factors at s1 levels, k2 at s2 levels, and so on.

Total factors: $\Sigma_{i=1}^{p} k_i$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mixed level Orthogonal arrays: Index and balance

The formula used for computing the index λ of an orthogonal array does not apply to the mixed level orthogonal array as the count of values for each factor is a variable.

The balance property of orthogonal arrays remains intact for mixed level orthogonal arrays in that any N x t sub array contains each t-tuple corresponding to the t columns, exactly the same number of times, which is λ.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

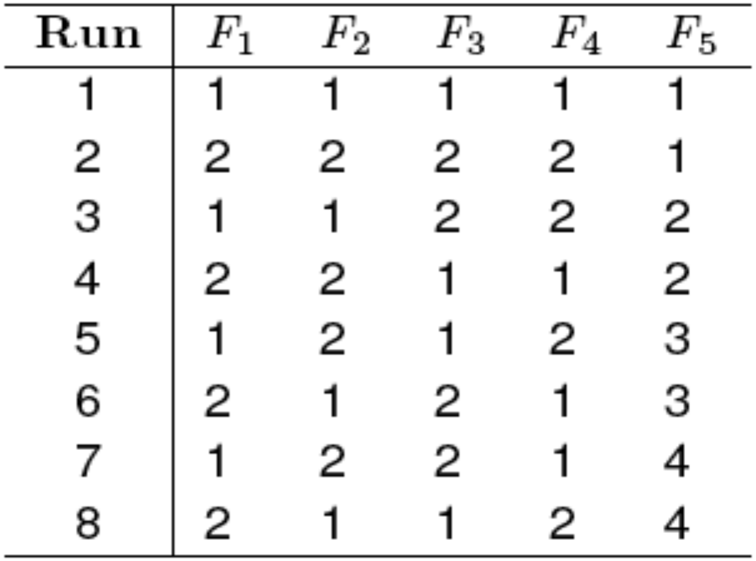
Contents

Mixed level Orthogonal arrays: Example

$$MA(8, 2^4 4^1, 2)$$

This array can be used to design test configurations for an application that contains 4 factors each at 2 levels and 1 factor at 4 levels.

Can you identify some properties?



Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	2	2	1
3	1	1	2	2	2
4	2	2	1	1	2
5	1	2	1	2	3
6	2	1	2	1	3
7	1	2	2	1	4
8	2	1	1	2	4

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

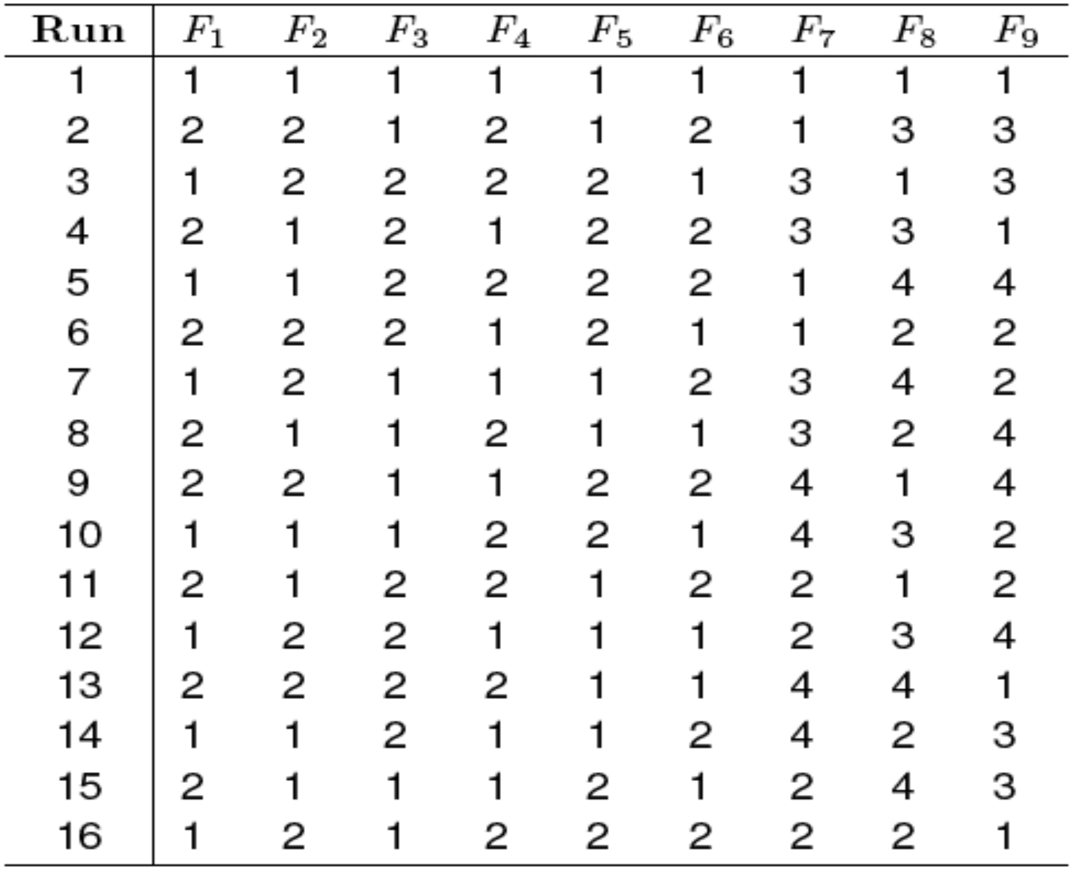
Balance: In any sub array of size 8 x 2, each possible pair occurs exactly the same number of times. In the two leftmost columns, each pair occurs exactly twice. In columns 1 and 3, each pair also occurs exactly twice. In columns 1 and 5, each pair occurs exactly once.

Contents

Mixed level Orthogonal arrays: Example

$MA(16, 2^6, 4^3, 2)$

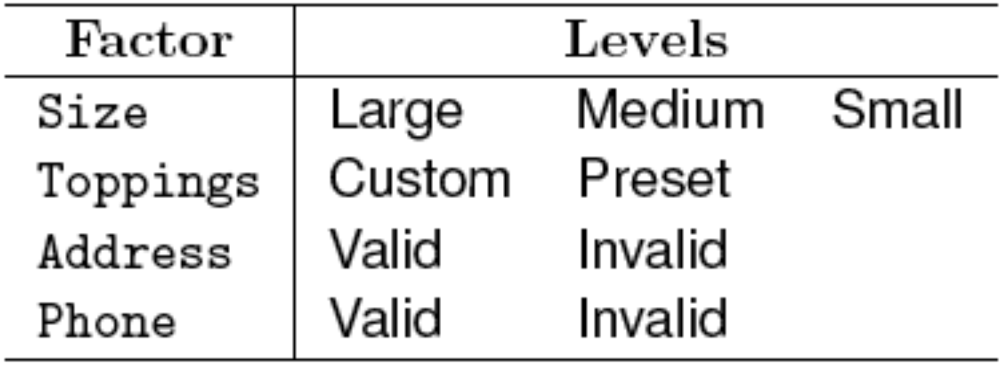
This array can be used to generate test configurations when there are six binary factors, labeled F_1 through F_6 and three factors each with four possible levels, labeled F_7 through F_9.



Run	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
1	1	1	1	1	1	1	1	1	1
2	2	2	1	2	1	2	1	3	3
3	1	2	2	2	2	1	3	1	3
4	2	1	2	1	2	2	3	3	1
5	1	1	2	2	2	2	1	4	4
6	2	2	2	1	2	1	1	2	2
7	1	2	1	1	1	2	3	4	2
8	2	1	1	2	1	1	3	2	4
9	2	2	1	1	2	2	4	1	4
10	1	1	1	2	2	1	4	3	2
11	2	1	2	2	1	2	2	1	2
12	1	2	2	1	1	1	2	3	4
13	2	2	2	2	1	1	4	4	1
14	1	1	2	1	1	2	4	2	3
15	2	1	1	1	2	1	2	4	3
16	1	2	1	2	2	2	2	2	1

Contents

Mixed level Orthogonal arrays: Test generation: Pizza delivery



Factor	Levels		
Size	Large	Medium	Small
Toppings	Custom	Preset	
Address	Valid	Invalid	
Phone	Valid	Invalid	

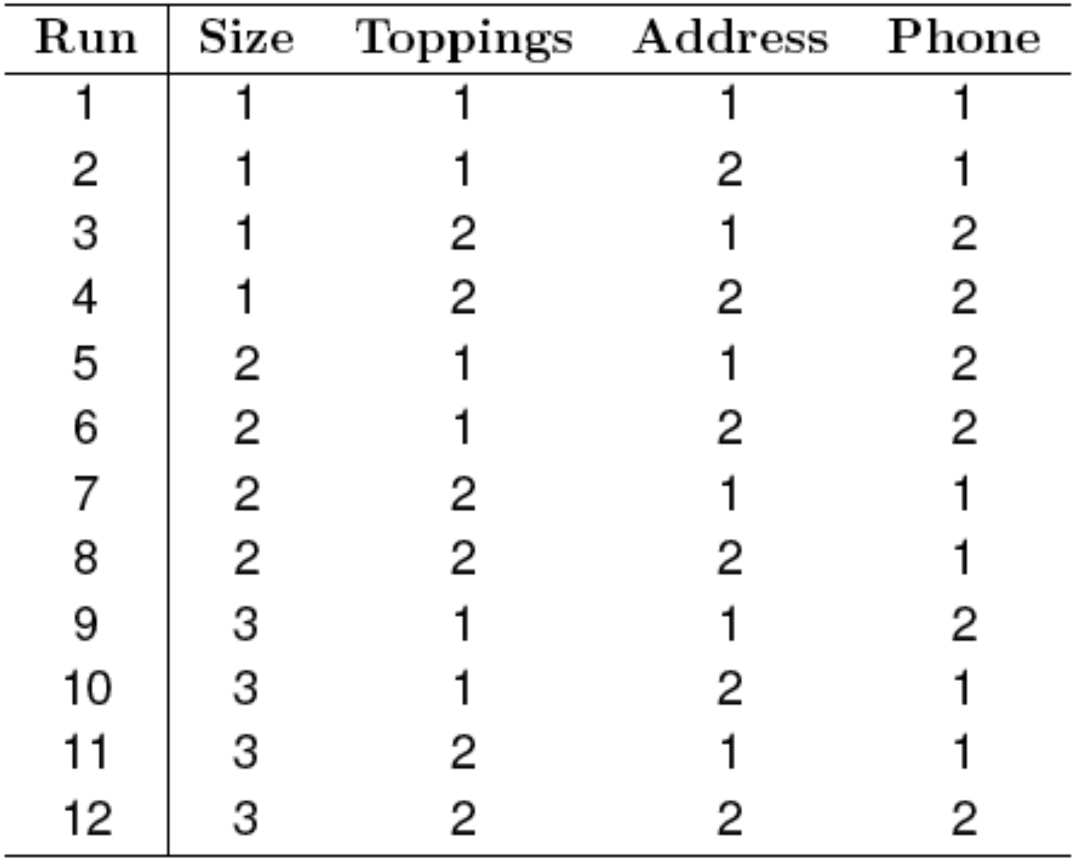
We have 3 binary factors and one factor at 3 levels. Hence we can use the following array to generate test configurations:

$$MA(12, 2^3, 3^1, 2)$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation: Pizza delivery: Array



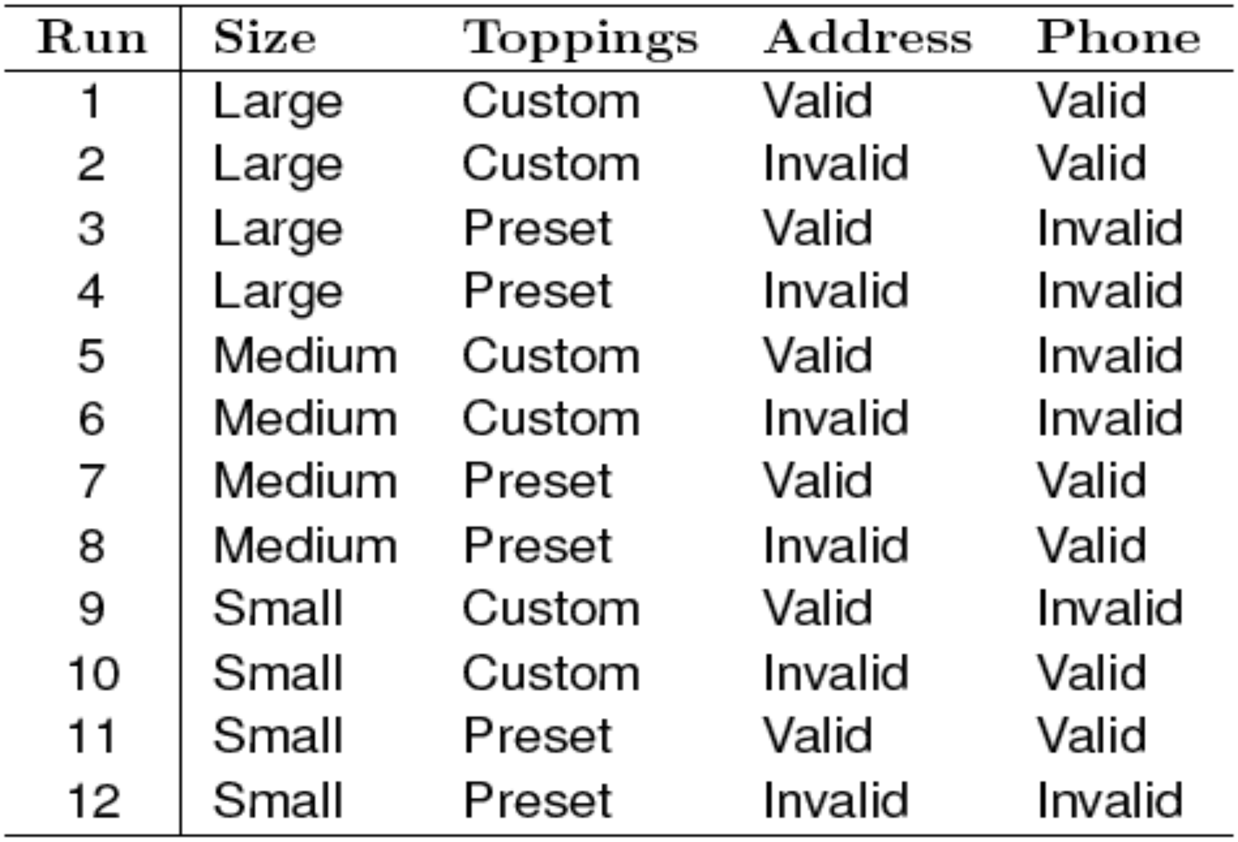
Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	1	1	2	1
3	1	2	1	2
4	1	2	2	2
5	2	1	1	2
6	2	1	2	2
7	2	2	1	1
8	2	2	2	1
9	3	1	1	2
10	3	1	2	1
11	3	2	1	1
12	3	2	2	2

Check that all possible pairs of factor combinations are covered in the design above. What kind of errors will likely be revealed when testing using these 12 configurations?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test generation: Pizza delivery: test configurations



Run	Size	Toppings	Address	Phone
1	Large	Custom	Valid	Valid
2	Large	Custom	Invalid	Valid
3	Large	Preset	Valid	Invalid
4	Large	Preset	Invalid	Invalid
5	Medium	Custom	Valid	Invalid
6	Medium	Custom	Invalid	Invalid
7	Medium	Preset	Valid	Valid
8	Medium	Preset	Invalid	Valid
9	Small	Custom	Valid	Invalid
10	Small	Custom	Invalid	Valid
11	Small	Preset	Valid	Valid
12	Small	Preset	Invalid	Invalid

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

6.9. Covering and mixed-level covering arrays

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

The "Balance" requirement

Observation [Dalal and Mallows, 1998]: The balance requirement is often essential in statistical experiments, it is not always so in software testing.

For example, if a software application has been tested once for a given pair of factor levels, there is generally no need for testing it again for the same pair, unless the application is known to behave non-deterministically.

For deterministic applications, and when repeatability is not the focus, we can relax the balance requirement and use covering arrays, or mixed level covering arrays for combinatorial designs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Covering array

A covering array CA(N, k, s, t) is an N x k matrix in which entries are from a finite set S of s symbols such that each N x t sub-array contains each possible t-tuple at least λ times.

N denotes the number of runs, k the number factors, s, the number of levels for each factor, t the strength, and λ the index

While generating test cases or test configurations for a software application, we use λ=1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Covering array and orthogonal array

While an orthogonal array OA(N, k, s, t) covers each possible t-tuple λ times in any N x t sub array, a covering array CA(N, k, s, t) covers each possible t-tuple at least λ times in any N x t sub array.

Thus, covering arrays do not meet the balance requirement that is met by orthogonal arrays. This difference leads to combinatorial designs that are often smaller in size than orthogonal arrays.

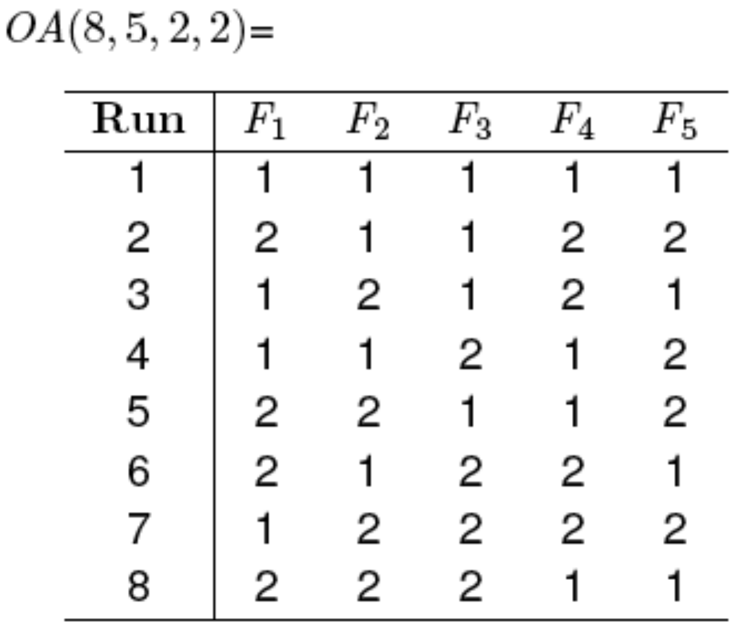
Covering arrays are also referred to as unbalanced designs. We are interested in minimal covering arrays.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

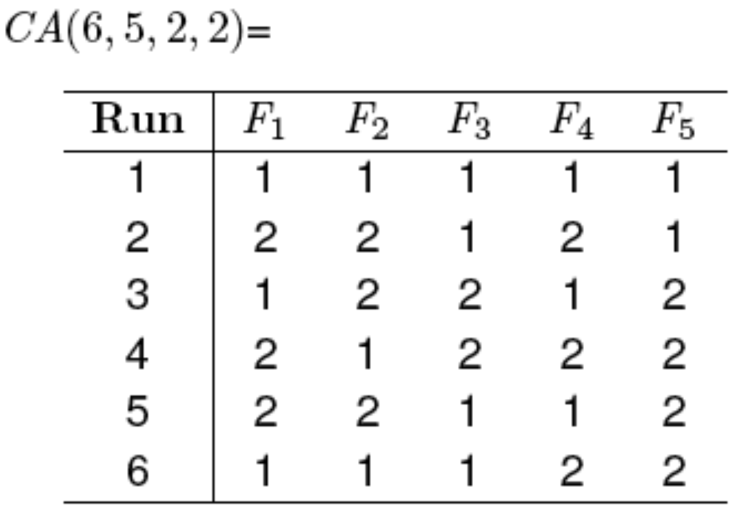
Covering array: Example

A balanced design of strength 2 for 5 binary factors, requires 8 runs and is denoted by OA(8, 5, 2, 2). However, a covering design with the same parameters requires only 6 runs.



$OA(8, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	1	1	2	2
3	1	2	1	2	1
4	1	1	2	1	2
5	2	2	1	1	2
6	2	1	2	2	1
7	1	2	2	2	2
8	2	2	2	1	1



$CA(6, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	1	2	1
3	1	2	2	1	2
4	2	1	2	2	2
5	2	2	1	1	2
6	1	1	1	2	2

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mixed level covering arrays

A mixed-level covering array is denoted as

$$MCA(N, s_1^{k_1} s_2^{k_2} \ldots s_p^{k_p}, t)$$

and refers to an N x Q matrix of entries such that, Q= $\sum_{i=1}^{p} k_i$ and each N x t sub-array contains at least one occurrence of each t-tuple corresponding to the t columns.

s1, s2,,… denote the number of levels of each the corresponding factor.

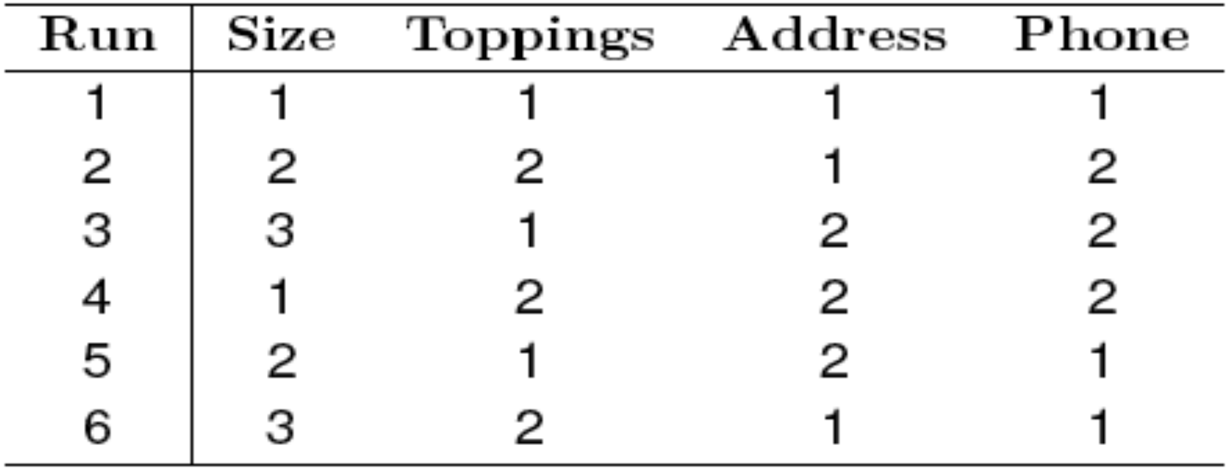
Mixed-level covering arrays are generally smaller than mixed-level orthogonal arrays and more appropriate for use in software testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mixed level covering array: Example

$MCA(6, 2^3 3^1, 2)$



Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	2	2	1	2
3	3	1	2	2
4	1	2	2	2
5	2	1	2	1
6	3	2	1	1

Comparing this with $MA(12, 2^3 3^1, 2)$ notice a reduction of 6 configurations.

Is the above array balanced?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

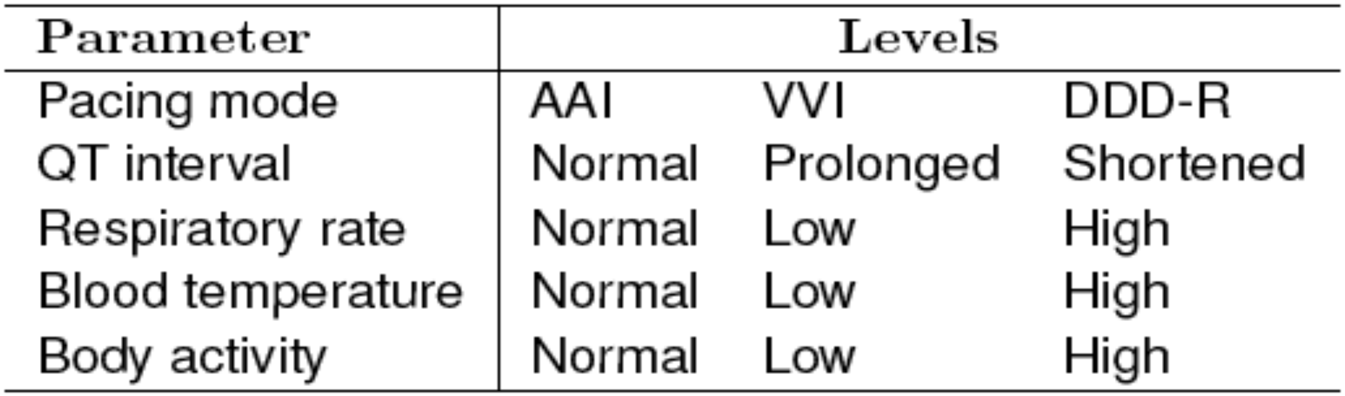
Contents

6.10. Arrays of strength >2

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Arrays of strength >2

Designs with strengths higher than 2 are sometimes needed to achieve higher confidence in the correctness of software. Consider the following factors in a pacemaker.



Parameter	Levels		
Pacing mode	AAI	VVI	DDD-R
QT interval	Normal	Prolonged	Shortened
Respiratory rate	Normal	Low	High
Blood temperature	Normal	Low	High
Body activity	Normal	Low	High

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Pacemaker example

Due to the high reliability requirement of the pacemaker, we would like to test it to ensure that there are no pairwise or 3-way interaction errors.

Thus, we need a suitable combinatorial object with strength 3. We could use an orthogonal array OA(54, 5, 3, 3) that has 54 runs for 5 factors each at 3 levels and is of strength 3. Thus, a total of 54 tests will be required to test for all 3-way interactions of the 5 pacemaker parameters

Could a design of strength 2 cover some triples and higher order tuples?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Generating mixed level covering arrays

We will now study a procedure due to Lei and Tai for the generation of mixed level covering arrays. The procedure is known as In-parameter Order (IPO) procedure.

Inputs: (a) n ≥2: Number of parameters (factors). (b) Number of values (levels) for each parameter.

Output: MCA

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

6.11. Generating covering arrays

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

IPO procedure

Consists of three steps:

Step 1: Main procedure.

Step 2: Horizontal growth.

Step 3: Vertical growth.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

IPO procedure: Example

Consider a program with three factors A, B, and C. A assumes values from the set {a1, a2, a3}, B from the set {b1, b2}, and C from the set {c1, c2, c3}. We want to generate a mixed level covering array for these three factors..

We begin by applying the Main procedure which is the first step in the generation of an MCA using the IPO procedure.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

IPO procedure: main procedure

Main: Step 1: Construct all runs that consist of pairs of values of the first two parameters. We obtain the following set.

$$\mathcal{T} = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$$

Let us denote the elements of $\mathcal{T}$ as t1, t2,…t6.

The entire IPO procedure would terminate at this point if the number of parameters n=2. In our case n=3 hence we continue with horizontal growth.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

IPO procedure: Horizontal growth

HG: Step 1: Compute the set of all pairs AP between parameters A and C, and parameters B and C. This leads us to the following set of fifteen pairs.

$$AP=\{(a_1,c_1),(a_1,c_2),(a_1,c_3),(a_2,c_1),(a_2,c_2),(a_2,c_3),(a_3,c_1),(a_3,c_2),(a_3,c_3)$$

$$(b_1,c_1),(b_1,c_2),(b_1,c_3),(b_2,c_1),(b_2,c_2),(b_2,c_3)\}$$

HG: Step 2: AP is the set of pairs yet to be covered. Let T’ denote the set of runs obtained by extending the runs in T. At this point T’ is empty as we have not extended any run in T.

Copyright © 2013 Doı ’ (India) Pvt. Ltd

Contents

Horizontal growth: Extend

HG: Steps 3, 4: Expand t1, t2, t3 by appending c1, c2, c3. This gives us:

t1’=(a1, b1, c1), t2’=(a1, b2, c2), and t3’=(a2, b1, c3)

Update T’ which now becomes {a1, b1, c1), (a1, b2, c2), (a2, b1, c3)}

Update pairs remaining to be covered AP={(a1, c3), (a2, c1), (a2, c2), (a3, c1), (a3, c2), (a3, c3), (b1, c2), (b2, c1), (b2, c3)}

Update T’ which becomes {(a1, b1, c1), (a1, b2, c2), (a2, b1, c3)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Horizontal growth: Optimal extension

HG. Step 5: We have not extended t4, t5, t6 as C does not have enough elements. We find the best way to extend these in the next step.

HG: Step 6: Expand t4, t5, t6 by suitably selected values of C.

If we extend t4=(a2, b2) by c1 then we cover two of the uncovered pairs from AP, namely, (a2, c1) and (b2, c1). If we extend it by c2 then we cover one pair from AP. If we extend it by c3 then we cover one pairs in AP. Thus, we choose to extend t4 by c1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Horizontal growth: Update and extend remaining

T' ={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1)}

AP= {(a1, c3), (a2, c2), (a3, c1), (a3, c2), (a3, c3), (b1, c2), (b2, c3)}

HG: Step 6: Similarly we extend t5 and t6 by the best possible values of parameter C. This leads to:

t5' =(a3, b1, c3) and t6' =(a3, b2, c1)

T' ={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1), (a3, b1, c3), (a3, b2, c1)}

AP= {(a1, c3), (a2, c2), (a3, c2), (b1, c2), (b2, c3)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Horizontal growth: Done

We have completed the horizontal growth step. However, we have five pairs remaining to be covered. These are:

AP= {(a1, c3), (a2, c2), (a3, c2), (b1, c2), (b2, c3)}

Also, we have generated six complete runs namely:

T' ={(a1, b1, c1), (a1, b2, c2), (a2, b1, c3), (a2, b2, c1), (a3, b1, c3), (a3, b2, c1)}

We now move to the vertical growth step of the main IPO procedure to cover the remaining pairs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Vertical growth

For each missing pair p from AP, we will add a new run to T' such that p is covered. Let us begin with the pair p= (a1, c3).

The run t= (a1, *, c3) covers pair p. Note that the value of parameter Y does not matter and hence is indicated as a * which denotes a don't care value.

Next , consider p=(a2, c2). This is covered by the run (a2, *, c2)

Next , consider p=(a3, c2). This is covered by the run (a3, *, c2)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Vertical growth (contd.)

Next , consider p=(b2, c3). We already have (a1, *, c3) and hence we can modify it to get the run (a1, b2, c3). Thus, p is covered without any new run added.

Finally, consider p=(b1, c2). We already have (a3, *, c2) and hence we can modify it to get the run (a3, b1, c2). Thus, p is covered without any new run added.

We replace the don’t care entries by an arbitrary value of the corresponding factor and get:

T={(a1, b1, c1), (a1, b2, c2), (a1, b1, c3), (a2, b1, c2), (a2, b2, c1), (a2, b2, c3), (a3, b1, c3), (a3, b2, c1), (a3, b1, c2)}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Final covering array

MCA(9, 2^1 3^2, 2)

Run	F1(X)	F2(Y)	F3(Z)
1	1	1	1
2	1	2	2
3	1	2	3
4	2	1	2
5	2	1	3
6	2	2	1
7	3	1	2
8	3	1	3
9	3	2	1

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Practicalities

That completes our presentation of an algorithm to generate covering arrays. A detailed analysis of the algorithm has been given by Lei and Tai.

Lei and Tai offer several other algorithms for horizontal and vertical growth that are faster than the algorithm mentioned here.

Lei and Tai found that the IPO algorithm performs almost as well as AETG in the size of the generated arrays.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Tools

AETG from Telcordia is a commercial tool to generate covering arrays. It allows users to specify constraints across parameters. For example, parameter A might not assume a value a2 when parameter B assumes value b3.

AETG is covered by US patent 5,542,043.

Publicly available tool: ACTS from Jeff Lie's group a UT Arlington.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary

Combinatorial design techniques assist with the design of test configurations and test cases. By requiring only pair-wise coverage and relaxing the “balance requirement,” combinatorial designs offer a significant reduction in the number of test configurations/test cases.

MOLS, Orthogonal arrays, covering arrays, and mixed-level covering arrays are used as combinatorial objects to generate test configurations/test cases. For software testing, most useful amongst these are mixed level covering arrays.

Handbooks offer a number covering and mixed level covering arrays. We introduced one algorithm for generating covering arrays. This continues to be a research topic of considerable interest.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 7

Test Adequacy Measurement and Enhancement: Control and Data flow

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 16, 2013

Contents

Learning Objectives

- What is test adequacy? What is test enhancement? When to measure test adequacy and how to use it to enhance tests?
- Control flow based test adequacy; statement, decision, condition, multiple condition, LCSAJ, and MC/DC coverage
- Data flow coverage
- Strengths and limitations of code coverage based measurement of test adequacy
- The "subsumes" relation amongst coverage criteria
- Tools for the measurement of code coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.1 Test adequacy: basics

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

What is adequacy?

- Consider a program P written to meet a set R of functional requirements. We notate such a P and R as (P, R). Let R contain n requirements labeled R1, R2,…, Rn .

- Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R . Also, P has been executed against each test in T and has produced correct behavior.

- We now ask: Is T good enough? This question can be stated differently as: Has P been tested thoroughly?, or as: Is T adequate?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Measurement of adequacy

- In the context of software testing, the terms ``thorough,"  ``good enough," and ``adequate," used in the questions above, have the same meaning.

- Adequacy is measured for a given test set designed to test P to determine whether or not P meets its requirements.

- This measurement is done against a given criterion C . A test set is considered adequate with respect to criterion C when it satisfies C. The determination of whether or not a test set T for program P satisfies criterion C depends on the criterion itself and is explained later.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example

Program sumProduct must meet the following requirements:

R1 Input two integers, say x and y , from the standard input device.

R2.1 Find and print to the standard output device the sum of x and y if $x<y$.

R2.2 Find and print to the standard output device the product of x and y if $x \geq y$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Suppose now that the test adequacy criterion C is specified as:

C : A test T for program (P, R) is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r .

Obviously, T={t: <x=2, y=3> is inadequate with respect to C for program sumProduct. The lone test case t in T tests R1 and R2.1, but not R2.2.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Black-box and white-box criteria

For each adequacy criterion C, we derive a finite set known as the coverage domain and denoted as Ce.

A criterion C is a white-box test adequacy criterion if the corresponding coverage domain Ce depends solely on program P under test.

A criterion C is a black-box test adequacy criterion if the corresponding coverage domain Ce depends solely on requirements R for the program P under test.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Coverage

We want to measure the adequacy of T. Given that Ce has n≥ 0 elements, we say that T covers Ce if for each element e' in Ce there is at least one test case in T that tests e'. The notion of "tests" is explained later through examples.

T is considered adequate with respect to C if it covers all elements in the coverage domain. T is considered inadequate with respect to C if it covers k elements of Ce where k<n .

The fraction k/n is a measure of the extent to which T is adequate with respect to C . This fraction is also known as the coverage of T with respect to C , P , and R .

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example

Let us again consider the following criterion: "A test T for program (P, R) is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r."

In this case the finite set of elements Ce={R1, R2.1, R2.2}. T covers R1 and R2.1 but not R2.2 . Hence T is not adequate with respect to C . The coverage of T with respect to C, P, and R is 0.66.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Another Example

Consider the following criterion: "A test T for program (P, R) is considered adequate if each path in P is traversed at least once."

Assume that P has exactly two paths, one corresponding to condition x<y and the other to x≥ y. We refer to these as p1 and p2, respectively. For the given adequacy criterion C we obtain the coverage domain Ce to be the set { p1, p2}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Another Example (contd.)

To measure the adequacy of T of sumProduct against C , we execute P against each test case in T .

As T contains only one test for which x<y , only the path p1 is executed. Thus, the coverage of T with respect to C, P , and R is 0.5 and hence T is not adequate with respect to C. We can also say that p1 is tested and p2 is not tested.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Code-based coverage domain

In the previous example we assumed that P contains exactly two paths. This assumption is based on a knowledge of the requirements. However, when the coverage domain must contain elements from the code, these elements must be derived by analyzing the code and not only by an examination of its requirements.

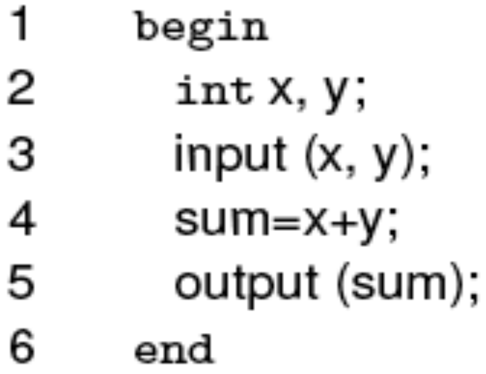
Errors in the program and incomplete or incorrect requirements might cause the program, and hence the coverage domain, to be different from the expected.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example

sumProduct1



```
begin
  int x, y;
  input (x, y);
  sum=x+y;
  output (sum);
end
```

This program is obviously incorrect as per the requirements of sumProduct.

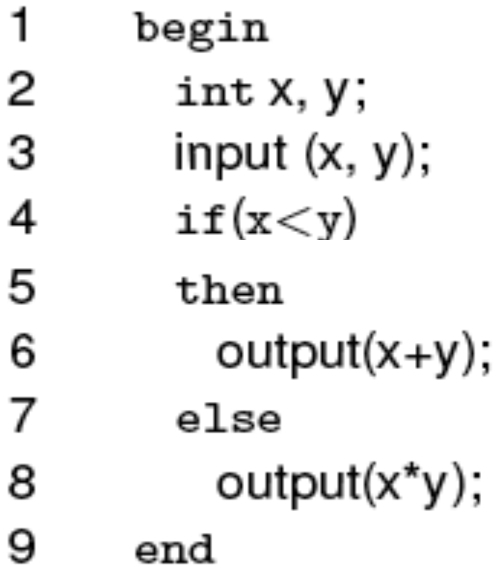
There is only one path denoted as p1. This path traverses all the statements. Using the path-based coverage criterion C, we get coverage domain Ce={ p1}. T={t: <x=2, y=3> }is adequate w.r.t. C but does not reveal the error.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

sumProduct2



```
begin
  int x, y;
  input (x, y);
  if(x<y)
  then
    output(x+y);
  else
    output(x*y);
end
```

This program is correct as per the requirements of sumProduct. It has two paths denoted by p1 and p2.

Ce={ p1, p2}. T={t: <x=2, y=3>} is inadequate w.r.t. the path-based coverage criterion C.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Lesson

An adequate test set might not reveal even the most obvious error in a program. This does not diminish in any way the need for the measurement of test adequacy as increasing coverage might reveal an error!.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.1.3 Test enhancement

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test Enhancement

While a test set adequate with respect to some criterion does not guarantee an error-free program, an inadequate test set is a cause for worry. Inadequacy with respect to any criterion often implies test deficiency.

Identification of this deficiency helps in the enhancement of the inadequate test set. Enhancement in turn is also likely to test the program in ways it has not been tested before such as testing untested portion, or testing the features in a sequence different from the one used previously. Testing the program differently than before raises the possibility of discovering any uncovered errors.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test Enhancement: Example

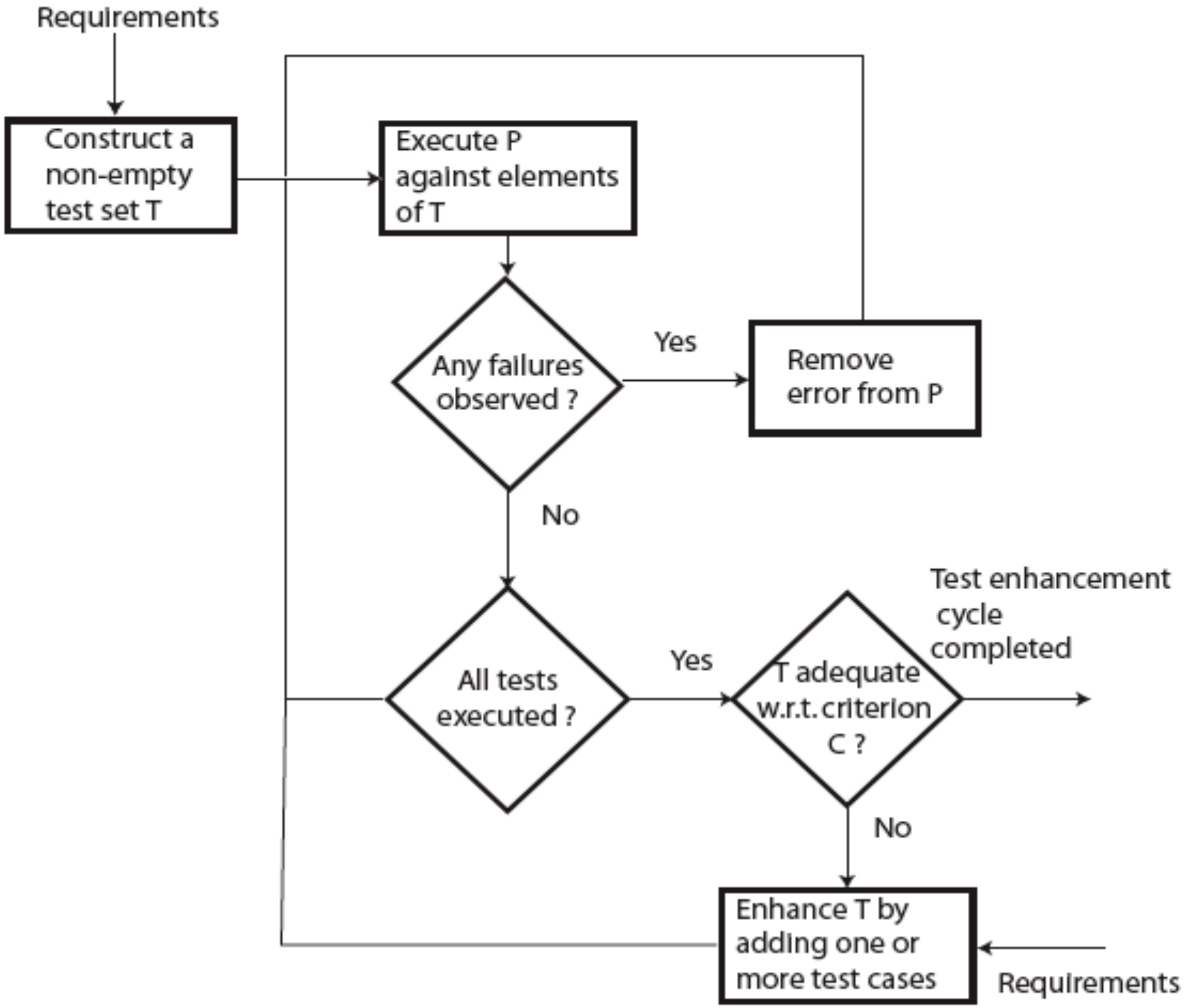
For sumProduct2, to make T adequate with respect to the path coverage criterion we need to add a test that covers p2. One test that does so is {<x=3>, y=1>}. Adding this test to T and denoting the expanded test set by T' we get:

T'={t1: <x=3, y=4>, t2: <x=3, y=1>}

Executing sum-product-2 against the two tests in T’ causes paths p1 and p2 are traversed. Thus, T' is adequate with respect to the path coverage criterion.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

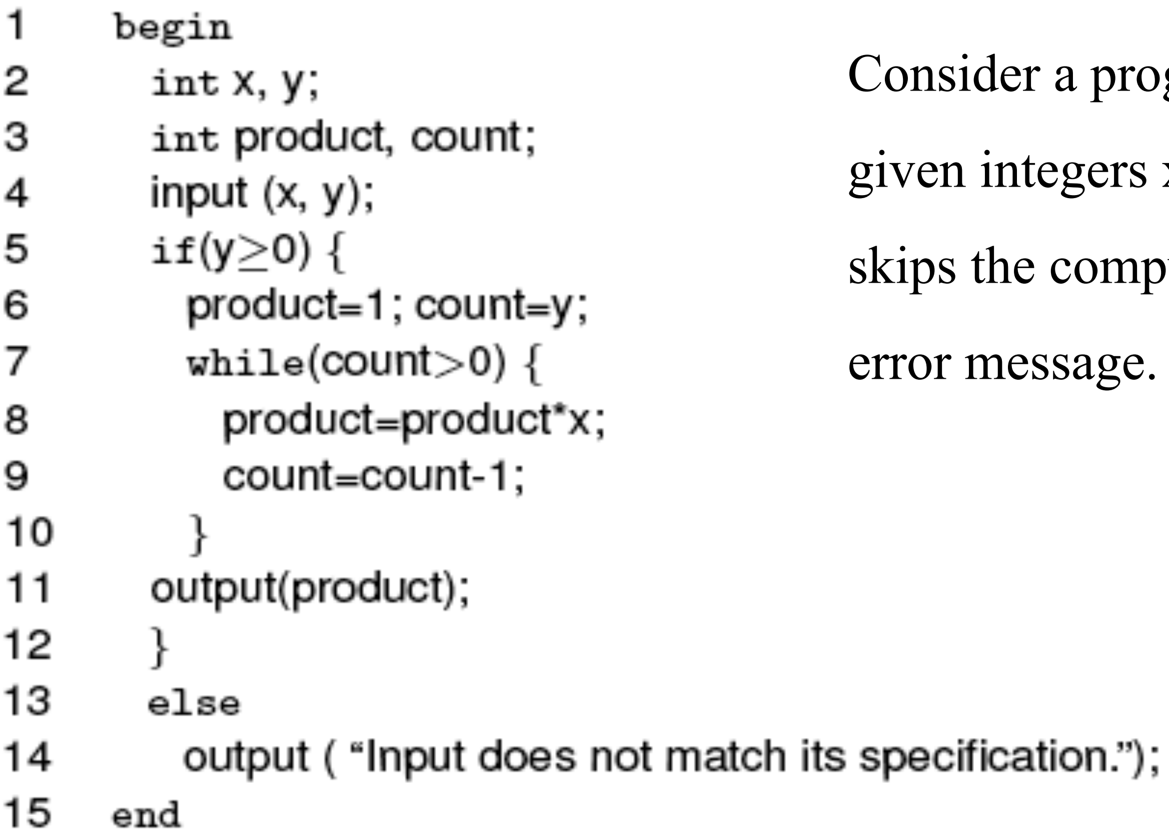
Test Enhancement: Procedure


Requirements
Construct a non-empty test set T
Execute P against elements of T
Any failures observed ?
Yes
Remove error from P
No
All tests executed ?
Yes
T adequate w.r.t. criterion C ?
Test enhancement cycle completed
No
Enhance T by adding one or more test cases
Requirements

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test Enhancement: Example



```
begin
  int x, y;
  int product, count;
  input (x, y);
  if(y≥0) {
    product=1; count=y;
    while(count>0) {
      product=product*x;
      count=count-1;
    }
  output(product);
  }
  else
    output ( “Input does not match its specification.”);
end
```

Consider a program intended to compute x^y given integers x and y. For y<0 the program skips the computation and outputs a suitable error message.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test Enhancement: Example (contd.)

Suppose that test T is considered adequate if it tests the exponentiation program for at least one zero and one non-zero value of each of the two inputs x and y.

The coverage domain for C can be determined using C alone and without any inspection of the program For C we get Ce={x=0, y=0}, x≠0, y≠ 0. Again, one can derive an adequate test set for the program by an examination of Ce. One such test set is

T={t1: <x=0, y=1>, t2: <x=1, y=0>}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test Enhancement: Example: Path coverage

Criterion C of the previous example is a black-box coverage criterion as it does not require an examination of the program under test for the measurement of adequacy

Let us now consider the path coverage criterion defined in in an earlier example. An examination of the exponentiation program reveals that it has an indeterminate number of paths due to the while loop. The number of paths depends on the value of y and hence that of count.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

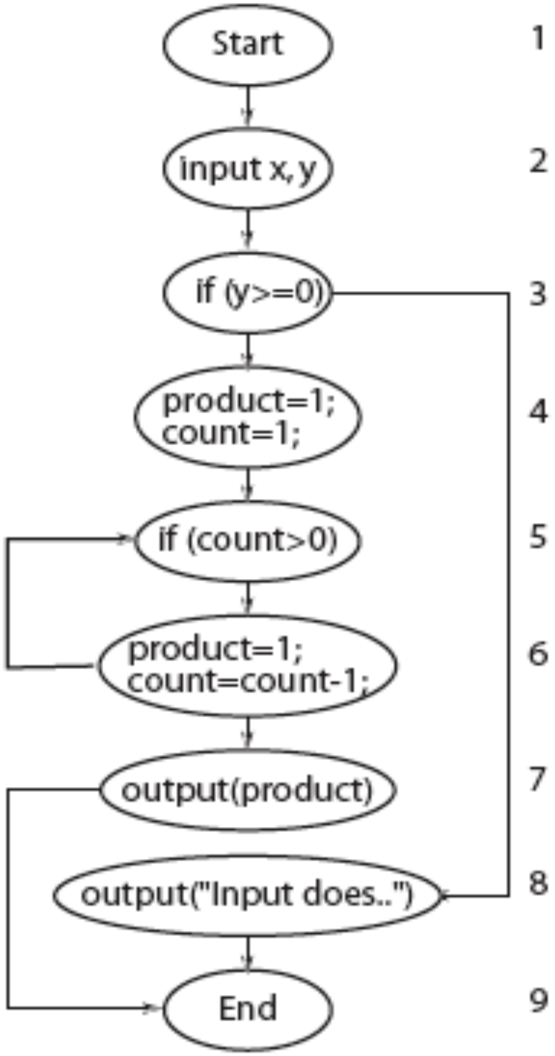
Example: Path coverage (contd.)

Given that y is any non-negative integer, the number of paths can be arbitrarily large. This simple analysis of paths in exponentiation reveals that for the path coverage criterion we cannot determine the coverage domain.

The usual approach in such cases is to simplify C and reformulate it as follows: *A test T is considered adequate if it tests all paths. In case the program contains a loop, then it is adequate to traverse the loop body zero times and once.*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: Path coverage (contd.)



The modified path coverage criterion leads to C'_e={p1, p2, p3}. The elements of C_e' are enumerated below with respect to flow graph for the exponentiation program.

p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9]$;

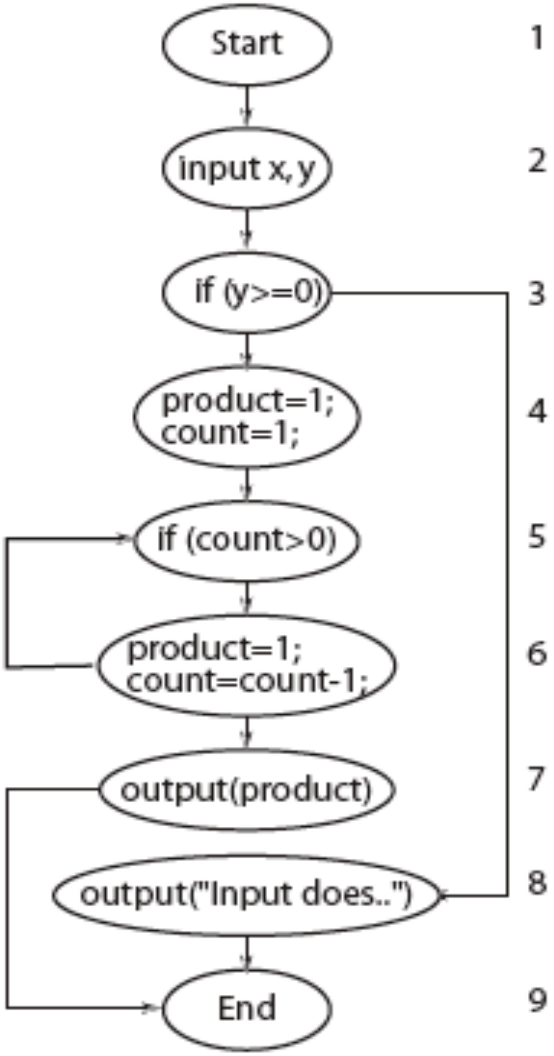
p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 6 \rightarrow 5 \rightarrow 9]$;

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 9]$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: Path coverage (contd.)



We measure the adequacy of T with respect to C'. As T does not contain any test with y<0, p3 remains uncovered. Thus, the coverage of T with respect to C' is 2/3=0.66.

p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9]$;

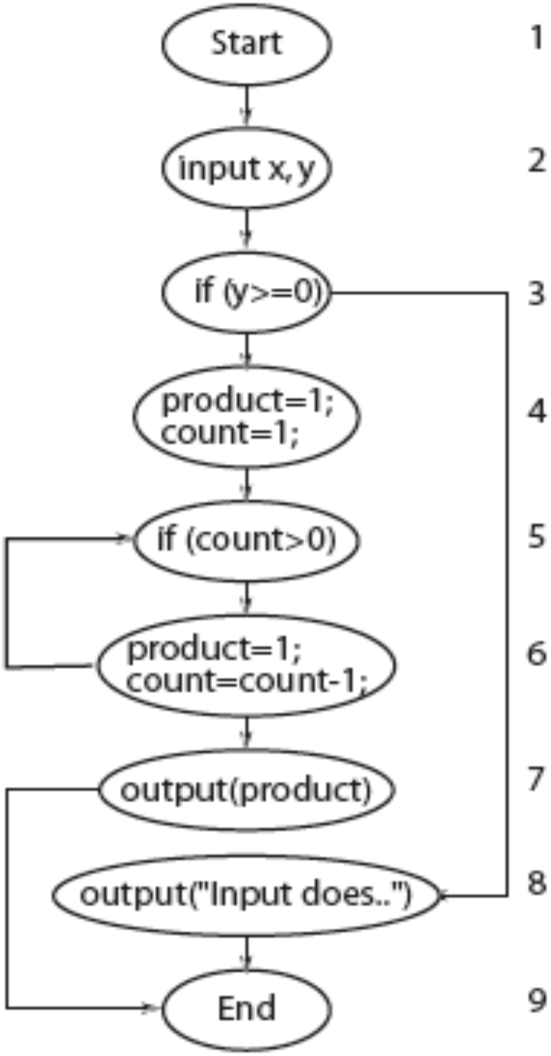
p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 6 \rightarrow 5 \rightarrow 9]$;

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 9]$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: Path coverage (contd.)



Any test case with y<0 will cause p3 to be traversed. Let us use t:<x=5, y=-1>. Test t covers path p3 and P behaves correctly. We add t to T. The loop in the enhancement terminates as we have covered all feasible elements of C_e'.

The enhanced test set is:

T={<x=0, y=1>, <x=1, y=0>, <x=5, y=-1>}

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.1.4 Infeasibility and test adequacy

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Infeasibility

An element of the coverage domain is infeasible if it cannot be covered by any test in the input domain of the program under test.

There does not exist an algorithm that would analyze a given program and determine if a given element in the coverage domain is infeasible or not. Thus, it is usually the tester who determines whether or not an element of the coverage domain is infeasible.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Demonstrating feasibility

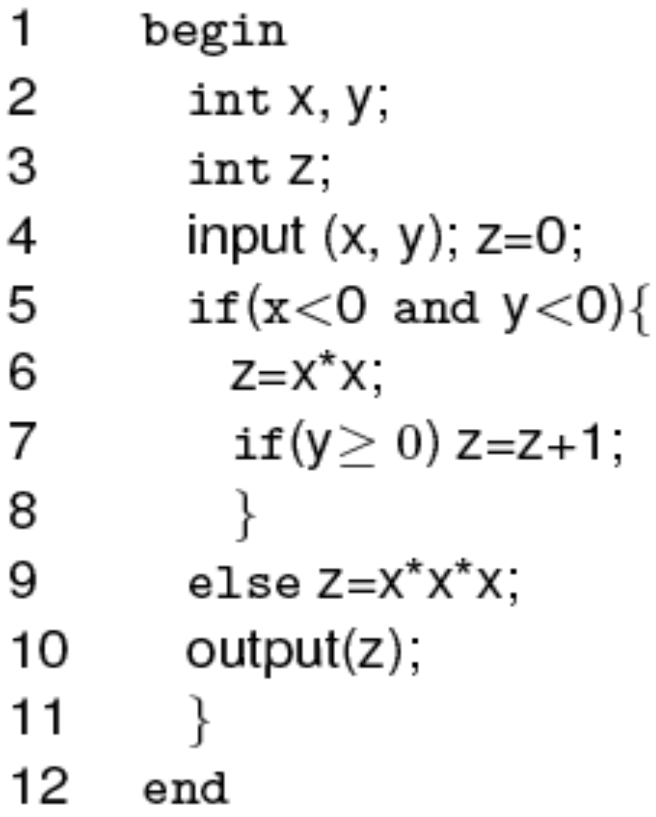
Feasibility can be demonstrated by executing the program under test against a test case and showing that indeed the element under consideration is covered.

Infeasibility cannot be demonstrated by program execution against a finite number of test cases. In some cases simple arguments can be constructed to show that a given element is infeasible. For complex programs the problem of determining infeasibility could be difficult. Thus, an attempt to enhance a test set by executing a test aimed at covering element e of program P, might fail.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Infeasible path: Example

This program inputs two integers x and y, and computes z. C_e={p1, p2, p3}.



```
1    begin
2      int x, y;
3      int z;
4      input (x, y); z=0;
5      if(x<0 and y<0){
6        z=x*x;
7        if(y≥ 0) z=z+1;
8        }
9      else z=x*x*x;
10     output(z);
11     }
12   end
```

p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

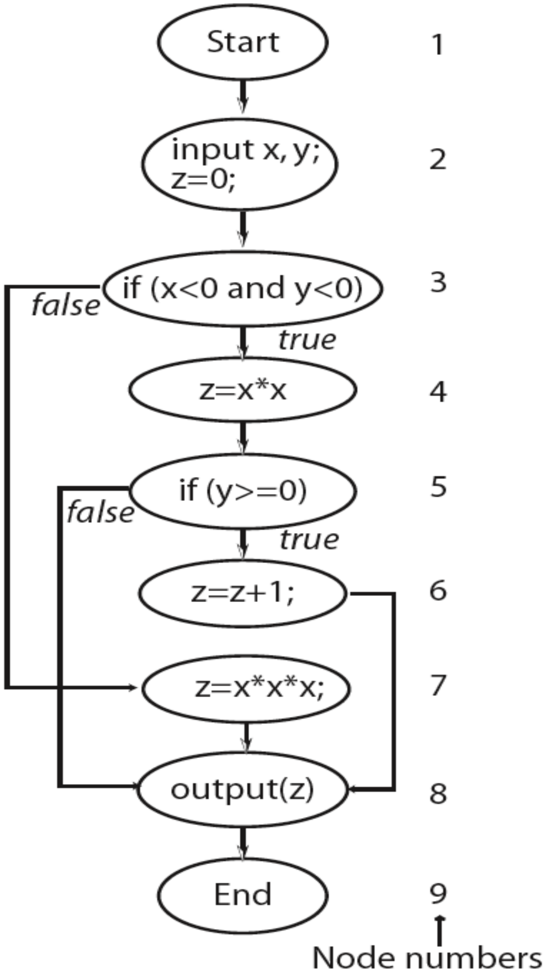
p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

Copyright © 2013 Dorling Kin ey (India) Pvt. Ltd

Contents

Example: Flow graph and paths



p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9]$

p1 is infeasible and cannot be traversed by any test case. This is because when control reaches node 5, condition y≥0 is false and hence control can never reach node 6.

Thus, any test adequate with respect to the path coverage criterion for the exponentiation program will only cover p2 and p3

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Adequacy and infeasibility

In the presence of one or more infeasible elements in the coverage domain, a test is considered adequate when all feasible elements in the domain have been covered.

While programmers might not be concerned with infeasible elements, testers attempting to obtain code coverage are. Prior to test enhancement, a tester usually does not know which elements of a coverage domain are infeasible. Unfortunately, it is only during an attempt to construct a test case to cover an element that one might realize the infeasibility of an element.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.1.5 Error detection and test enhancement

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test enhancement

The purpose of test enhancement is to determine test cases that test the untested parts of a program or exercise the program using uncovered portions of the input domain. Even the most carefully designed tests based exclusively on requirements can be enhanced.

The more complex the set of requirements, the more likely it is that a test set designed using requirements is inadequate with respect to even the simplest of various test adequacy criteria.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example

A program to meet the following requirements is to be developed.

R_1: Upon start the program offers the following three options to the user:

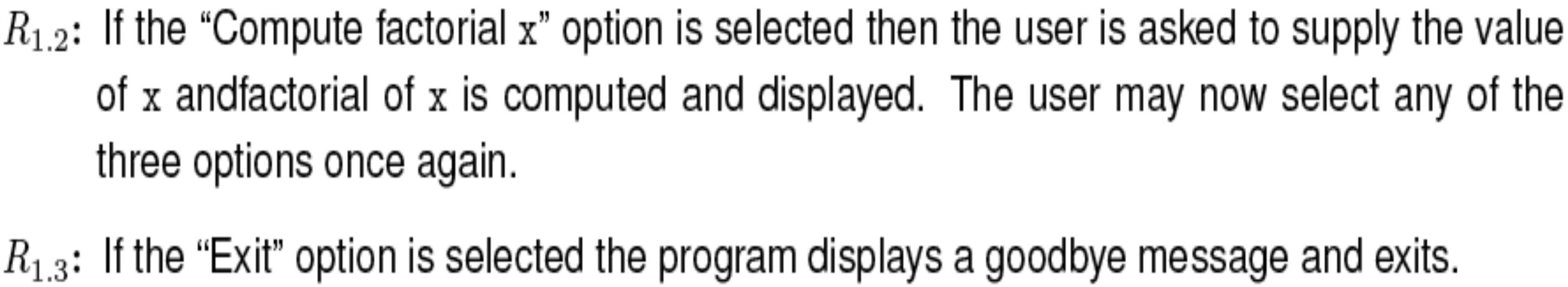
- Compute x^y for integers x and $\mathrm{y} \geq 0$.
- Compute the factorial of integer $x \geq 0$.
- Exit.

$R_{1.1}$: If the "Compute x^y" option is selected then the user is asked to supply the values of x and y, x^y is computed and displayed. The user may now select any of the three options once again.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)



$R_{1.2}$: If the “Compute factorial x” option is selected then the user is asked to supply the value of x andfactorial of x is computed and displayed. The user may now select any of the three options once again.

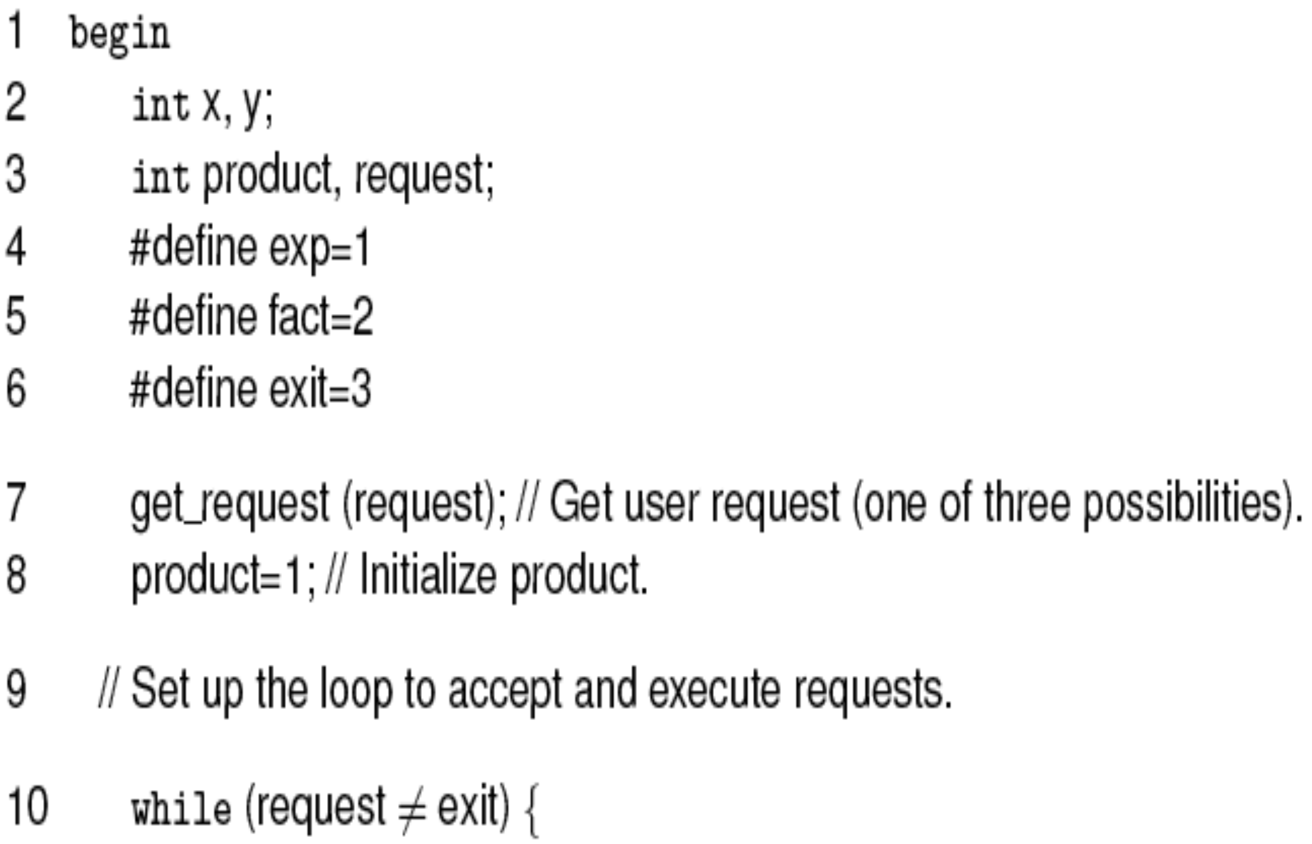
$R_{1.3}$: If the “Exit” option is selected the program displays a goodbye message and exits.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Consider the following program written to meet the requirements stated earlier.

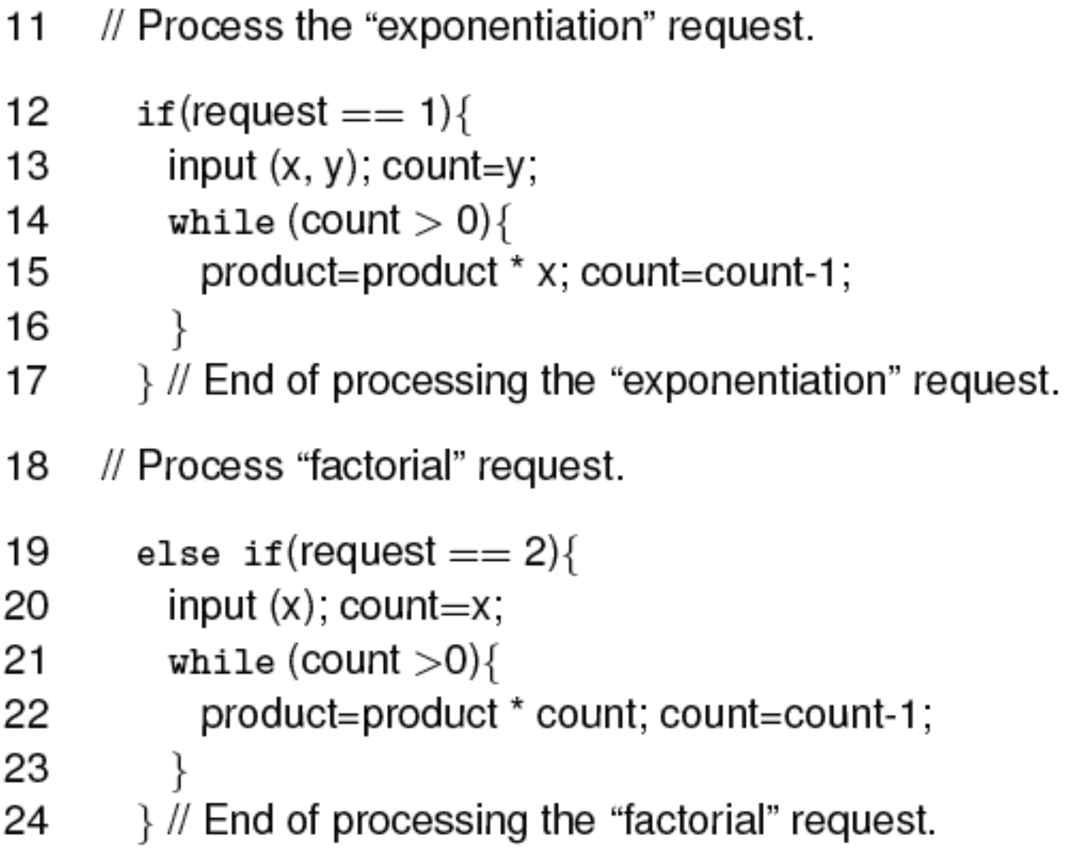


```
begin
    int x, y;
    int product, request;
    #define exp=1
    #define fact=2
    #define exit=3

    get_request (request); // Get user request (one of three possibilities).
    product=1; // Initialize product.

  // Set up the loop to accept and execute requests.

    while (request ≠ exit) {
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

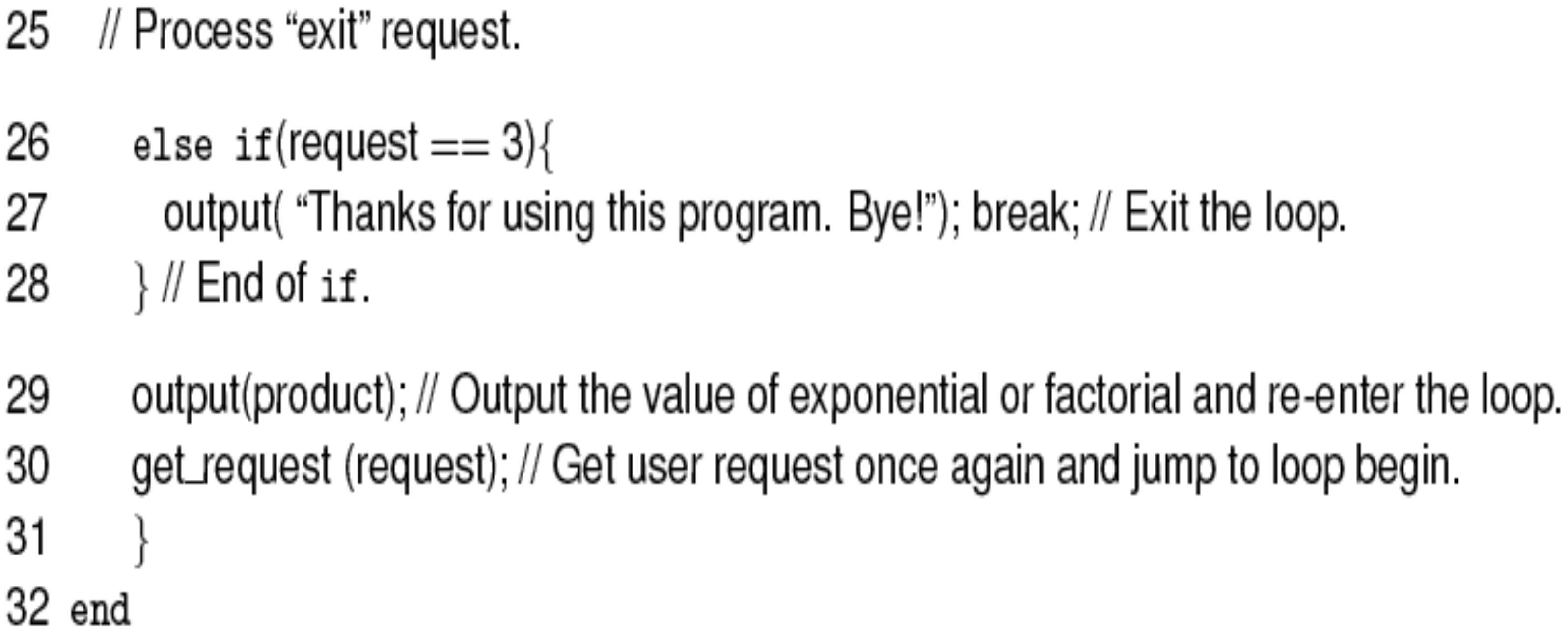


```
// Process the “exponentiation” request.
  if(request == 1){
    input (x, y); count=y;
    while (count > 0){
      product=product * x; count=count-1;
    }
  } // End of processing the “exponentiation” request.
// Process “factorial” request.
  else if(request == 2){
    input (x); count=x;
    while (count >0){
      product=product * count; count=count-1;
    }
  } // End of processing the “factorial” request.
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)



```
// Process "exit" request.

   else if(request == 3){
     output( "Thanks for using this program. Bye!"); break; // Exit the loop.
   } // End of if.

   output(product); // Output the value of exponential or factorial and re-enter the loop.
   get_request (request); // Get user request once again and jump to loop begin.
   }
end
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Suppose now that the following set containing three tests has been developed to test whether or not our program meets its requirements.

T={<request=1, x=2, y=3>, <request=2, x=4>, <request=3>}

For the first two of the three requests the program correctly outputs 8 and 24, respectively. The program exits when executed against the last request. This program behavior is correct and hence one might conclude that the program is correct. *It will not be difficult for you to believe that this conclusion is incorrect.*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Let us now evaluate T against the path coverage criterion.

In class exercise: Go back to the example program and extract the paths not covered by T.

The coverage domain consists of all paths that traverse each of the three loops zero and once in the same or different executions of the program. This is left as an exercise and we continue with one sample, and “tricky,” uncovered path.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Consider the path p that begins execution at line 1, reaches the outermost while at line 10, then the first if at line 12, followed by the statements that compute the factorial starting at line 20, and then the code to compute the exponential starting at line 13.

p is traversed when the program is launched and the first input request is to compute the factorial of a number, followed by a request to compute the exponential. It is easy to verify that the sequence of requests in T does not exercise p. Therefore T is inadequate with respect to the path coverage criterion.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

To cover p we construct the following test:

T' ={<request=2, x=4>, <request=1, x=2, y=3>, <request=3>}

When the values in T' are input to our example program in the sequence given, the program correctly outputs 24 as the factorial of 4 but incorrectly outputs 192 as the value of 2^3 .

This happens because T' traverses our "tricky" path which makes the computation of the exponentiation begin without initializing product. In fact the code at line 14 begins with the value of product set to 24.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

In our effort to increase the path coverage we constructed T'. Execution of the program under test on T' did cover a path that was not covered earlier and revealed an error in the program.

This example has illustrated a benefit of test enhancement based on code coverage.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.1.6 Single and multiple executions

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multiple executions

In the previous example we constructed two test sets T and T' . Notice that both T and T' contain three tests one for each value of variable request. Should T (or T') be considered a single test or a sequence of three tests?

T' ={<request=2, x=4>, <request=1, x=2, y=3>, <request=3>}

we assumed that all three tests, one for each value of request, are input in a sequence during a single execution of the test program. Hence we consider T as a test set containing one test case and write it as follows:

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multiple executions (contd.)

We assumed that all three tests, one for each value of request, are input in a sequence during a single execution of the test program. Hence we consider T as a test set containing one test case and write it, it as follows:

$$T = \left\{ \begin{array}{lll} t_1: & << request = 1, x = 2, y = 3 > & \rightarrow \\ & < request = 2, x = 4 > & \rightarrow \quad < request = 3 >> \end{array} \right\}$$

T”=T∪T’

$$T'' = \left\{ \begin{array}{llll} t_1: & << request = 1, x = 2, y = 3 > & \rightarrow & < request = 2, x = 4 > \quad \rightarrow \\ & < request = 3 >> & & \\ t_2: & << request = 2, x = 4 > & \rightarrow & \\ & < request = 1, x = 2, y = 3 > & \rightarrow & < request = 3 >> \end{array} \right\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.1 Statement and block coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Declarations and basic blocks

Any program written in a procedural language consists of a sequence of statements. Some of these statements are declarative, such as the #define and int statements in C, while others are executable, such as the assignment, if, and while statements in C and Java.

Recall that a basic block is a sequence of consecutive statements that has exactly one entry point and one exit point. For any procedural language, adequacy with respect to the statement coverage and block coverage criteria are defined next.

Notation: (P, R) denotes program P subject to requirement R.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Statement coverage

The statement coverage of T with respect to (P, R) is computed as Sc/(Se-Si) , where Sc is the number of statements covered, Si is the number of unreachable statements, and Se is the total number of statements in the program, i.e. the size of the coverage domain.

T is considered adequate with respect to the statement coverage criterion if the statement coverage of T with respect to (P, R) is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Block coverage

The block coverage of T with respect to (P, R) is computed as Bc/(Be -Bi) , where Bc is the number of blocks covered, Bi is the number of unreachable blocks, and Be is the total number of blocks in the program, i.e. the size of the block coverage domain.

T is considered adequate with respect to the block coverage criterion if the statement coverage of T with respect to (P, R) is 1.

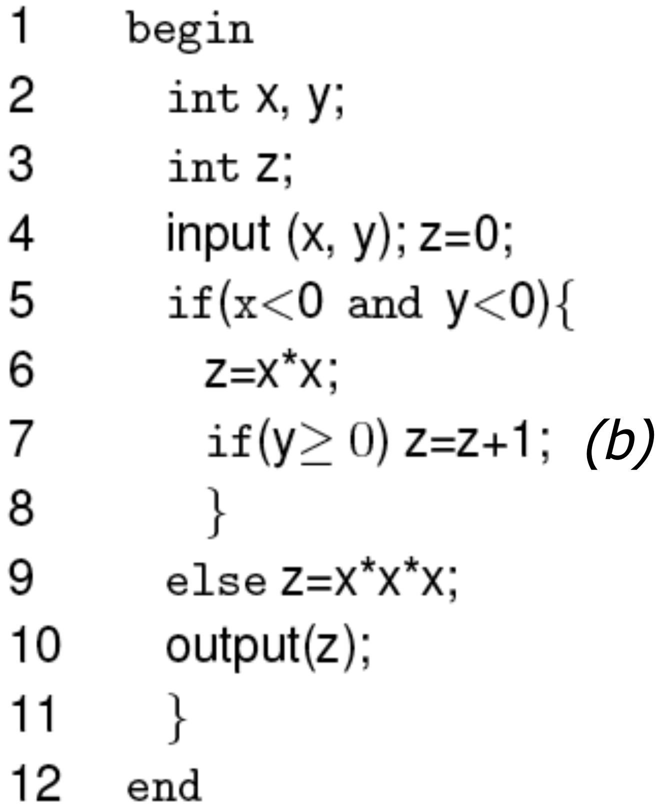
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: statement coverage

Coverage domain: Se={2, 3, 4, 5, 6, 7, 7b, 9, 10}

Let T_1={t1:<x=-1, y=-1>,t 2:<x=1, y=1>}



```
1    begin
2      int x, y;
3      int z;
4      input (x, y); z=0;
5      if(x<0 and y<0){
6        z=x*x;
7        if(y≥ 0) z=z+1;
8        }
9      else z=x*x*x;
10     output(z);
11     }
12   end
```

(b)

Statements covered:

t1: 2, 3, 4, 5, 6, 7, and 10

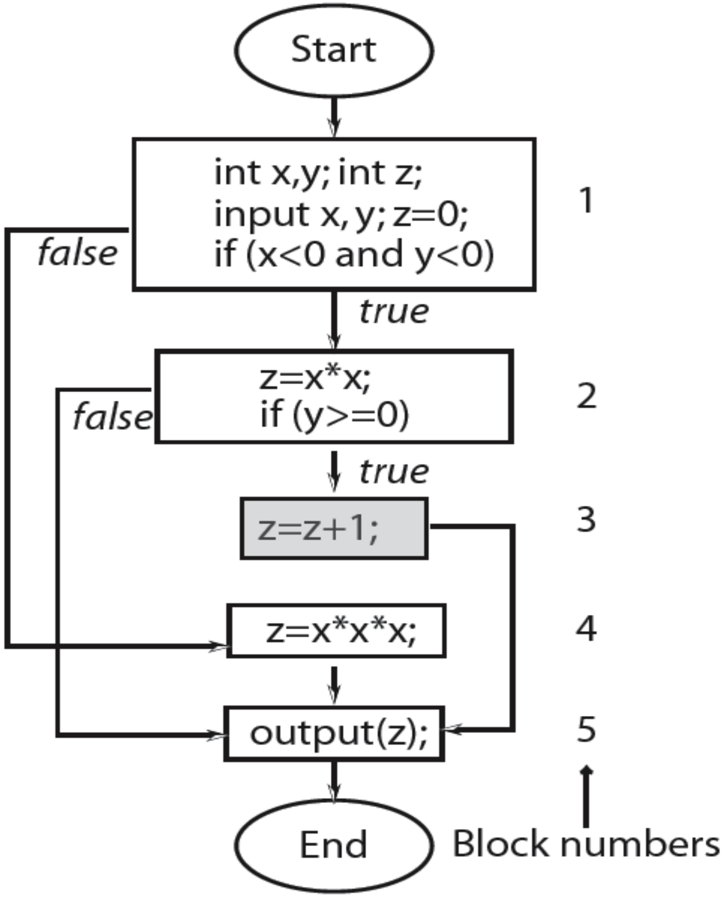
t2: 2, 3, 4, 9, and 10.

Sc=6, Si=1, Se=7. The statement coverage for T is 6/(7-1)=1 . Hence we conclude that T_1 is adequate for (P, R) with respect to the statement coverage criterion. Note: 7b is unreachable.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example: block coverage



Coverage domain: Be={1, 2, 3, 4, 5

$$T_2 = \left\{ \begin{array}{llll} t_1: & < x = -1 & y = -1 > \\ t_2: & < x = -3 & y = -1 > \\ t_3: & < x = -1 & y = -3 > \end{array} \right\}$$

Blocks covered:

t1: Blocks 1, 2, 5

t2, t3: same coverage as of t1.

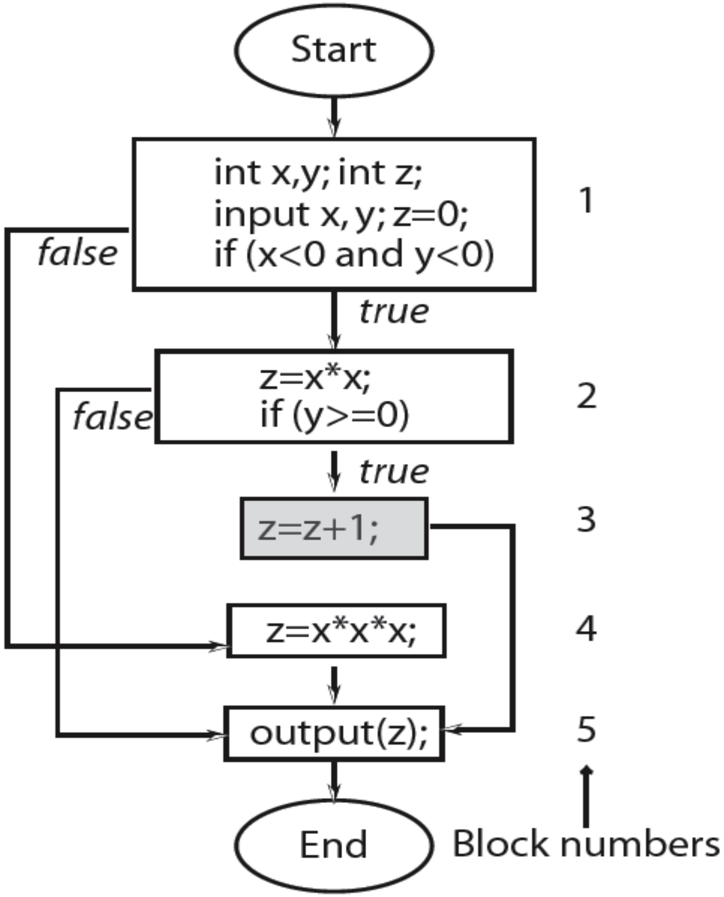
Be=5 , Bc=3, Bi=1.

Block coverage for T_2= 3/(5-1)=0.75.

Hence T_2 is not adequate for (P, R) with respect to the block coverage criterion.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: block coverage (contd.)



T1 is adequate w.r.t. block coverage criterion. In class exercise: Verify this statement!

Also, if test t2 in T1 is added to T2, we obtain a test set adequate with respect to the block coverage criterion for the program under consideration. In class exercise: Verify this statement!

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Coverage values

The formulae given for computing various types of code coverage, yield a coverage value between 0 and 1. However, while specifying a coverage value, one might instead use percentages. For example, a statement coverage of 0.65 is the same as 65% statement coverage.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.2 Conditions and decisions

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Conditions

Any expression that evaluates to true or false constitutes a condition. Such an expression is also known as a predicate.

Given that A , and B are Boolean variables, and x and y are integers, A , x > y , A OR B , A AND (x<y), (A AND B), are all sample conditions.

Note that in programming language C, x and x+y are valid conditions, and the constants 1 and 0 correspond to, respectively, **true** and **false**.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Simple and compound conditions

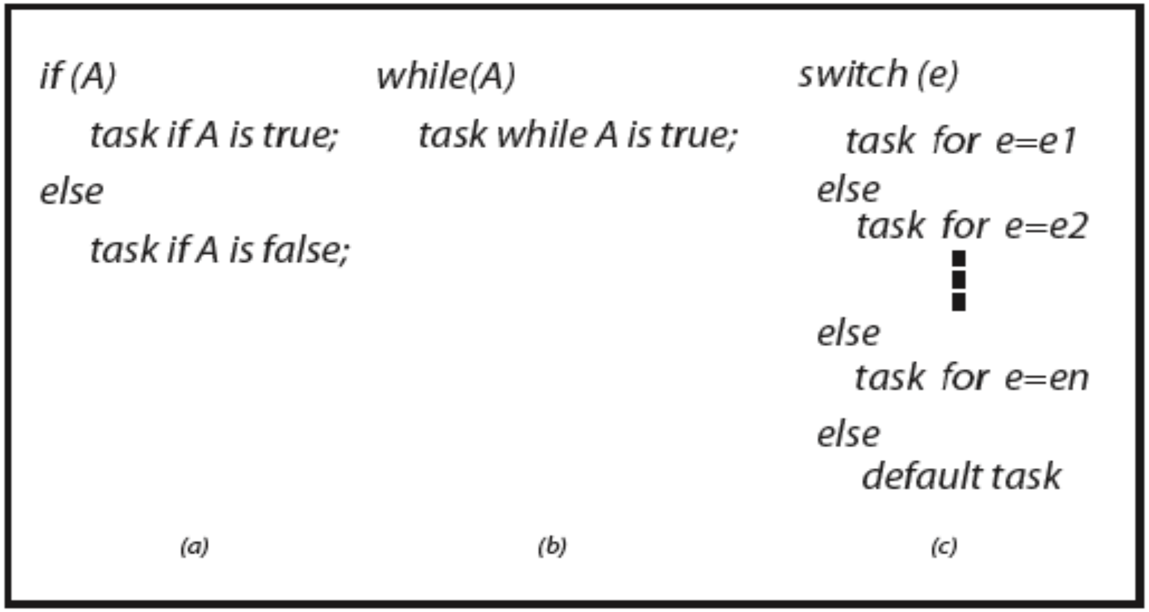
A simple condition does not use any Boolean operators except for the not operator. It is made up of variables and at most one relational operator from the set {<, ≤ >, ≥, ==, ≠ }. Simple conditions are also referred to as atomic or elementary conditions because they cannot be parsed any further into two or more conditions.

A compound condition is made up of two or more simple conditions joined by one or more Boolean operators.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Conditions as decisions

Any condition can serve as a decision in an appropriate context within a program. most high level languages provide **if**, **while**, and **switch** statements to serve as contexts for decisions.



```
if (A)
    task if A is true;
else
    task if A is false;
```

(a)

```
while(A)
    task while A is true;
```

(b)

```
switch (e)
    task for e=e1
  else
    task for e=e2
       ⋮
  else
    task for e=en
  else
    default task
```

(c)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Outcomes of a decision

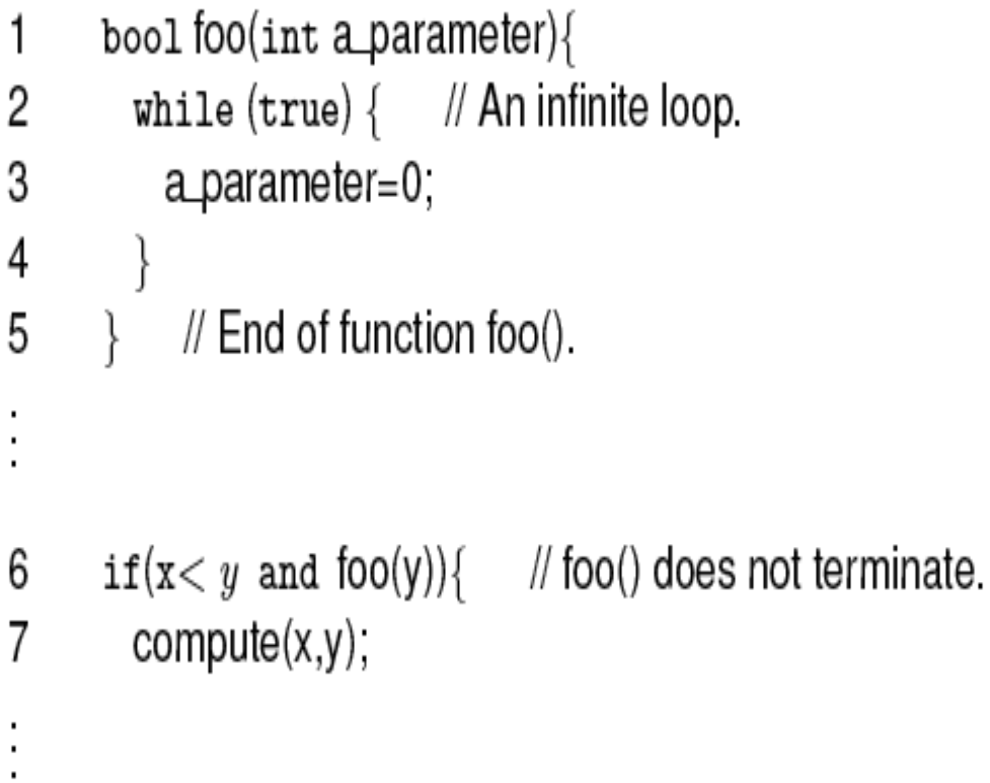
A decision can have three possible outcomes, true, false, and undefined. When the condition corresponding to a decision to take one or the other path is taken.

In some cases the evaluation of a condition might fail in which case the corresponding decision's outcome is undefined.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Undefined condition



```
1   bool foo(int a_parameter){
2     while (true) {      // An infinite loop.
3       a_parameter=0;
4     }
5   }     // End of function foo().
⋮

6   if(x< y and foo(y)){      // foo() does not terminate.
7     compute(x,y);
⋮
```

The condition inside the if statement at line 6 will remain undefined because the loop at lines 2-4 will never terminate. Thus, the decision at line 6 evaluates to undefined.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Coupled conditions

How many simple conditions are there in the compound condition: Cond=(A AND B) OR (C AND A)? The first occurrence of A is said to be coupled to its second occurrence.

Does Cond contain three or four simple conditions? Both answers are correct depending on one's point of view. Indeed, there are three distinct conditions A, B, and C. The answer is four when one is interested in the number of occurrences of simple conditions in a compound condition.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

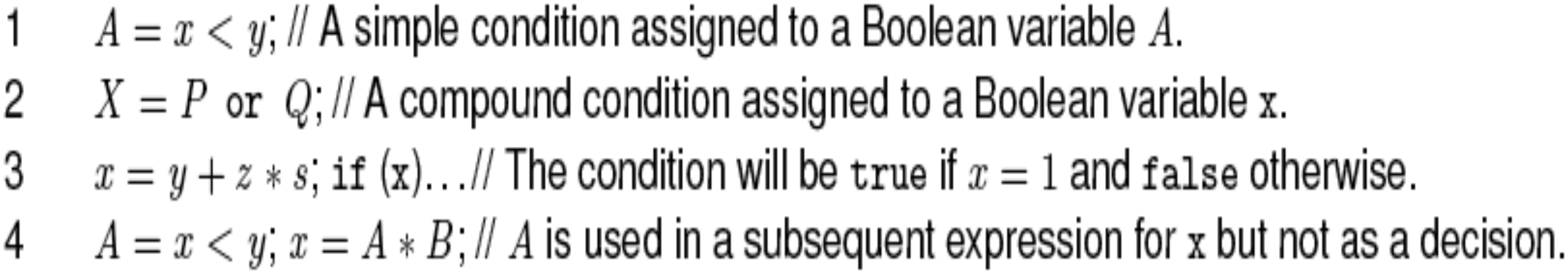
7.2.3 Decision coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Conditions within assignments

Strictly speaking, a condition becomes a decision only when it is used in the appropriate context such as within an if statement.

At line 4, x<y does not constitute a decision and neither does A*B.



1 $A = x < y$; // A simple condition assigned to a Boolean variable A.

2 $X = P$ or Q; // A compound condition assigned to a Boolean variable x.

3 $x = y + z * s$; if (x)...// The condition will be true if $x = 1$ and false otherwise.

4 $A = x < y$; $x = A * B$; // A is used in a subsequent expression for x but not as a decision.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision coverage

A decision is considered covered if the flow of control has been diverted to all possible destinations that correspond to this decision, i.e. all outcomes of the decision have been taken.

This implies that, for example, the expression in the if or a while statement has evaluated to true in some execution of the program under test and to false in the same or another execution.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Decision coverage: switch statement

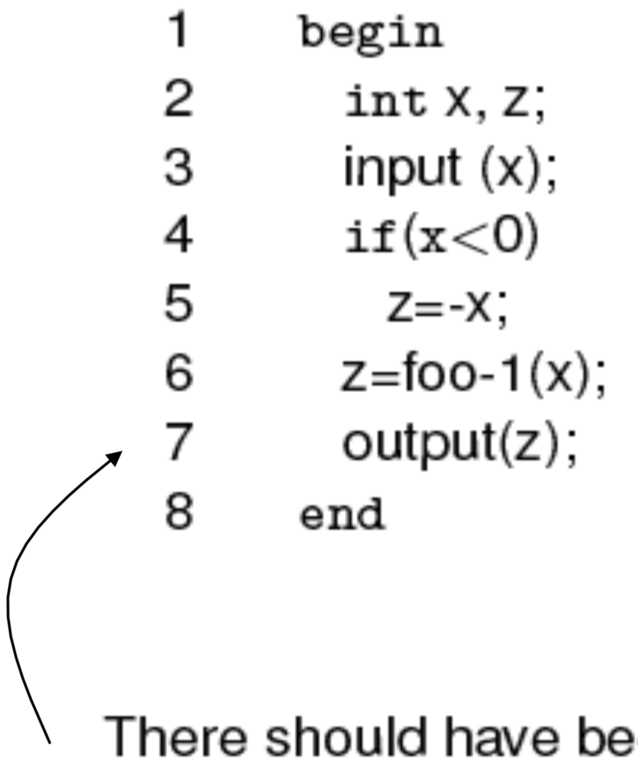
A decision implied by the switch statement is considered covered if during one or more executions of the program under test the flow of control has been diverted to all possible destinations.

Covering a decision within a program might reveal an error that is not revealed by covering all statements and all blocks.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision coverage: Example



```
begin
  int x, z;
  input (x);
  if(x<0)
    z=-x;
  z=foo-1(x);
  output(z);
end
```

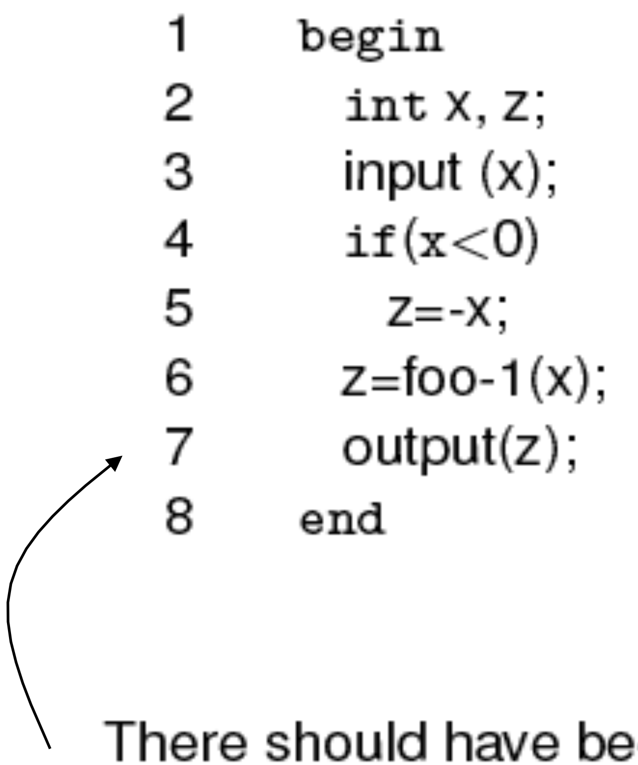
This program inputs an integer x, and if necessary, transforms it into a positive value before invoking foo-1 to compute the output z. The program has an error. As per its requirements, the program is supposed to compute z using foo-2 when x≥0.

There should have been an else clause before this statement.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision coverage: Example (contd.)



```
begin
  int x, z;
  input (x);
  if(x<0)
    z=-x;
  z=foo-1(x);
  output(z);
end
```

There should have been an else clause before this statement.

Consider the test set T={t1:<x=-5>}. It is adequate with respect to statement and block coverage criteria, but does not reveal the error.

Another test set T'={t1:<x=-5> t2:<x=3>} does reveal the error. It covers the decision whereas T does not. Check!

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision coverage: Computation

The previous example illustrates how and why decision coverage might help in revealing an error that is not revealed by a test set adequate with respect to statement and block coverage.

The decision coverage of T with respect to (P, R) is computed as Dc/(De -Di) , where Dc is the number of decisions covered, Di is the number of infeasible decisions, and De is the total number of decisions in the program, i.e. the size of the decision coverage domain.

T is considered adequate with respect to the decisions coverage criterion if the decision coverage of T with respect to (P, R) is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision coverage: domain

The domain of decision coverage consists of all decisions in the program under test.

Note that each if and each while contribute to one decision whereas a switch contribute to more than one.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.4 Condition coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Condition coverage

A decision can be composed of a simple condition such as $x<0$, or of a more complex condition, such as (($x<0$ AND $y<0$) OR ($p\geq q$)).

AND, OR, XOR are the logical operators that connect two or more simple conditions to form a compound condition.

A simple condition is considered covered if it evaluates to true and false in one or more executions of the program in which it occurs. A compound condition is considered covered if each simple condition it is comprised of is also covered.

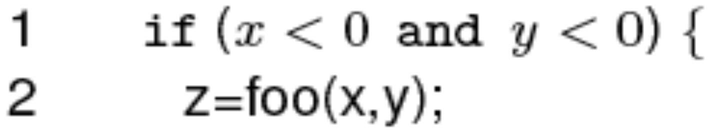
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.2.5 Condition/decision coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Decision and condition coverage

Decision coverage is concerned with the coverage of decisions regardless of whether or not a decision corresponds to a simple or a compound condition. Thus, in the statement



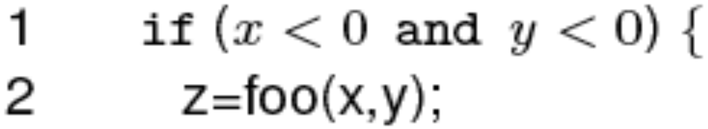
```
1     if (x < 0 and y < 0) {
2       z=foo(x,y);
```

there is only one decision that leads control to line 2 if the compound condition inside the if evaluates to true. However, a compound condition might evaluate to true or false in one of several ways.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision and condition coverage (contd)



```
1    if (x < 0 and y < 0) {
2      z=foo(x,y);
```

The condition at line 1 evaluates to false when x≥0 regardless of the value of y. Another condition, such as x<0 OR y<0, evaluates to true regardless of the value of y, when x<0.

With this evaluation characteristic in view, compilers often generate code that uses short circuit evaluation of compound conditions.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Decision and condition coverage (contd)

Here is a possible translation:

```
if (x < 0 and y < 0) {
  z=foo(x,y);
```

→

```
if (x<0)
  if (y<0)
    z=foo(x,y);
```

We now see two decisions, one corresponding to each simple condition in the if statement.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Condition coverage

The condition coverage of T with respect to (P, R) is computed as Cc/(Ce -Ci) , where Cc is the number of simple conditions covered, Ci is the number of infeasible simple conditions, and |Ce is the total number of simple conditions in the program, i.e. the size of the condition coverage domain.

T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Condition coverage: alternate formula

An alternate formula where each simple condition contributes 2, 1, or 0 to Cc depending on whether it is covered, partially covered, or not covered, respectively. is:

$$\frac{C_c}{2 \times (C_e - C_i)}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Condition coverage: Example

```
begin
  int x, y, z;
  input (x, y);
  if(x<0 and y<0)
    z=foo1(x,y);
  else
    z=foo2(x,y);
  output(z);
end
```

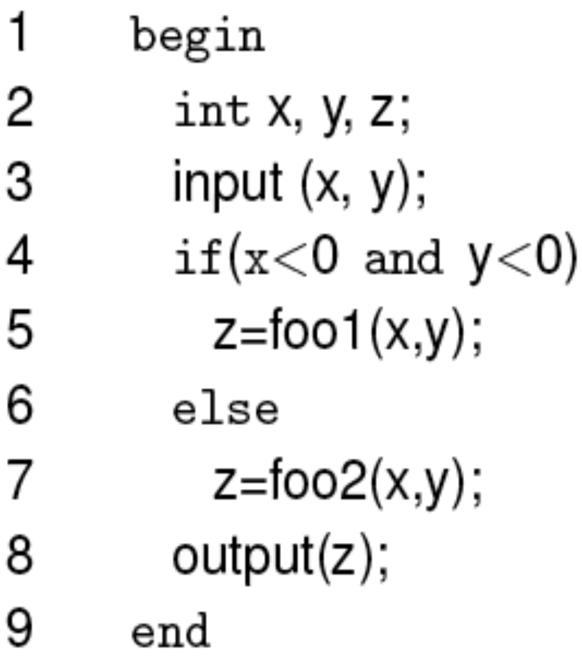
Partial specifications for computing z:

$x< 0$	$y< 0$	Output (z)
true	true	foo1(x,y)
true	false	foo2(x,y)
false	true	foo2(x,y)
false	false	foo1(x,y)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Condition coverage: Example (contd.)



```
begin
  int x, y, z;
  input (x, y);
  if(x<0 and y<0)
    z=foo1(x,y);
  else
    z=foo2(x,y);
  output(z);
end
```

Consider the test set:

$$T = \{t_1 :< x = -3, y = -2 > \; t_2 :< x = -4, y = -2 >\}$$

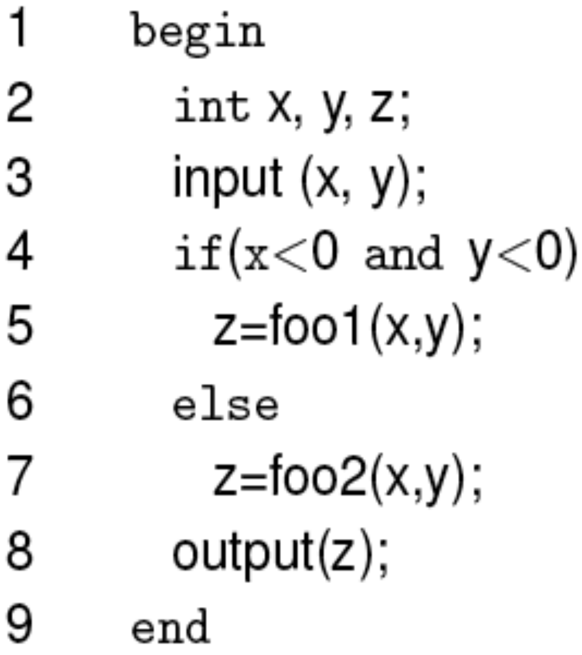
Check that T is adequate with respect to the statement, block, and decision coverage criteria and the program behaves correctly against t1 and t2.

Cc=1, Ce=2, Ci=0. Hence condition coverage for T=0.5.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Condition coverage: Example (contd.)



```
1    begin
2      int x, y, z;
3      input (x, y);
4      if(x<0 and y<0)
5        z=foo1(x,y);
6      else
7        z=foo2(x,y);
8      output(z);
9    end
```

Add the following test case to T:

t3: <x=3, y=4>

Check that the enhanced test set T is adequate with respect to the condition coverage criterion and possibly reveals an error in the program.

Under what conditions will a possible error at line 7 be revealed by t3?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Condition/decision coverage

When a decision is composed of a compound condition, decision coverage does not imply that each simple condition within a compound condition has taken both values true and false.

Condition coverage ensures that each component simple condition within a condition has taken both values true and false.

However, as illustrated next, condition coverage does not require each decision to have taken both outcomes. Condition/decision coverage is also known as branch condition coverage.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Condition/decision coverage: Example

Consider the following program and two test sets.

```
begin
  int x, y, z;
  input (x, y);
  if(x<0 or y<0)
    z=foo-1(x,y);
  else
    z=foo-2(x,y);
  output(z);
end
```

$$T_1 = \left\{ \begin{array}{lll} t_1: & < x = -3 & y = -2 > \\ t_2: & < x = 4 & y = -2 > \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{lll} t_1: & < x = -3 & y = 2 > \\ t_2: & < x = 4 & y = -2 > \end{array} \right\}$$

In class exercise: Confirm that T1 is adequate with respect to to decision coverage but not condition coverage.

In class exercise: Confirm that T2 is adequate with respect to condition coverage but not decision coverage.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

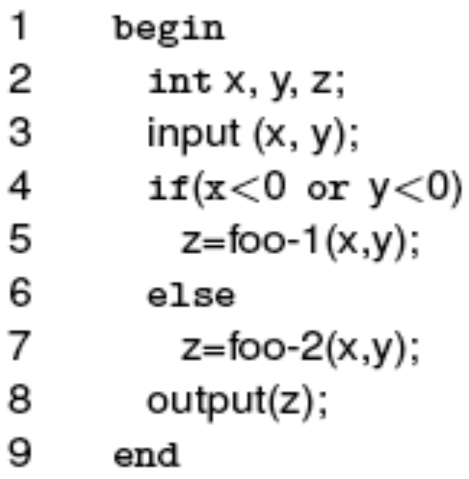
Condition/decision coverage: Definition

The condition/decision coverage of T with respect to (P, R) is computed as (Cc +Dc)/((Ce -Ci) +(De-Di)) , where Cc is the number of simple conditions covered, Dc is the number of decisions covered, Ce and De are the number of simple conditions and decisions respectively, and Ci and Di are the number of infeasible simple conditions and decisions, respectively.

T is considered adequate with respect to the multiple condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Condition/decision coverage: Example



```
begin
  int x, y, z;
  input (x, y);
  if(x<0 or y<0)
    z=foo-1(x,y);
  else
    z=foo-2(x,y);
  output(z);
end
```

In class exercise: Check that the following test set is adequate with respect to the condition/decision coverage criterion.

$$T = \left\{ \begin{array}{lll} t_1: & < x = -3 & y = -2 > \\ t_2: & < x = 4 & y = 2 > \end{array} \right\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.6 Multiple Condition coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multiple condition coverage

Consider a compound condition with two or more simple conditions. Using condition coverage on some compound condition C implies that each simple condition within C has been evaluated to true and false.

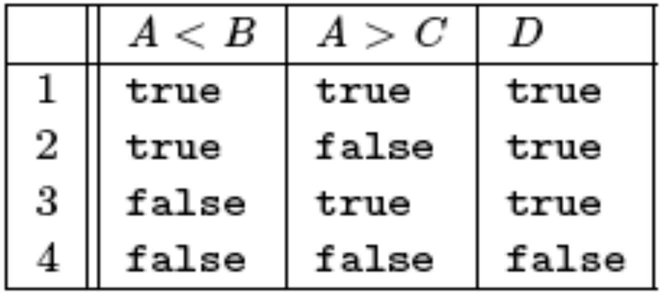
However, does it imply that all combinations of the values of the individual simple conditions in C have been exercised?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Multiple condition coverage: Example

Consider D=(A<B) OR (A>C) composed of two simple conditions A< B and A> C . The four possible combinations of the outcomes of these two simple conditions are enumerated in the table. Consider T:



	$A < B$	$A > C$	D
1	true	true	true
2	true	false	true
3	false	true	true
4	false	false	false

$$T = \left\{ \begin{array}{llll} t_1: & < A=2 & B=3 & C=1 > \\ t_2: & < A=2 & B=1 & C=3 > \end{array} \right\}$$

Check: Does T cover all four combinations?

Check: Does T' cover all four combinations?

$$T' = \left\{ \begin{array}{llll} t_1: & < A=2 & B=3 & C=1 > \\ t_2: & < A=2 & B=1 & C=3 > \\ t_3: & < A=2 & B=3 & C=5 > \\ t_4: & < A=2 & B=1 & C=5 > \end{array} \right\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Multiple condition coverage: Definition

Suppose that the program under test contains a total of n decisions. Assume also that each decision contains k1, k2, …, kn simple conditions. Each decision has several combinations of values of its constituent simple conditions.

For example, decision i will have a total of 2^{ki} combinations. Thus, the total number of combinations to be covered is

$$\sum_{i=1}^{n} 2^{ki}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multiple condition coverage: Definition (contd.)

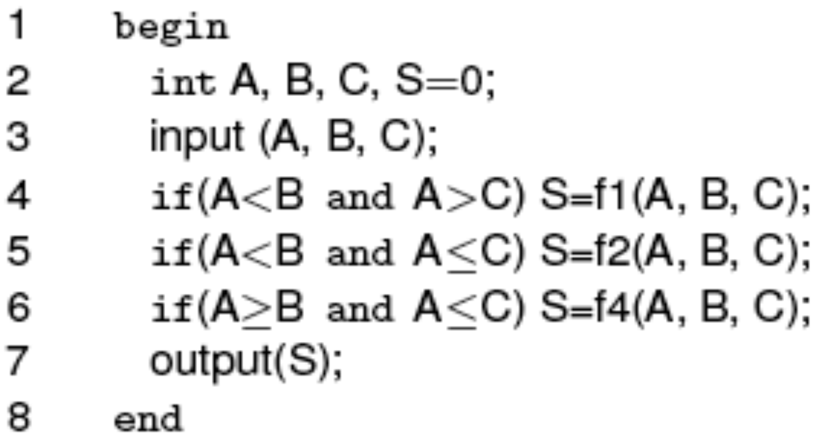
The multiple condition coverage of T with respect to (P, R) is computed as Cc/(C_e - Ci) , where Cc is the number of combinations covered, $_{Ci}$ is the number of infeasible simple combinations, and C_e is the total number of combinations in the program.

T is considered adequate with respect to the multiple condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

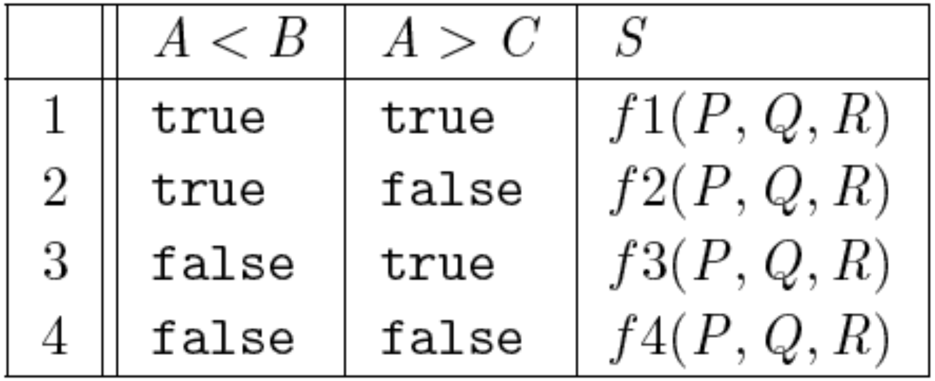
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Multiple condition coverage: Example

Consider the following program with specifications in the table.



```
begin
  int A, B, C, S=0;
  input (A, B, C);
  if(A<B and A>C) S=f1(A, B, C);
  if(A<B and A≤C) S=f2(A, B, C);
  if(A≥B and A≤C) S=f4(A, B, C);
  output(S);
end
```



	$A < B$	$A > C$	S
1	true	true	$f1(P, Q, R)$
2	true	false	$f2(P, Q, R)$
3	false	true	$f3(P, Q, R)$
4	false	false	$f4(P, Q, R)$

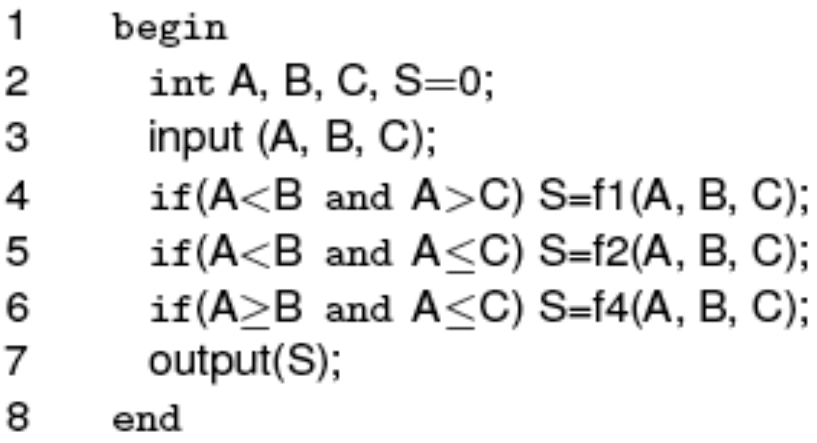
There is an obvious error in the program, computation of S for one of the four combinations, line 3 in the table, has been left out.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Multiple condition coverage: Example (contd.)

Is T adequate with respect to decision coverage? Multiple condition coverage? Does it reveal the error?



```
begin
  int A, B, C, S=0;
  input (A, B, C);
  if(A<B and A>C) S=f1(A, B, C);
  if(A<B and A≤C) S=f2(A, B, C);
  if(A≥B and A≤C) S=f4(A, B, C);
  output(S);
end
```

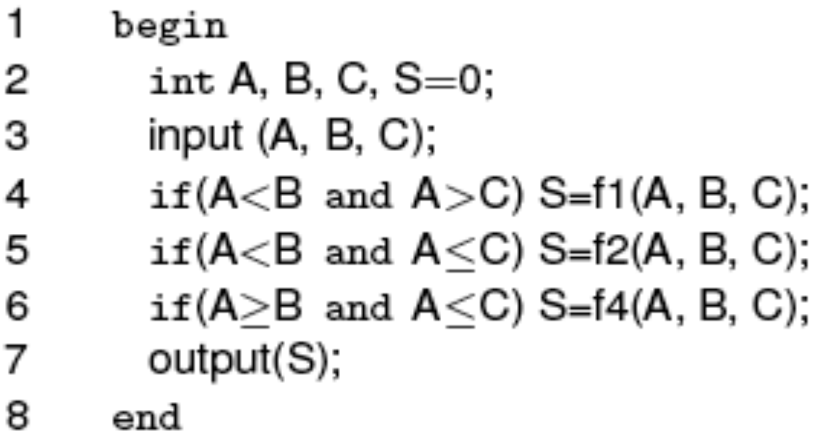
$$T = \left\{ \begin{array}{lllll} t_1: & < A = 2 & B = 3 & C = 1 & > \\ t_2: & < A = 2 & B = 1 & C = 3 & > \end{array} \right\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Multiple condition coverage: Example (contd.)

To improve decision coverage we add t3 to T and obtain T'.



```
begin
  int A, B, C, S=0;
  input (A, B, C);
  if(A<B and A>C) S=f1(A, B, C);
  if(A<B and A≤C) S=f2(A, B, C);
  if(A≥B and A≤C) S=f4(A, B, C);
  output(S);
end
```

$$T' = \left\{ \begin{array}{ll} t_1: & < A = 2, B = 3, C = 1 > \\ t_2: & < A = 2, B = 1, C = 3 > \\ t_3: & < A = 2, B = 3, C = 5 > \end{array} \right\}$$

Does T' reveal the error?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Multiple condition coverage: Example (contd.)

In class exercise: Construct a table showing the simple conditions covered by T' . Do you notice that some combinations of simple conditions remain uncovered?

Now add a test to T' to cover the uncovered combinations. Does your test reveal the error? If yes, then under what conditions?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.2.7 LCSAJ coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Linear Code Sequence and Jump (LCSAJ)

Execution of sequential programs that contain at least one condition, proceeds in pairs where the first element of the pair is a sequence of statements, executed one after the other, and terminated by a jump to the next such pair.

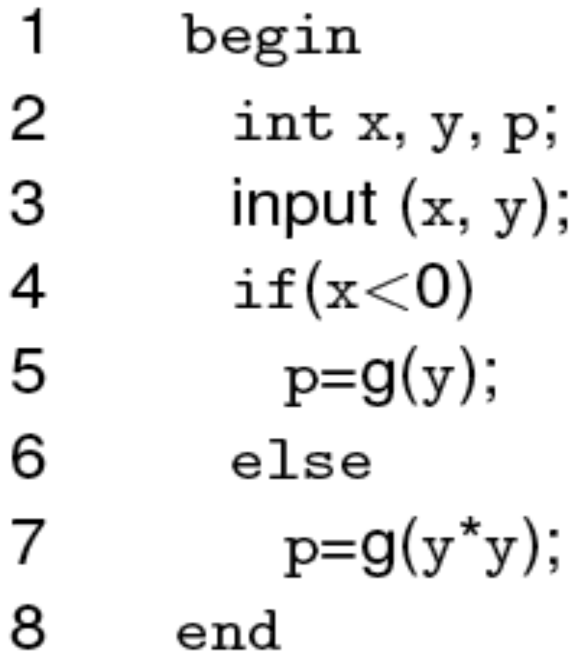
A Linear Code Sequence and Jump is a program unit comprised of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump.

An LCSAJ is represented as a triple (X, Y, Z) where X and Y are, respectively, locations of the first and the last statements and Z is the location to which the statement at Y jumps.

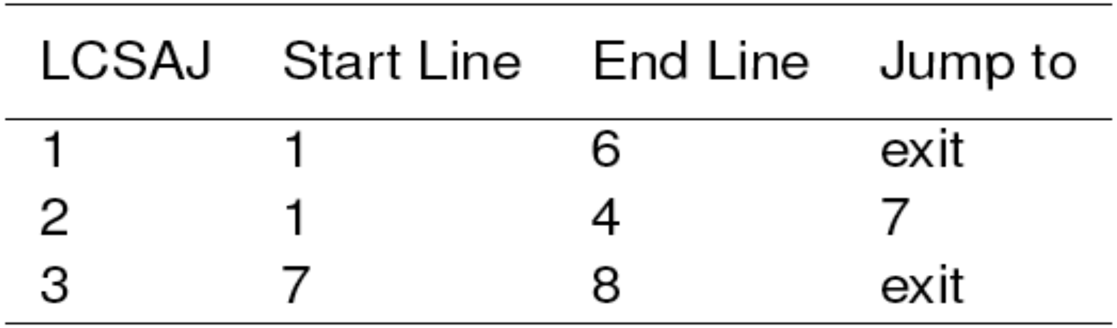
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Linear Code Sequence and Jump (LCSAJ)

Consider this program.



```
1   begin
2     int x, y, p;
3     input (x, y);
4     if(x<0)
5       p=g(y);
6     else
7       p=g(y*y);
8   end
```



LCSAJ	Start Line	End Line	Jump to
1	1	6	exit
2	1	4	7
3	7	8	exit

The last statement in an LCSAJ (X, Y, Z) is a jump and Z may be program exit. When control arrives at statement X, follows through to statement Y, and then jumps to statement Z, we say that the LCSAJ (X, Y, Z) is traversed or covered or exercised.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

LCSAJ coverage: Example 1

```
1  begin
2    int x, y, p;
3    input (x, y);
4    if(x<0)
5      p=g(y);
6    else
7      p=g(y*y);
8  end
```

$$T = \left\{ \begin{array}{lll} t_1: & < x = -5 & y = 2 > \\ t_2: & < x = 9 & y = 2 > \end{array} \right\}$$

t1 covers (1,4,7) and (7, 8, exit). t2 covers (1, 6, exit) is executed. T covers all three LCSAJs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

LCSAJ coverage: Example 2

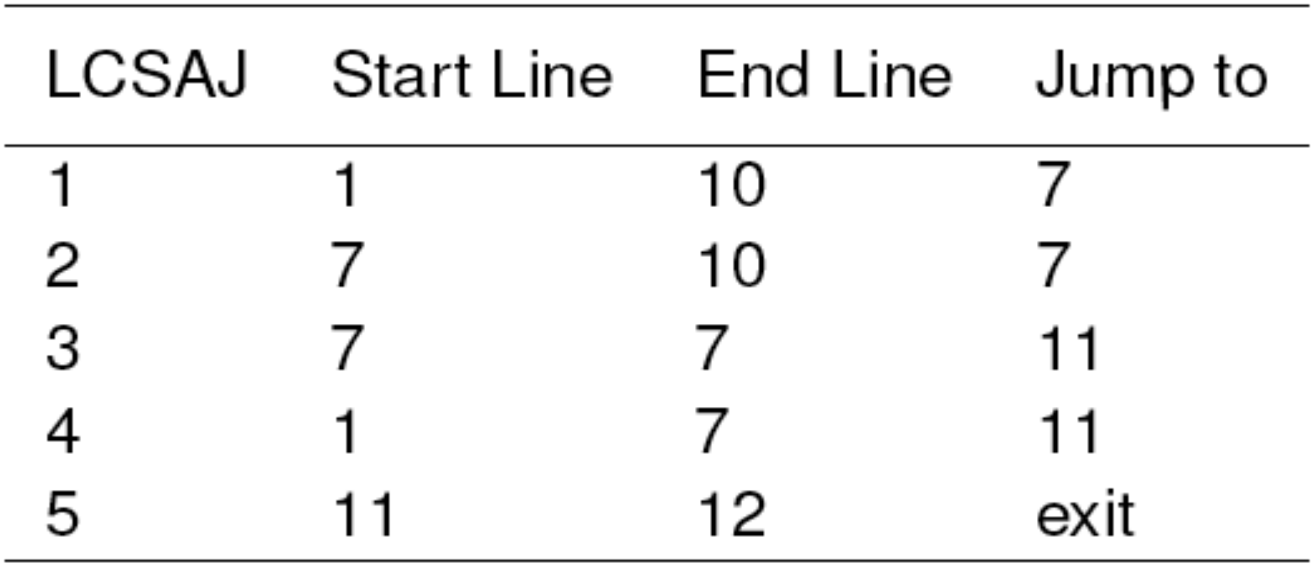
```
1    begin
2    // Compute $x^y$ given non-negative integers x and y.
3      int x, y, p;
4      input (x, y);
5      p=1;
6      count=y;
7      while(count>0){
8        p=p*x;
9        count=count-1;
10     }
11     output(p);
12   end
```

In class exercise: Find all LCSAJs

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

LCSAJ coverage: Example 2 (contd.)



LCSAJ	Start Line	End Line	Jump to
1	1	10	7
2	7	10	7
3	7	7	11
4	1	7	11
5	11	12	exit

Verify: This set covers all LCSAJs.

$$T = \left\{ \begin{array}{lll} t_1 : & < x = 5 & y = 0 > \\ t_2 : & < x = 5 & y = 2 > \end{array} \right\}$$

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

LCSAJ coverage: Definition

The LCSAJ coverage of a test set T with respect to (P, R) is computed as

$$\frac{\text{Number of LCSAJs exercised}}{\text{Total number of feasible LCSAJs}}$$

T is considered adequate with respect to the LCSAJ coverage criterion if the LCSAJ coverage of T with respect to (P, R) is .

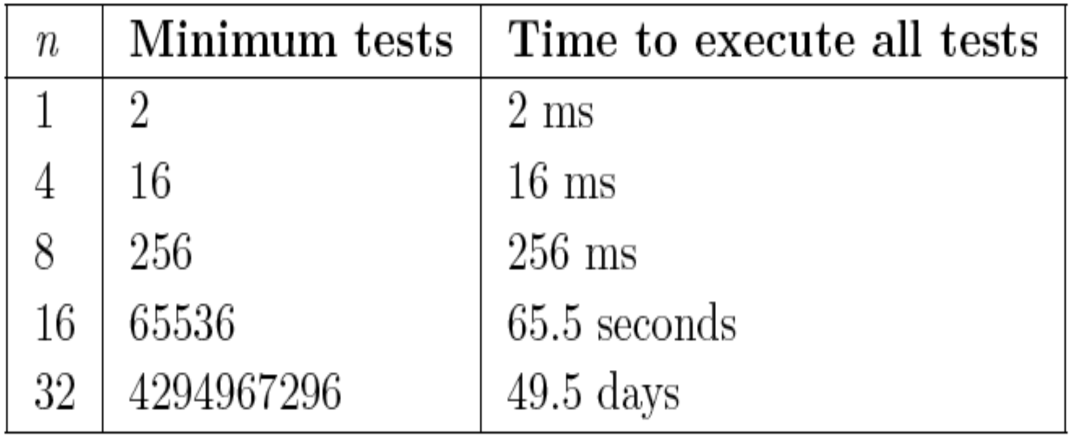
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.2.8 Modified condition/decision coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Modified Condition/Decision (MC/DC) Coverage

Obtaining multiple condition coverage might become expensive when there are many embedded simple conditions. When a compound condition C contains n simple conditions, the maximum number of tests required to cover C is 2^n .



n	Minimum tests	Time to execute all tests
1	2	2 ms
4	16	16 ms
8	256	256 ms
16	65536	65.5 seconds
32	4294967296	49.5 days

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Compound conditions and MC/DC

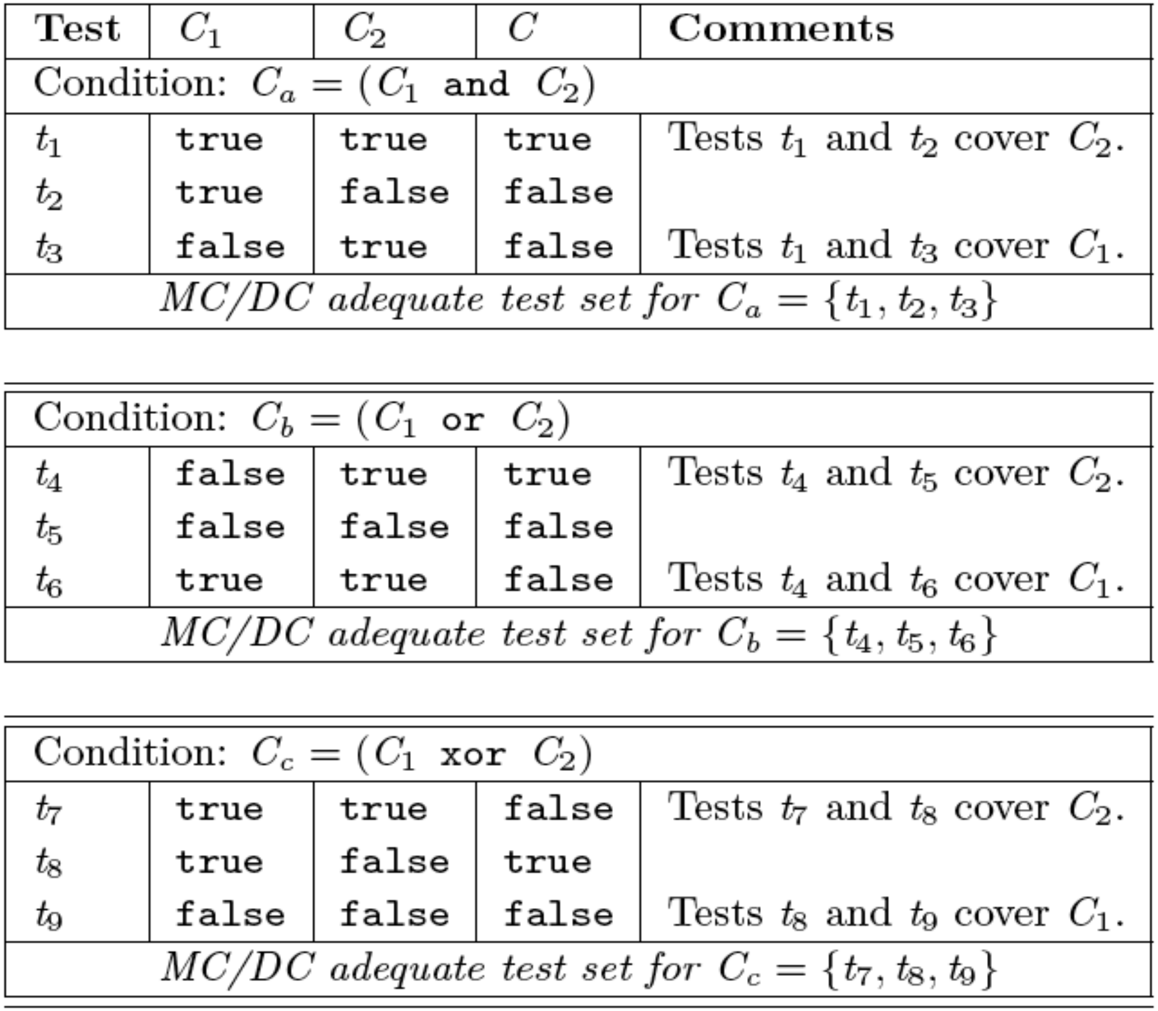
MC/DC coverage requires that every compound condition in a program must be tested by demonstrating that each simple condition within the compound condition has an independent effect on its outcome.

Thus, MC/DC coverage is a weaker criterion than the multiple condition coverage criterion.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC coverage: Simple conditions



Test	C_1	C_2	C	Comments
Condition: $C_a = (C_1 \texttt{ and } C_2)$				
t_1	`true`	`true`	`true`	Tests t_1 and t_2 cover C_2.
t_2	`true`	`false`	`false`	
t_3	`false`	`true`	`false`	Tests t_1 and t_3 cover C_1.
MC/DC adequate test set for $C_a = \{t_1, t_2, t_3\}$				
Condition: $C_b = (C_1 \texttt{ or } C_2)$				
t_4	`false`	`true`	`true`	Tests t_4 and t_5 cover C_2.
t_5	`false`	`false`	`false`	
t_6	`true`	`true`	`false`	Tests t_4 and t_6 cover C_1.
MC/DC adequate test set for $C_b = \{t_4, t_5, t_6\}$				
Condition: $C_c = (C_1 \texttt{ xor } C_2)$				
t_7	`true`	`true`	`false`	Tests t_7 and t_8 cover C_2.
t_8	`true`	`false`	`true`	
t_9	`false`	`false`	`false`	Tests t_8 and t_9 cover C_1.
MC/DC adequate test set for $C_c = \{t_7, t_8, t_9\}$				

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

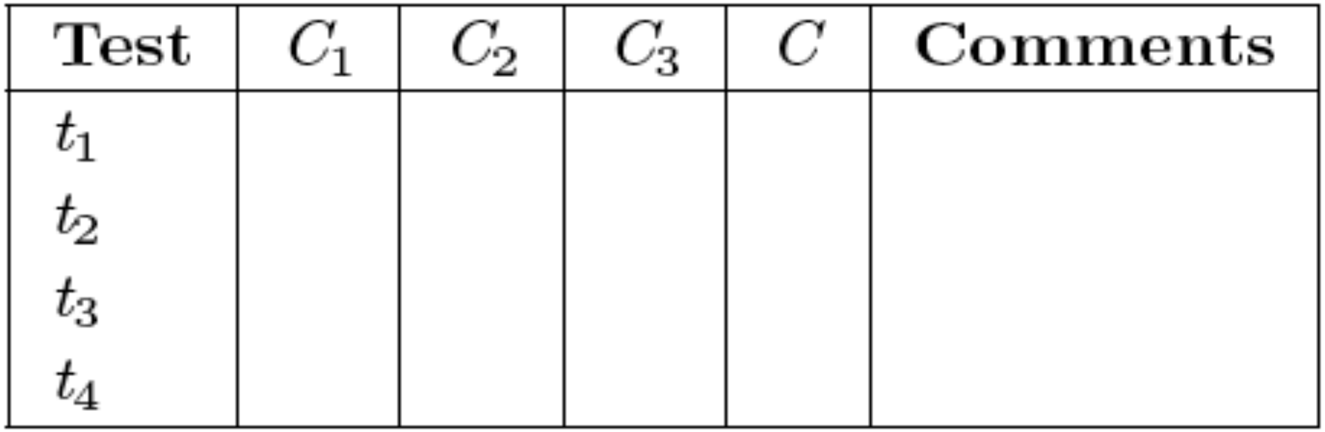
Contents

7.2.9 MC/DC adequate tests for compound conditions

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Generating tests for compound conditions

Let C=C1 and C2 and C3. Create a table with five columns and four rows. Label the columns as Test, C1, C2 , C3 and C, from left to right. An optional column labeled "Comments" may be added. The column labeled Test contains rows labeled by test case numbers t1 through t4 . The remaining entries are empty.



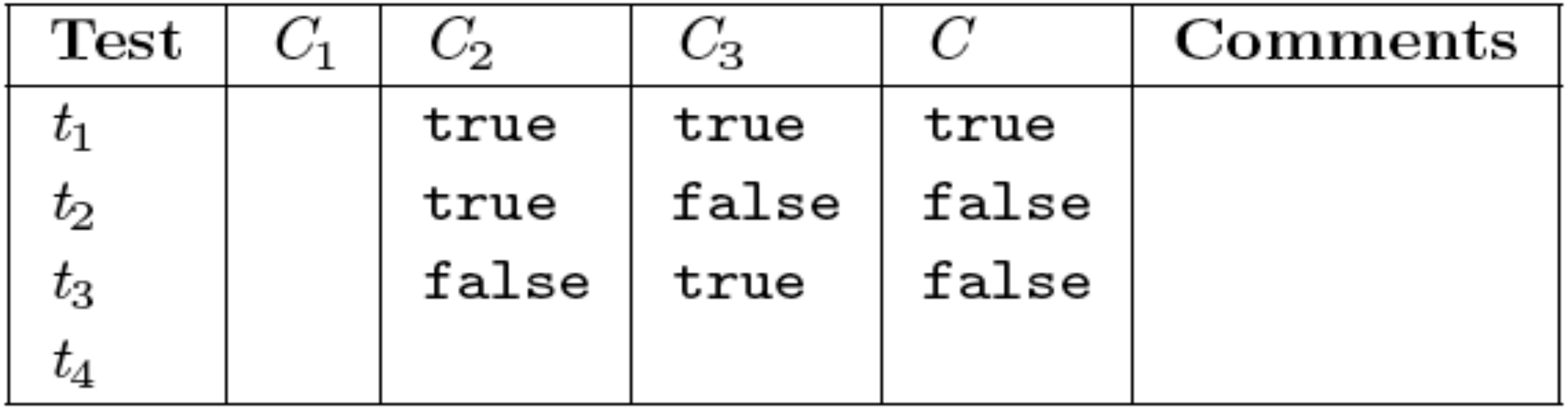
Test	C_1	C_2	C_3	C	Comments
t_1					
t_2					
t_3					
t_4					

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Generating tests for compound conditions (contd.)

Copy all entries in columns C1 , C2 , and C from the table for simple conditions into columns C2, C3, and C of the empty table.



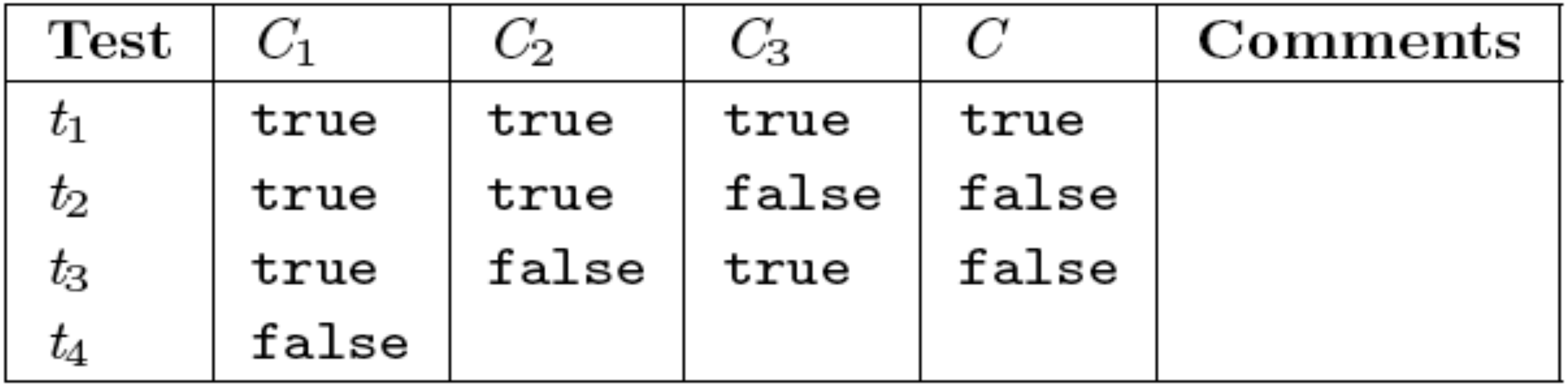
Test	C_1	C_2	C_3	C	Comments
t_1		true	true	true	
t_2		true	false	false	
t_3		false	true	false	
t_4					

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Generating tests for compound conditions (contd.)

Fill the first three rows in the column marked C1 with true and the last row with false.



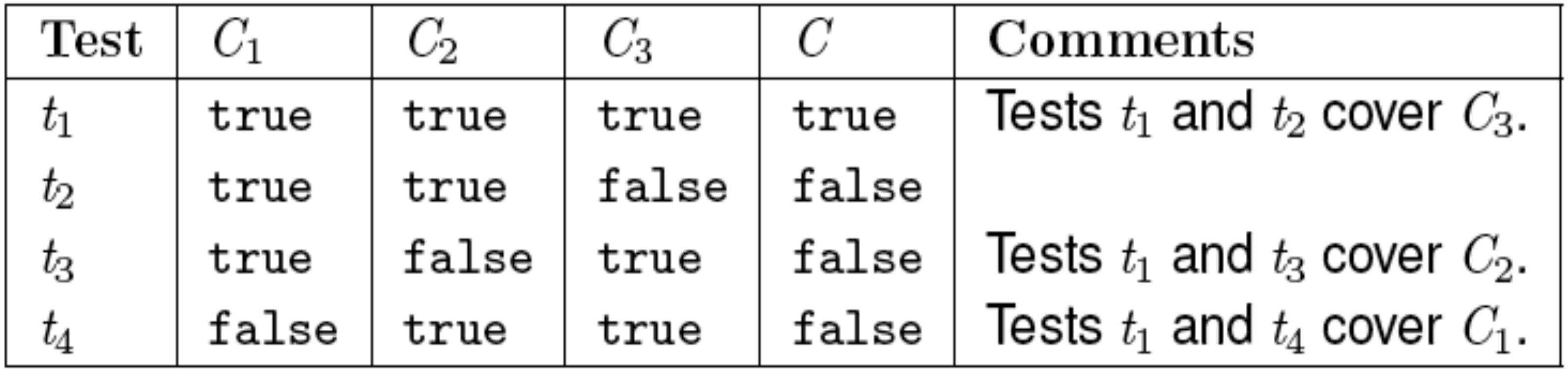
Test	C_1	C_2	C_3	C	Comments
t_1	`true`	`true`	`true`	`true`	
t_2	`true`	`true`	`false`	`false`	
t_3	`true`	`false`	`true`	`false`	
t_4	`false`				

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC coverage: Generating tests for compound conditions (contd.)

Fill the last row under columns labeled C2 , C3 , and C with true, true, and false, respectively.



Test	C_1	C_2	C_3	C	Comments
t_1	true	true	true	true	Tests t_1 and t_2 cover C_3.
t_2	true	true	false	false	
t_3	true	false	true	false	Tests t_1 and t_3 cover C_2.
t_4	false	true	true	false	Tests t_1 and t_4 cover C_1.

We now have a table containing MC/DC adequate tests for C=(C1 AND C2 AND C3) derived from tests for C=(C1 AND C2) .

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC coverage: Generating tests for compound conditions (contd.)

The procedure illustrated above can be extended to derive tests for any compound condition using tests for a simpler compound condition (solve Exercises 7.15 and 7.16).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.2.10 Definition of MC/DC coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MC/DC coverage: Definition

A test set T for program P written to meet requirements R, is considered adequate with respect to the MC/DC coverage criterion if upon the execution of P on each test in T, the following requirements are met.

- Each block in P has been covered.
- Each simple condition in P has taken both true and false values.
- Each decision in P has taken all possible outcomes.
- Each simple condition within a compound condition C in P has been shown to independently effect the outcome of C. *This is the MC part of the coverage we discussed.*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Analysis

The first three requirements above correspond to block, condition, and decision coverage, respectively.

The fourth requirement corresponds to ``MC'' coverage. Thus, the MC/DC coverage criterion is a mix of four coverage criteria based on the flow of control.

With regard to the second requirement, it is to be noted that conditions that are not part of a decision, such as the one in the following statement A= (p<q) OR (x>y) are also included in the set of conditions to be covered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Analysis (contd.)

With regard to the fourth requirement, a condition such as (A AND B) OR (C AND A) poses a problem. It is not possible to keep the first occurrence of A fixed while varying the value of its second occurrence.

Here the first occurrence of A is said to be coupled to its second occurrence. In such cases an adequate test set need only demonstrate the independent effect of any one occurrence of the coupled condition

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Adequacy

Let C_1, C_2, .., C_N be the conditions in P. ni denote the number of simple conditions in C_i , e_i the number of simple conditions shown to have independent affect on the outcome of C_i, and f_i the number of infeasible simple conditions in C_i .

The MC coverage of T for program P subject to requirements R, denoted by MC_c, is computed as follows.

$$\mathrm{MC}_c = \frac{\sum_{i=1}^{i=N} e_i}{\sum_{i=1}^{i=N} (e_i - f_i)}$$

Test set T is considered adequate with respect to the MC coverage if MC_c=1 of T is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example

Consider the following requirements:

R1.1: Invoke fire-1 when (x<y) AND (z * z > y) AND (prev=``East").

R1.2: Invoke fire-2 when (x<y) AND (z * z ≤ y) OR (current=``South").

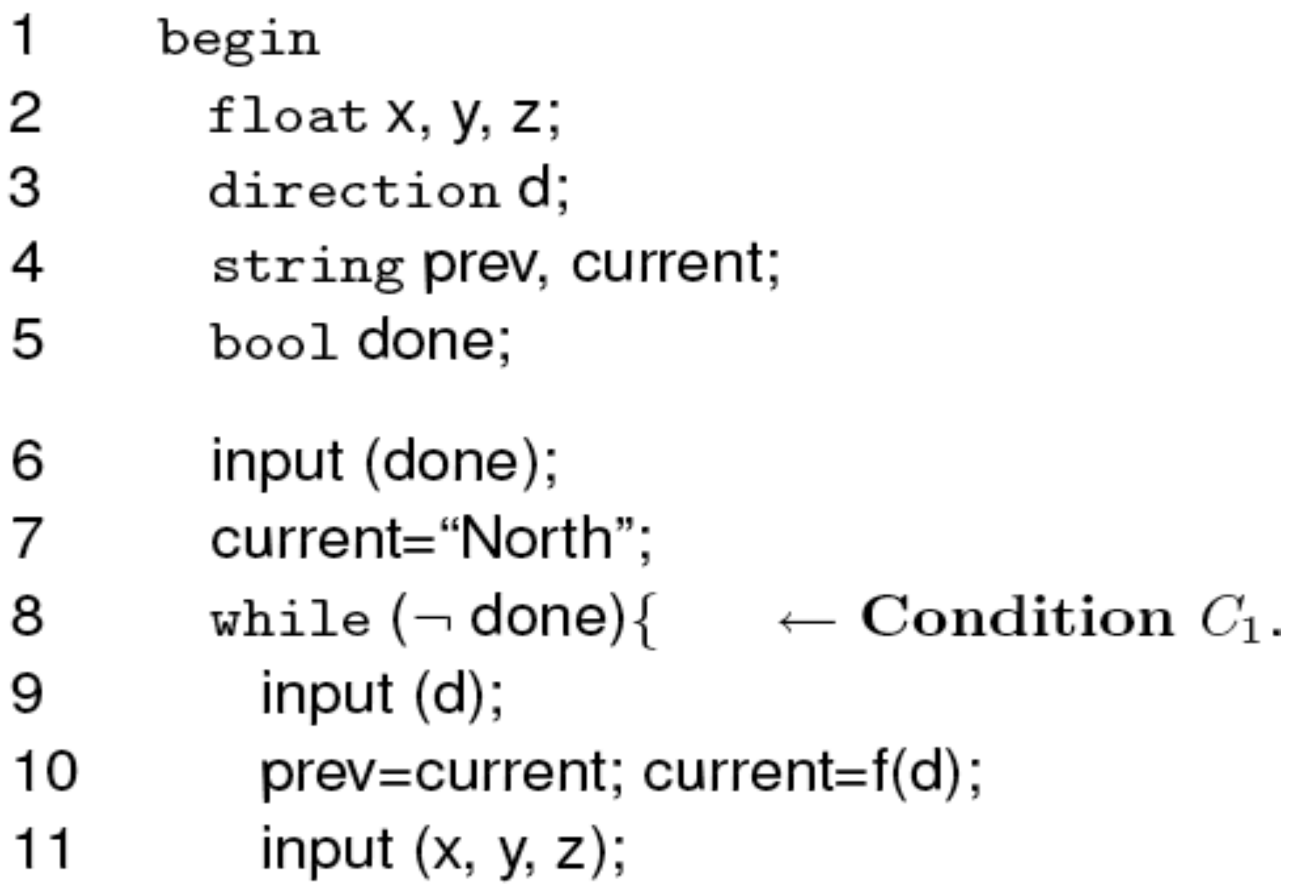
R1.3: Invoke fire-3 when none of the two conditions above is true.

R2: The invocation described above must continue until an input Boolean variable becomes true.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

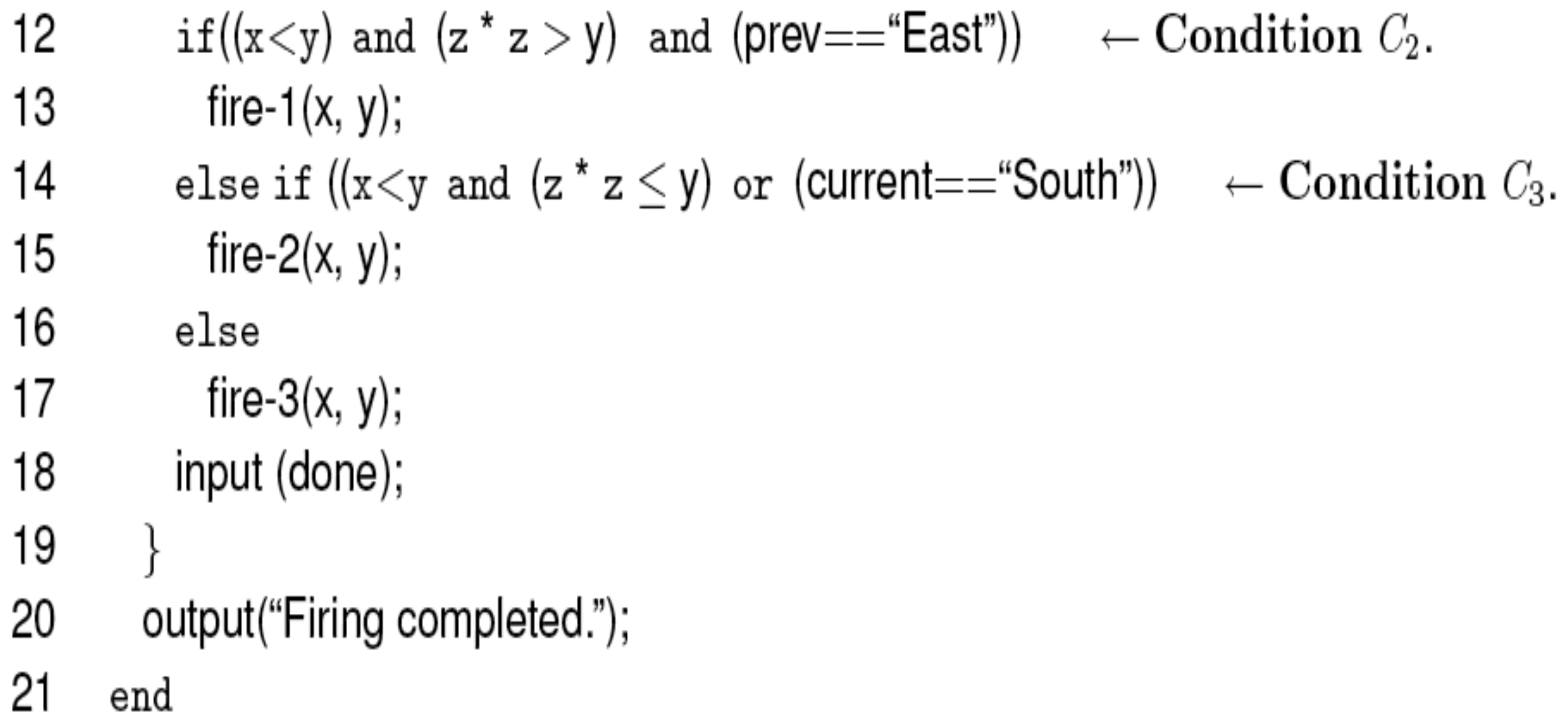


```
begin
  float x, y, z;
  direction d;
  string prev, current;
  bool done;

  input (done);
  current="North";
  while ($\neg$ done){        $\leftarrow$ Condition $C_1$.
    input (d);
    prev=current; current=f(d);
    input (x, y, z);
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)



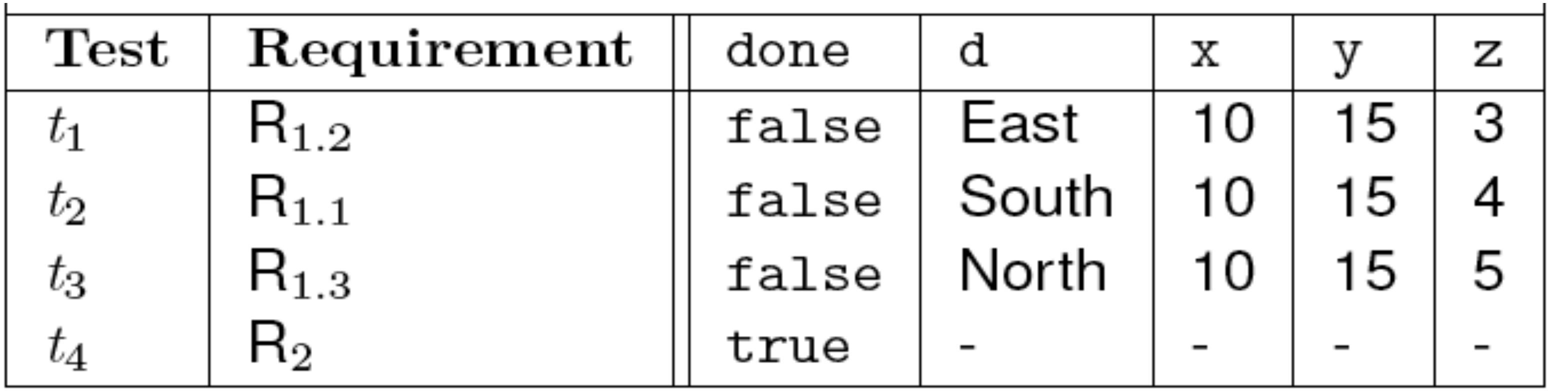
```
        if((x<y) and (z * z > y)  and (prev=="East"))      ← Condition C2.
          fire-1(x, y);
        else if ((x<y and (z * z ≤ y) or (current=="South"))      ← Condition C3.
          fire-2(x, y);
        else
          fire-3(x, y);
        input (done);
      }
    output("Firing completed.");
  end
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Verify that the following set T1 of four tests, executed in the given order, is adequate with respect to statement, block, and decision coverage criteria but not with respect to the condition coverage criterion.



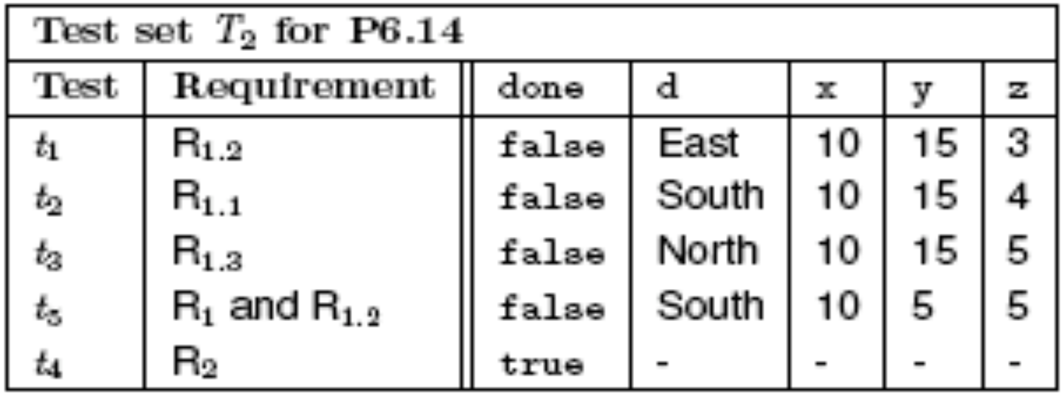
Test	Requirement	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_4	R_2	true	-	-	-	-

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Verify that the following set T2, obtained by adding t5 to T1, is adequate with respect to the condition coverage but not with respect to the multiple condition coverage criterion. Note that sequencing of tests is important in this case!



Test set T_2 for P6.14

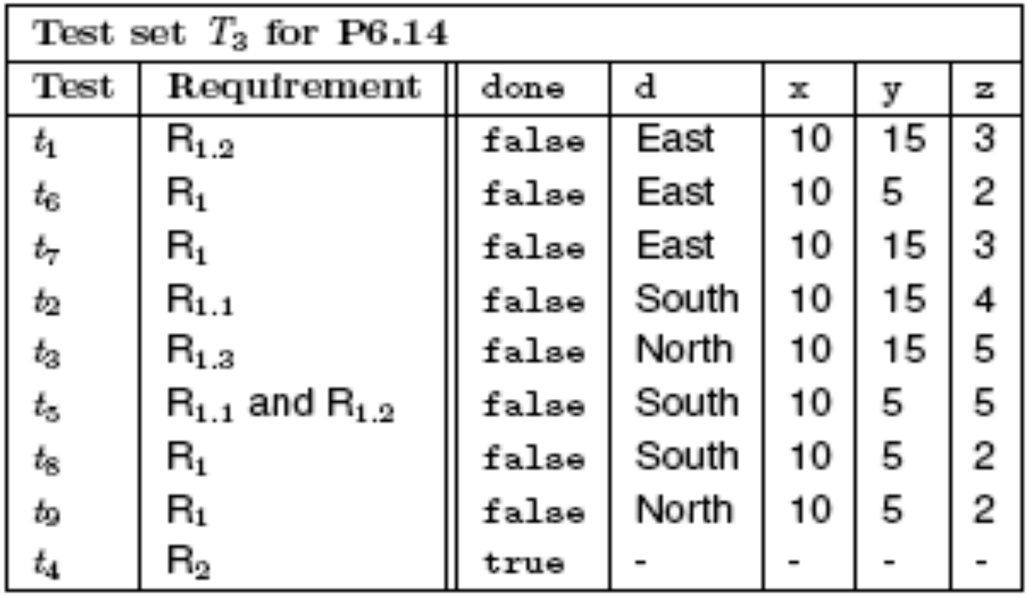
Test	Requirement	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_5	R_1 and $R_{1.2}$	false	South	10	5	5
t_4	R_2	true	-	-	-	-

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Verify that the following set T3, obtained by adding t6, t7, t8, and t9 to T2 is adequate with respect to MC/DC coverage criterion. Note again that sequencing of tests is important in this case (especially for t1 and t7)!



Test set T_3 for P6.14

Test	Requirement	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_6	R_1	false	East	10	5	2
t_7	R_1	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_5	$R_{1.1}$ and $R_{1.2}$	false	South	10	5	5
t_8	R_1	false	South	10	5	2
t_9	R_1	false	North	10	5	2
t_4	R_2	true	-	-	-	-

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.12 Error detection and MC/DC adequacy

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

MC/DC adequacy and error detection

We consider the following three types of errors.

Missing condition: One or more simple conditions is missing from a compound condition. For example, the correct condition should be (x<y AND done) but the condition coded is (done).

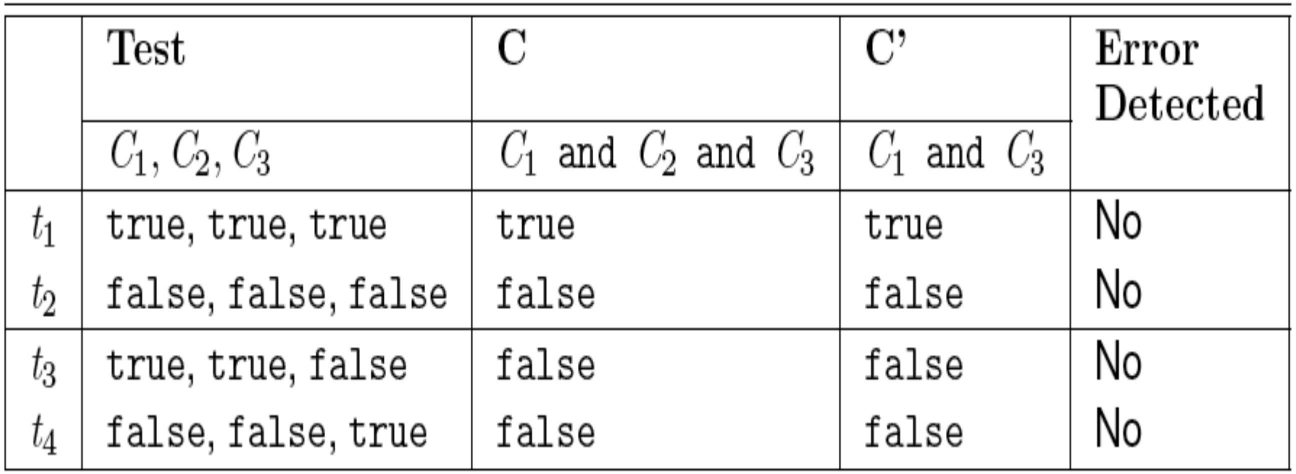
Incorrect Boolean operator: One or more Boolean operators is incorrect. For example, the correct condition is (x<y AND done) which has been coded as (x<y OR done).

Mixed: One or more simple conditions is missing and one or more Boolean operators is incorrect. For example, the correct condition should be (x<y AND z*x ≥ y AND d=``South") has been coded as (x<y OR z*x ≥ y).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example

Suppose that condition C=C_1 AND C_2 AND C_3 has been coded as C'=C_1 AND C_2. Four tests that form an MC/DC adequate set are in the following table. Verify that the following set of four tests is MC/DC adequate but does not reveal the error.



	Test	C	C’	Error Detected
	C_1, C_2, C_3	C_1 and C_2 and C_3	C_1 and C_3	
t_1	true, true, true	true	true	No
t_2	false, false, false	false	false	No
t_3	true, true, false	false	false	No
t_4	false, false, true	false	false	No

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC and condition coverage

Several examples in the book show that satisfying the MC/DC adequacy criteria does not necessarily imply that errors made while coding conditions will be revealed. *However, the examples do favor MC/DC over condition coverage.*

The examples also show that an MC/DC adequate test will *likely* reveal more errors than a decision or condition-coverage adequate test. (Note the emphasis on "likely.")

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC and short circuit evaluation

Consider C=C1 AND C2.

The outcome of the above condition does not depend on C2 when C1 is false. When using short-circuit evaluation, condition C2 is not evaluated if C1 evaluates to false.

Thus, the combination C1=false and C2=true, or the combination C1=false and C2=false may be infeasible if the programming language allows, or requires as in C, short circuit evaluation.

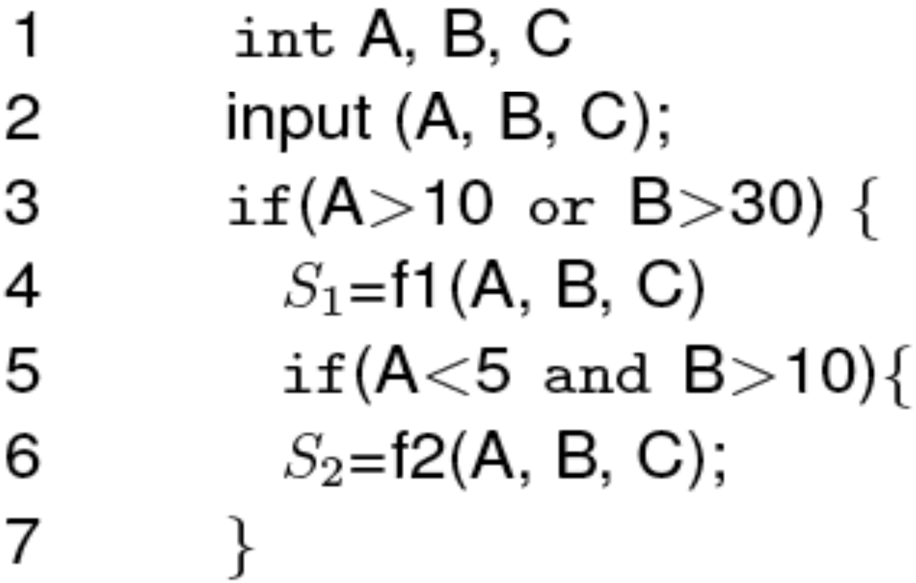
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

MC/DC and decision dependence

Dependence of one decision on another might also lead to an infeasible combination.

Consider, for example, the following sequence of statements.



```
int A, B, C
input (A, B, C);
if(A>10 or B>30) {
   S1=f1(A, B, C)
   if(A<5 and B>10){
   S2=f2(A, B, C);
}
```

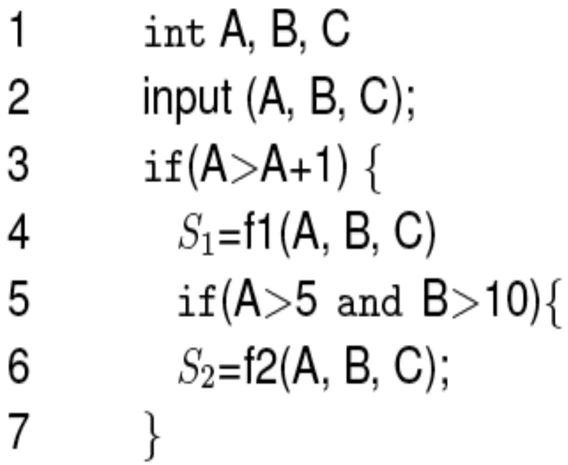
← Infeasible condition A<5

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Infeasibility and reachability

Note that infeasibility is different from reachability. A decision might be reachable but not feasible and vice versa. In the sequence above, both decisions are reachable but the second decision is not feasible. Consider the following sequence.



```
1    int A, B, C
2    input (A, B, C);
3    if(A>A+1) {
4      S1=f1(A, B, C)
5      if(A>5 and B>10){
6      S2=f2(A, B, C);
7    }
```

In this case the second decision is not reachable due an error at line 3. It may, however, be feasible.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.2.15 Tracing test cases to requirements

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test trace back

When enhancing a test set to satisfy a given coverage criterion, it is desirable to ask the following question: What portions of the requirements are tested when the program under test is executed against the newly added test case? The task of relating the new test case to the requirements is known as test trace-back.

Advantages of trace back: Assists us in determining whether or not the new test case is redundant.

It has the likelihood of revealing errors and ambiguities in the requirements.

It assists with the process of documenting tests against requirements.

See example 7.27.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.3 Concepts from data flow

7.3.1 Definitions and uses

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Basic concepts

We will now examine some test adequacy criteria based on the flow of “data” in a program. This is in contrast to criteria based on “flow of control” that we have examined so far.

Test adequacy criteria based on the flow of data are useful in improving tests that are adequate with respect to control-flow based criteria. Let us look at an example.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example: Test enhancement using data flow

Here is an MC/DC adequate test set that does not reveal the error.

```
begin
  int x, y; float z;
  input (x, y);
  z=0;
  if (x! =0)
    z=z+y;
  else z=z-y;
  if (y! =0) ← This condition should be (y! =0 and x! =0)
    z=z/x;
  else z=z*x;
  output(z);
end
```

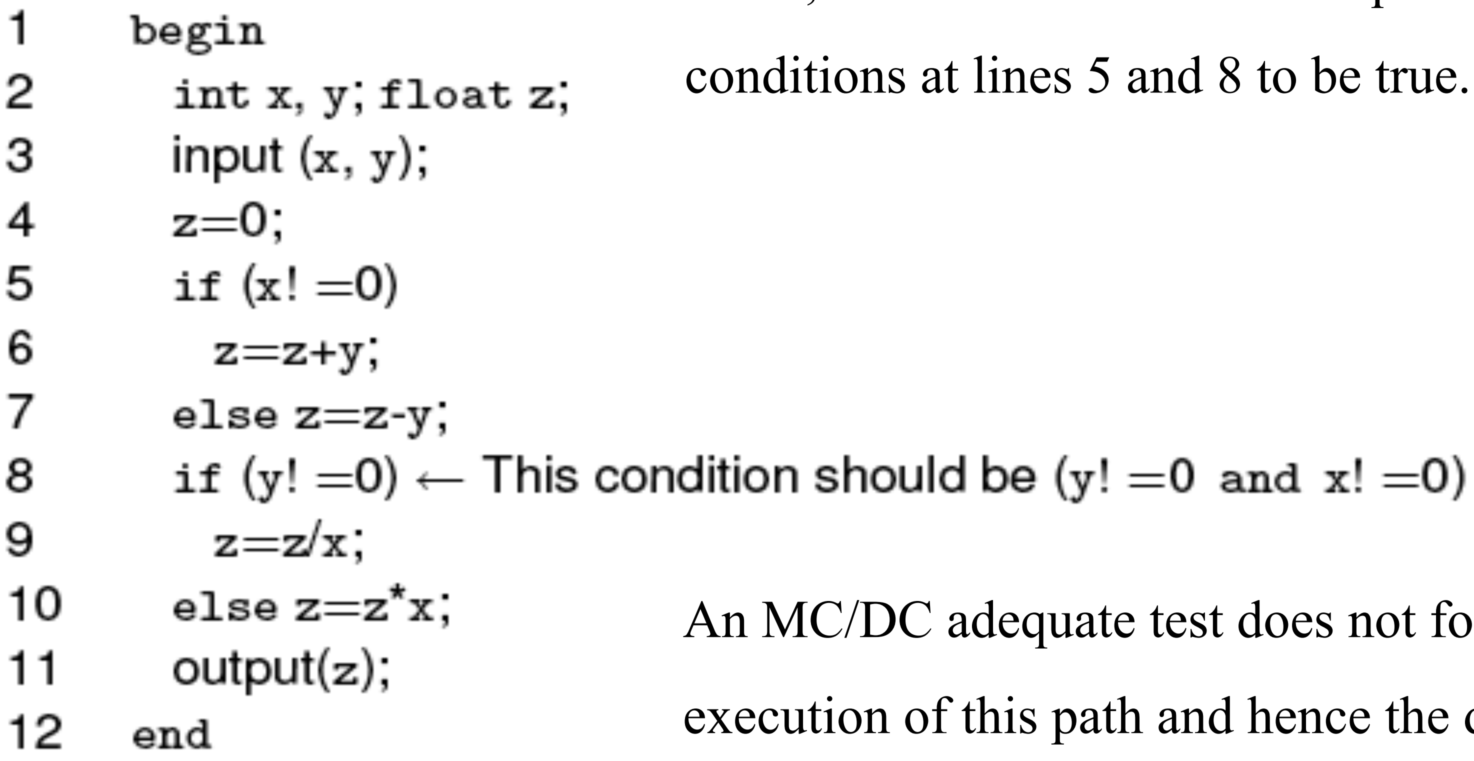
Test	x	y	z
t_1	0	0	0.0
t_2	1	1	1.0

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Neither of the two tests force the use of z defined on line 6, at line 9. To do so one requires a test that causes conditions at lines 5 and 8 to be true.



```
1    begin
2      int x, y; float z;
3      input (x, y);
4      z=0;
5      if (x! =0)
6        z=z+y;
7      else z=z-y;
8      if (y! =0) ← This condition should be (y! =0 and x! =0)
9        z=z/x;
10     else z=z*x;
11     output(z);
12   end
```

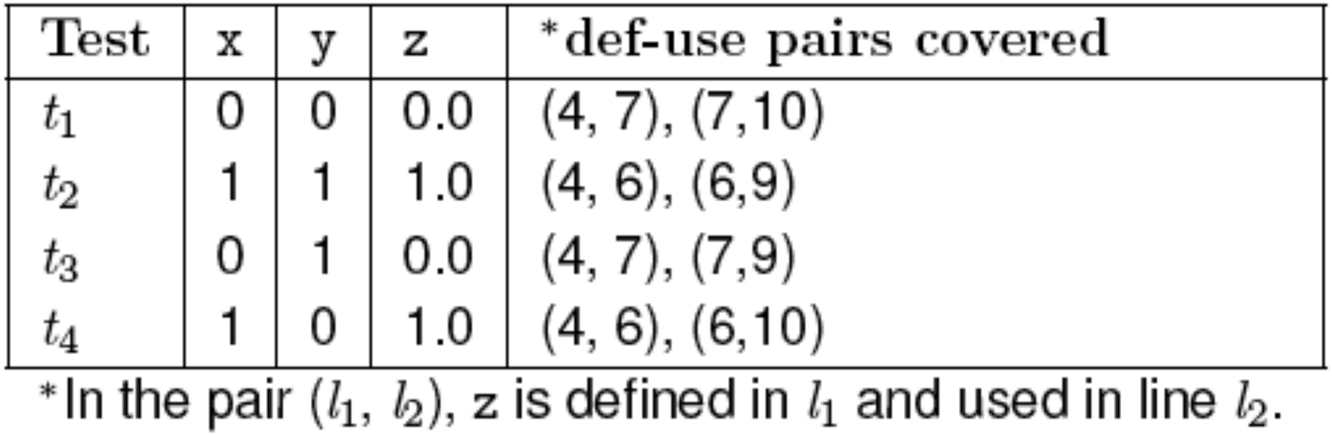
An MC/DC adequate test does not force the execution of this path and hence the divide by zero error is not revealed.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Example (contd.)

Verify that the following test set covers all def-use pairs of z and reveals the error.



Test	x	y	z	*def-use pairs covered
t_1	0	0	0.0	(4, 7), (7,10)
t_2	1	1	1.0	(4, 6), (6,9)
t_3	0	1	0.0	(4, 7), (7,9)
t_4	1	0	1.0	(4, 6), (6,10)

*In the pair (l_1, l_2), z is defined in l_1 and used in line l_2.

Would an LCSAJ adequate test also reveal the error?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Definitions and uses

A program written in a procedural language, such as C and Java, contains variables.

Variables are defined by assigning values to them and are used in expressions.

Statement x=y+z defines variable x and uses variables y and z.

Declaration int x, y, A[10]; defines three variables.

Statement scanf(``%d %d", &x, &y) defines variables x and y.

Statement printf(``Output: %d \n", x+y) uses variables x and y.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Definitions and uses (contd.)

A parameter x passed as call-by-value to a function, is considered as a use of, or a reference to, x.

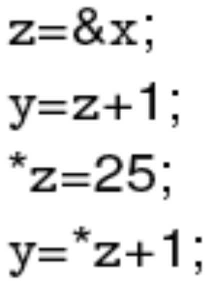
A parameter x passed as call-by-reference, serves as a definition and use of x

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Definitions and uses: Pointers

Consider the following sequence of statements that use pointers.



```
z=&x;
y=z+1;
*z=25;
y=*z+1;
```

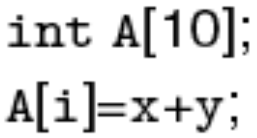
The first of the above statements defines a pointer variable z the second defines y and uses z the third defines x through the pointer variable z and the last defines y and uses x accessed through the pointer variable z.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Definitions and uses: Arrays

Arrays are also tricky. Consider the following declaration and two statements in C:



```
int A[10];
A[i]=x+y;
```

The first statement defines variable A. The second statement defines A and uses i , x, and y. Alternate: second statement defines A[i] and not the entire array A. The choice of whether to consider the entire array A as defined or the specific element depends upon how stringent is the requirement for coverage analysis.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.3.2 C-use and p-use

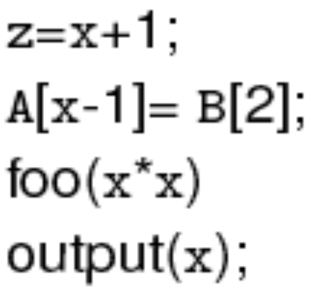
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

c-use

Uses of a variable that occur within an expression as part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions, are classified as c-use, where the ``c" in c-use stands for computational.

How many c-uses of x can you find in the following statements?



```
z=x+1;
A[x-1]= B[2];
foo(x*x)
output(x);
```

Answer: 5

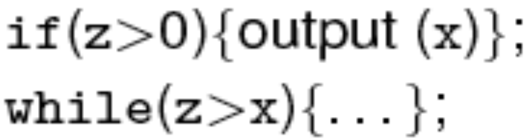
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

p-use

The occurrence of a variable in an expression used as a condition in a branch statement such as an if and a while, is considered as a p-use. The ``p'' in p-use stands for predicate.

How many p-uses of z and x can you find in the following statements?



```
if(z>0){output (x)};
while(z>x){...};
```

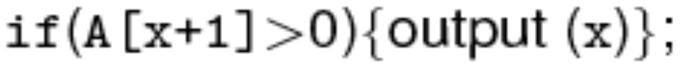
Answer: 3 (2 of z and 1 of x)

Contents

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

p-use: possible confusion

Consider the statement:



```
if(A[x+1]>0){output (x)};
```

The use of A is clearly a p-use.

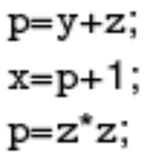
Is the use of x in the subscript, a c-use or a p-use? Discuss.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C-uses within a basic block

Consider the basic block



```
p=y+z;
x=p+1;
p=z*z;
```

While there are two definitions of p in this block, only the second definition will propagate to the next block. The first definition of p is considered local to the block while the second definition is global. *We are concerned with global definitions, and uses.*

Note that y and z are global uses; their definitions flow into this block from some other block.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

7.3.4 Data flow graph

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Data flow graph

A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.

It   is similar to a control flow graph of a program in that the nodes, edges, and all paths thorough the control flow graph are preserved in the data flow graph. An example follows.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example

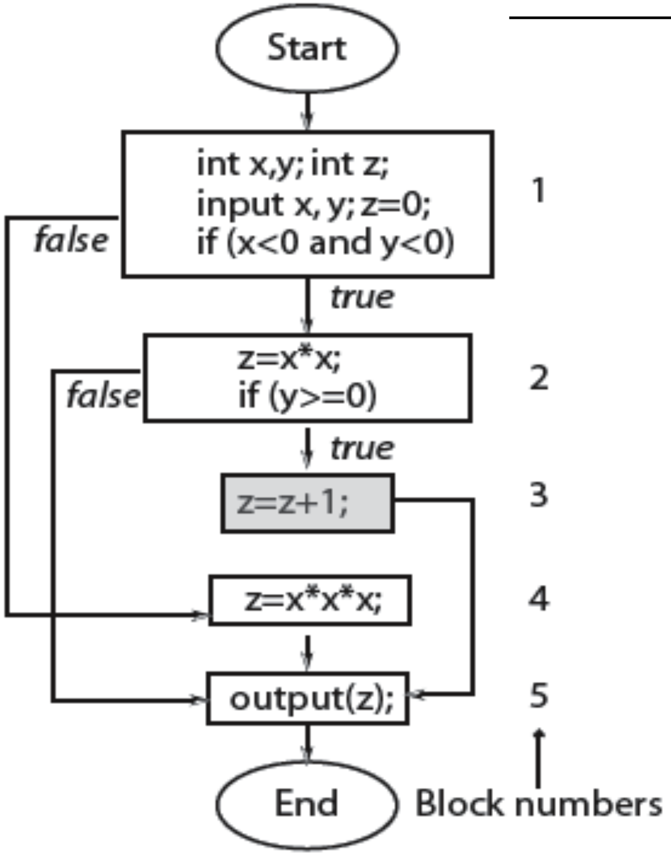
Given a program, find its basic blocks, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).

Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken.

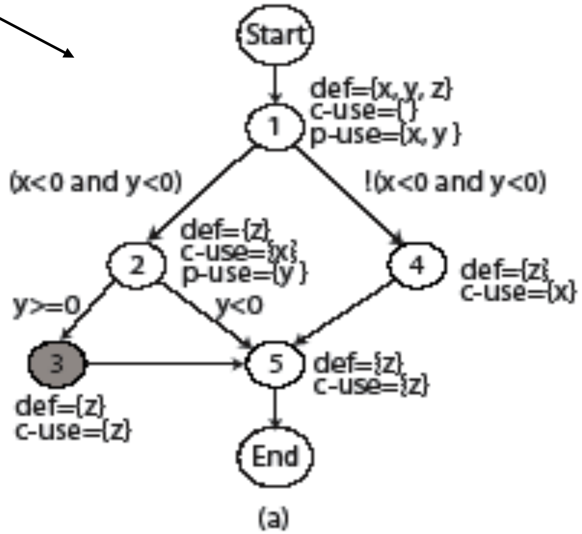
We use $d_i(x)$ to refer to the definition of variable x at node i. Similarly, $u_i(x)$ refers to the use of variable x at node i.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Example (contd.)



Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



Unreachable node

Contents

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.3.5 Def-clear paths

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

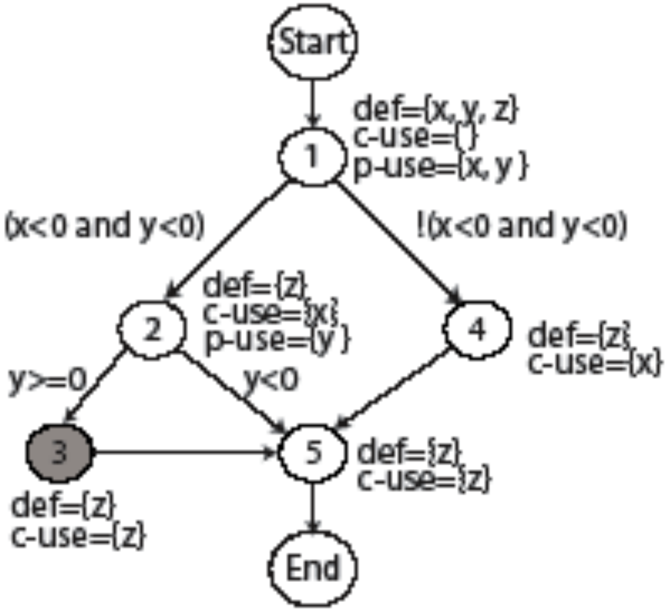
Contents

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a def-clear path for x.

Path 2-5 is def-clear for variable z defined at node 2 and used at node 5. Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus, definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.



(a)

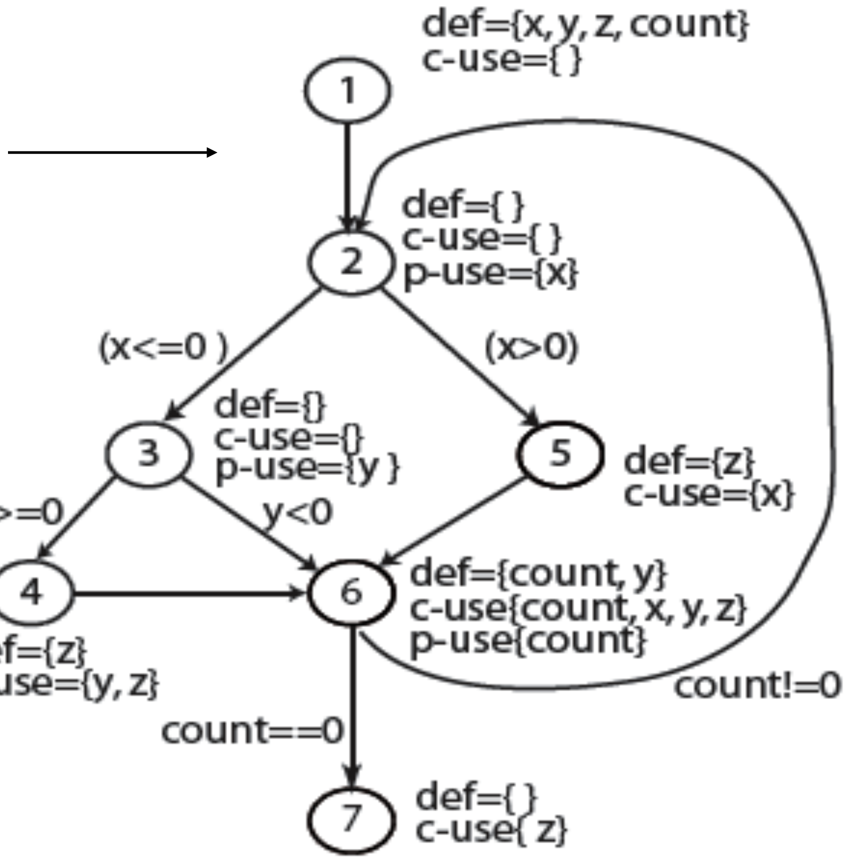
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Def-clear path (another example)

P7.16

```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```



Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Find def-clear paths for defs and uses of x and z.

Which definitions are live at node 4?

Contents

7.3.6 def-use pairs

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Def-use pairs

Def of a variable at line l_1 and its use at line l_2 constitute a def-use pair. l_1 and l_2 can be the same.

dcu (di(x)) denotes the set of all nodes where di(x)) is live and used.

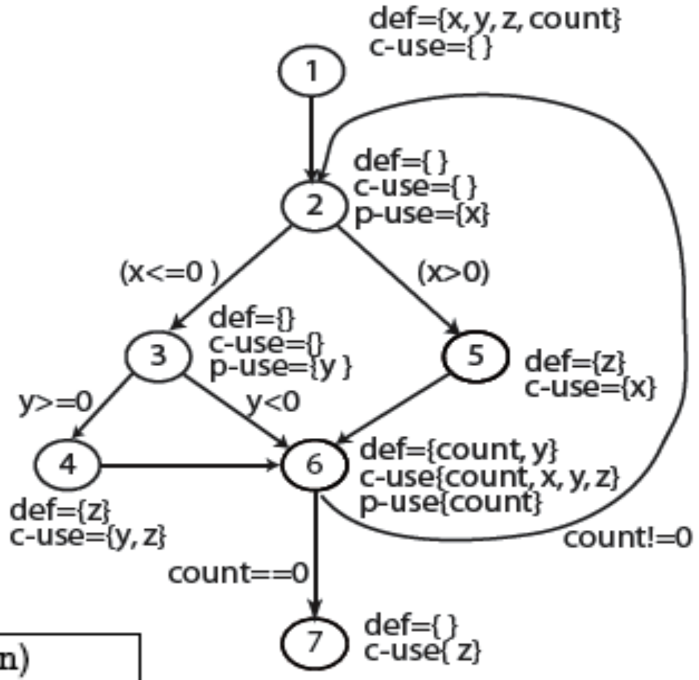
dpu (di(x)) denotes the set of all edges (k, l) such that there is a def-clear path from node i to edge (k, l) and x is used at node k.

We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes i to node j is executed. If $u_j(x))$ is a p-use then all edges of the kind (j, k) must also be taken during some executions.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Def-use pairs (example)



Variable (v)	Defined in node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7 }	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{(6, 2), (6, 7) }
count	6	{6}	{(6, 2), (6, 7) }

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Def-use pairs: Minimal set

Def-use pairs are items to be covered during testing. However, in some cases, coverage of a def-use pair implies coverage of another def-use pair. Analysis of the data flow graph can reveal a minimal set of def-use pairs whose coverage implies coverage of all def-use pairs.

Exercise: Analyze the def-use graph shown in the previous slide and determine a minimal set of def-uses to be covered.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Data flow based adequacy

CU: total number of c-uses in a program.

PU: total number of p-uses.

$$\mathsf{CU} = \Sigma_{i=1}^{n}\Sigma_{j=1}^{d_i} \mid \mathbf{dcu}(v_i, j) \mid$$

$$\mathsf{PU} = \Sigma_{i=1}^{n}\Sigma_{j=1}^{d_i} \mid \mathbf{dpu}(v_i, j) \mid$$

Given a total of n variables v_1, v_2...v_n each defined at d_i nodes.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

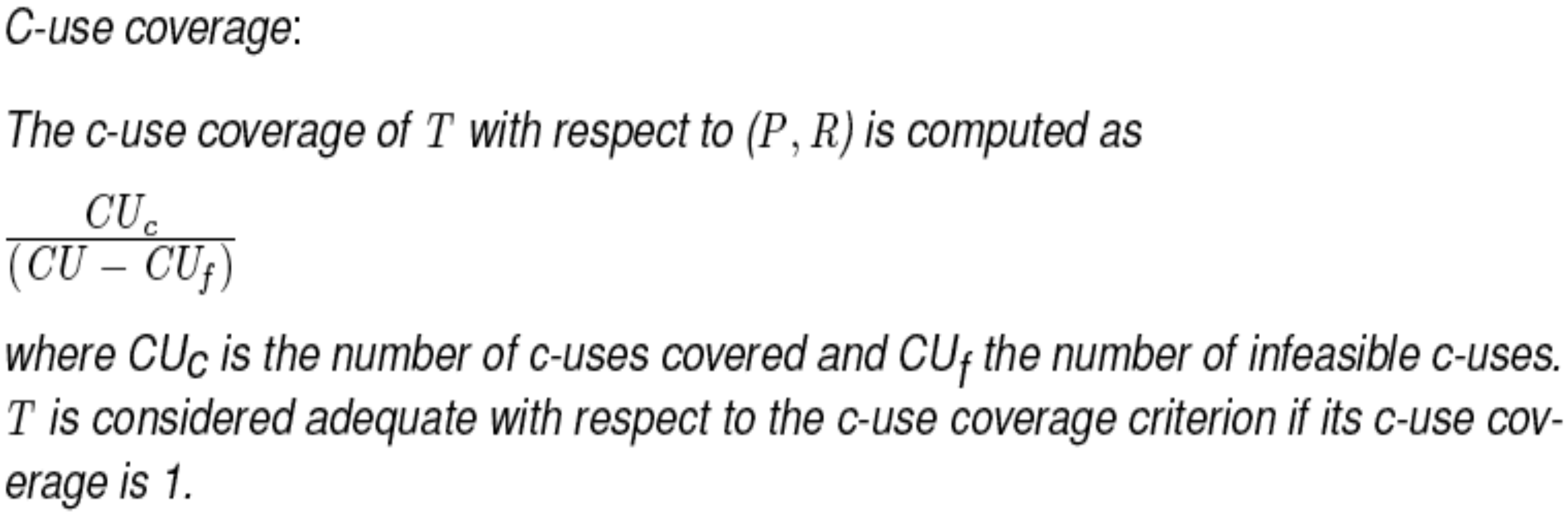
7.4 Adequacy criteria based on data flow

7.4.1, c-use coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C-use coverage



C-use coverage:

The c-use coverage of T with respect to (P, R) is computed as

$$\frac{CU_c}{(CU - CU_f)}$$

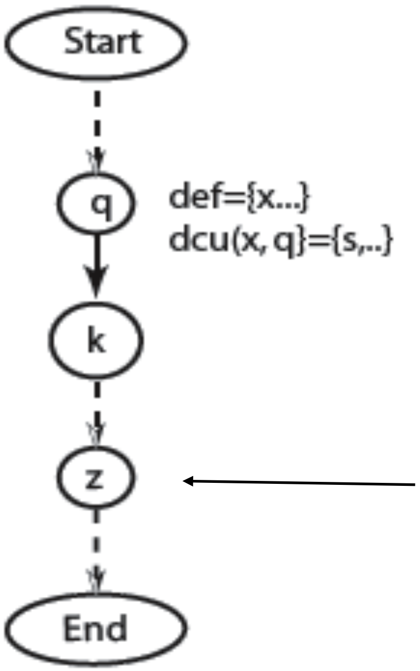
where CU_c is the number of c-uses covered and CU_f the number of infeasible c-uses. T is considered adequate with respect to the c-use coverage criterion if its c-use coverage is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

C-use coverage: path traversed

Path (Start, .. q, k, .., z, .. End) covers the c-use at node z of x defined at node q given that (k …, z) is def clear with respect to x



c-use of x

Exercise: Find the c-use coverage when program P7.16 is executed against the following test:

t1: <x=5, y=-1, count=1>

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

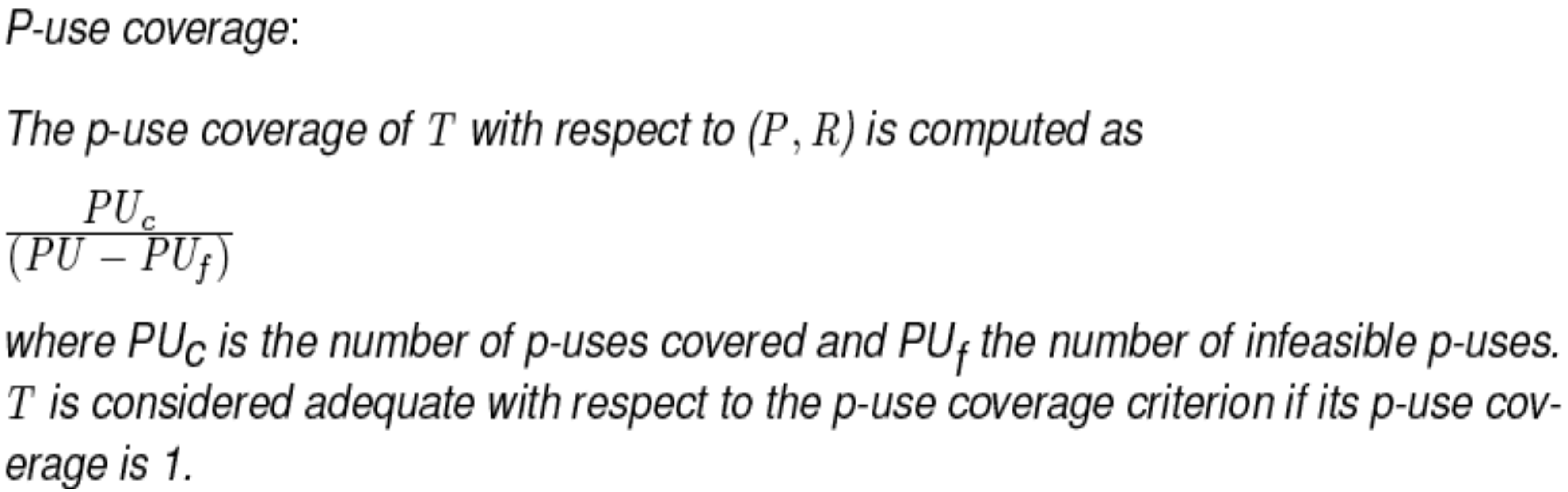
7.4 Adequacy criteria based on data flow

7.4.2 p-use coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

p-use coverage



P-use coverage:

The p-use coverage of T with respect to (P, R) is computed as

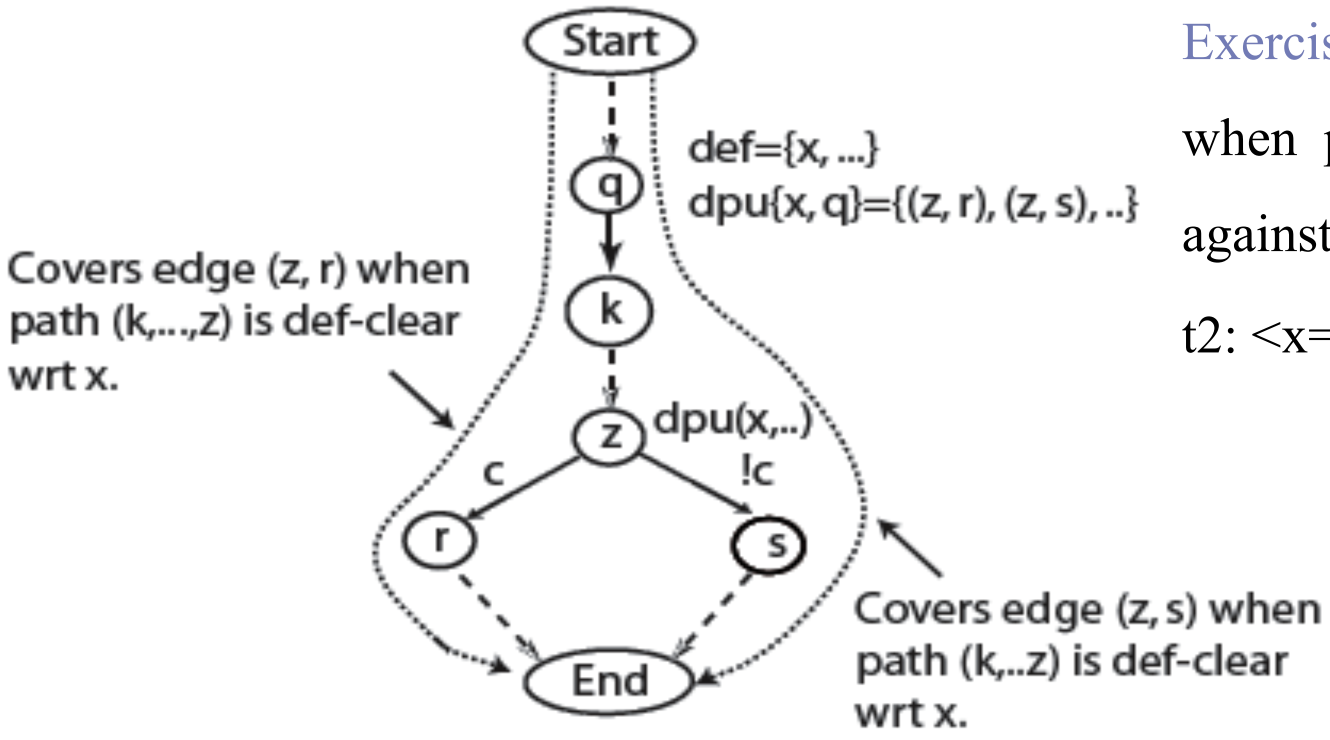
$$\frac{PU_c}{(PU - PU_f)}$$

where PU_c is the number of p-uses covered and PU_f the number of infeasible p-uses. T is considered adequate with respect to the p-use coverage criterion if its p-use coverage is 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

p-use coverage: paths traversed


Start
def={x, ...}
q
dpu{x, q}={(z, r), (z, s), ..}
k
Covers edge (z, r) when path (k,...,z) is def-clear wrt x.
z
dpu(x,..)
c
!c
r
s
Covers edge (z, s) when path (k,..z) is def-clear wrt x.
End

(b)

Exercise: Find the p-use coverage when program P7.16 is executed against the following test:

t2: <x=-2, y=-1, count=3>

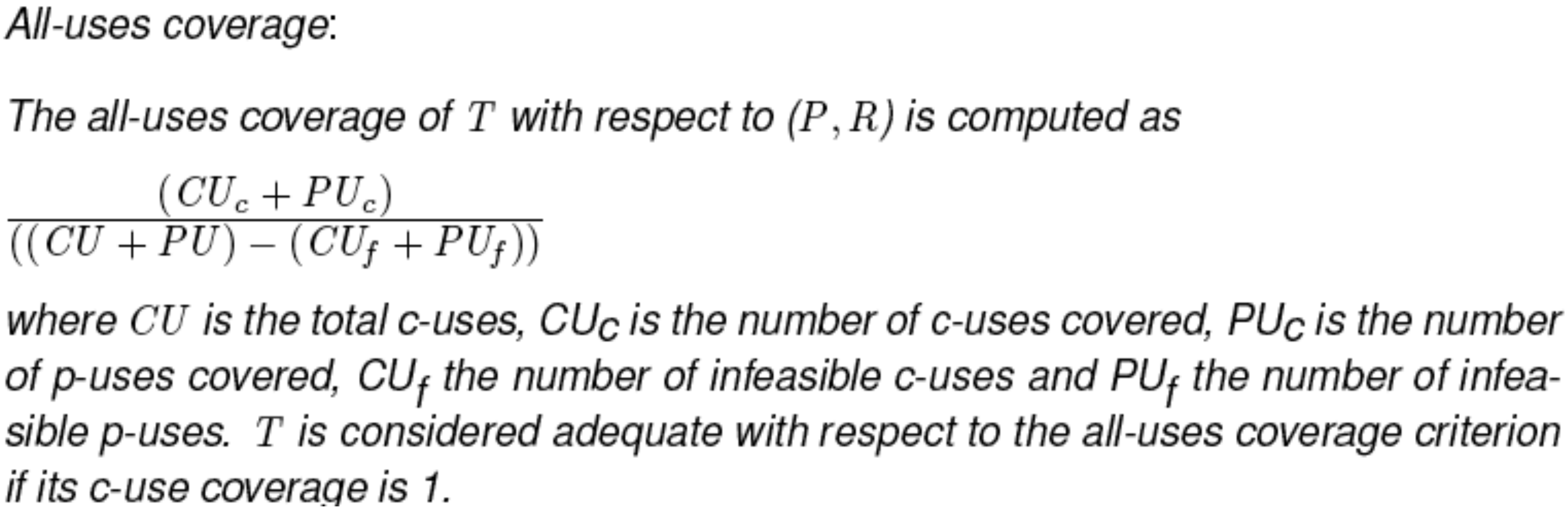
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.4 Adequacy criteria based on data flow
7.4.3, all-uses coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

All-uses coverage



All-uses coverage:

The all-uses coverage of T with respect to (P, R) is computed as

$$\frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f))}$$

where CU is the total c-uses, CU_c is the number of c-uses covered, PU_c is the number of p-uses covered, CU_f the number of infeasible c-uses and PU_f the number of infeasible p-uses. T is considered adequate with respect to the all-uses coverage criterion if its c-use coverage is 1.

Exercise: Is T={t1, t2} adequate w.r.t. to all-uses coverage for P7.16?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Infeasible p- and c-uses

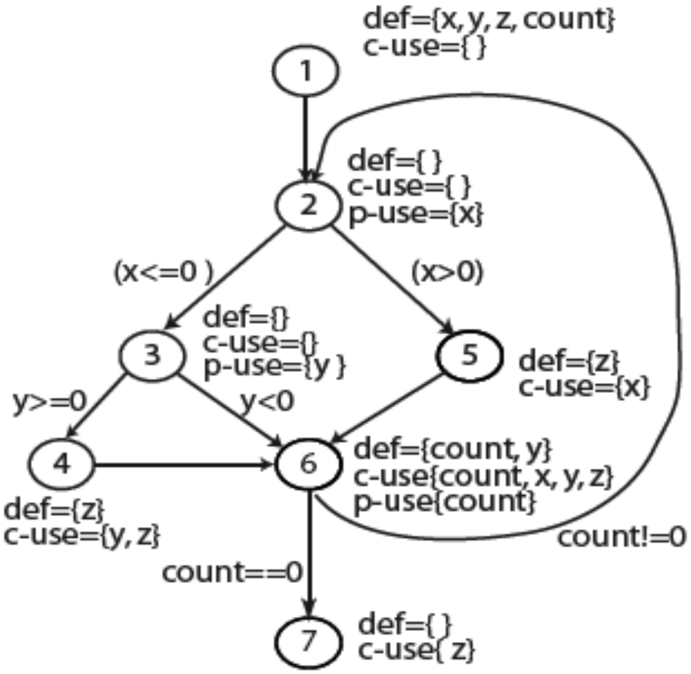
Coverage of a c- or a p-use requires a path to be traversed through the program. However, if this path is infeasible, then some c- and p-uses that require this path to be traversed might also be infeasible.

Infeasible uses are often difficult to determine without some hint from a test tool.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Infeasible c-use: Example



Consider the c-use at node 4 of *z* defined at node 5.

Show that this c-use is infeasible.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.4.4 k-dr chain coverage

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Other data-flow based criteria

There exist several other adequacy criteria based on data flows. Some of these are more powerful in their error-detection effectiveness than the c-, p-, and all-uses criteria.

Examples: (a) def-use chain or k-dr chain coverage. These are alternating sequences of def-use for one or more variables. (b) Data context and ordered data context coverage.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

7.6 The “subsumes” relation

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

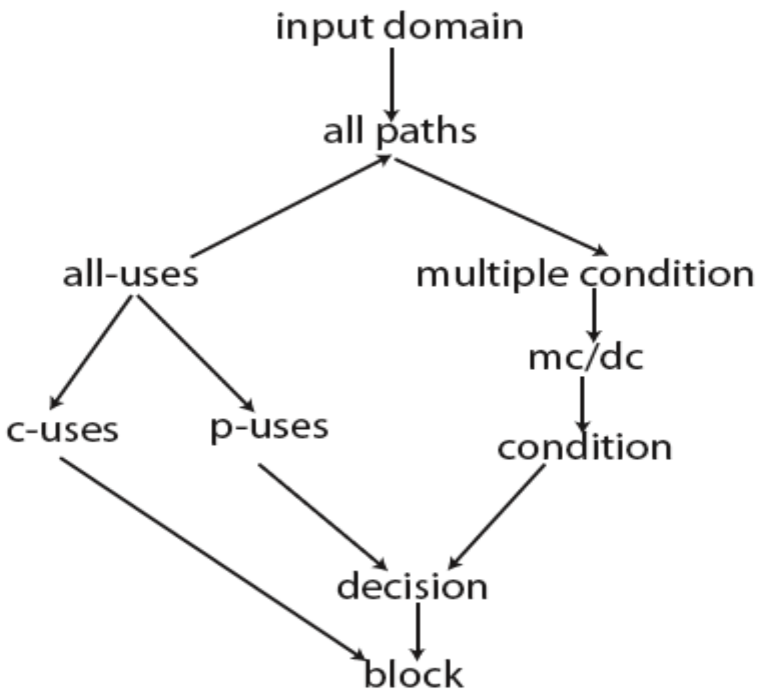
Subsumes relation

Subsumes: Given a test set T that is adequate with respect to criterion C1, what can we conclude about the adequacy of T with respect to another criterion C2?

Effectiveness: Given a test set T that is adequate with respect to criterion C, what can we expect regarding its effectiveness in revealing errors?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Subsumes relationship



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary

We have introduced the notion of test adequacy and enhancement.

Two types of adequacy criteria considered: one based on control flow and the other on data flow.

Control flow based: statement, decision, condition, multiple condition, MC/DC, and LCSAJ coverage. Many more exist.

Data flow based: c-use, p-uses, all-uses, k-dr chain, data context, elementary data context. Many more exist.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary (contd.)

Use of any of the criteria discussed here requires a test tool that measures coverage during testing and displays it in a user-friendly manner. xSUDS is one such set of tools. Several other commercial tools, such as PaRTe, Cobertura, and Bullseye, are available.

Several test organizations believe that code coverage is useful at unit-level. This is a myth and needs to be shattered. Incremental assessment of code coverage and enhancement of tests can allow the application of coverage-based testing to large programs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary (contd.)

Even though coverage is not guaranteed to reveal all program errors, it is the perhaps the most effective way to assess the amount of code that has been tested and what remains untested.

Tests derived using black-box approaches can almost always be enhanced using one or more of the assessment criteria discussed.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 8

Test Adequacy Measurement and Enhancement Using Mutation

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 18, 2013

Contents

Learning Objectives

- What is test adequacy? What is test enhancement? When to measure test adequacy and how to use it to enhance tests?
- What is program mutation?
- Competent programmer hypothesis and the coupling effect.
- Strengths and limitations of test adequacy based on program mutation.
- Mutation operators
- Tools for mutation testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

What is adequacy?

- Consider a program P written to meet a set R of functional requirements. We notate such a P and R as (P, R). Let R contain n requirements labeled R1, R2,…, Rn .

- Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R . Also, P has been executed against each test in T and has produced correct behavior.

- We now ask: Is T good enough? This question can be stated differently as: Has P been tested thoroughly?, or as: Is T adequate?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

What is program mutation?

- Suppose that program P has been tested against a test set T and P has not failed on any test case in T. Now suppose we do the following:

$$P \xrightarrow{\text{Changed to}} P'$$

What behavior do you expect from P' against tests in T?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

What is program mutation? [2]

- P’ is known as a mutant of P.

- There might be a test t in T such that P(t)≠P’(t). In this case we say that t distinguishes P’ from P. Or, that t has killed P’.

- There might be not be any test t in T such that P(t)≠P’(t). In this case we say that T is unable to distinguish P and P’. Hence P’ is considered live in the test process.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

What is program mutation? [3]

- If there does not exist any test case t in the input domain of P that distinguishes P from P' then P' is said to be equivalent to P.
- If P' is not equivalent to P but no test in T is able to distinguish it from P then T is considered inadequate.
- A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence enhance T.

We will refer to program mutation as mutation.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test adequacy using mutation [1]

- Given a test set T for program P that must meet requirements R, a test adequacy assessment procedure proceeds as follows.

- Step 1: Create a set M of mutants of P. Let M={M_0, M_1…M_k}. Note that we have k mutants.

- Step 2: For each mutant M_i find if there exists a t in T such that $M_i(t) \neq P(t)$. If such a t exists then M_i is considered killed and removed from further consideration.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test adequacy using mutation [2]

- Step 3: At the end of Step 2 suppose that $k_1 \leq k$ mutants have been killed and $(k-k_1)$ mutants are live.

Case 1: $(k-k_1)=0$: T is adequate with respect to mutation.

Case 2: $(k-k_1)>0$ then we compute the mutation score (MS) as follows:

$$MS=k_1/(k-e)$$

where e is the number of equivalent mutants. Note: $e \leq (k-k_1)$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test enhancement using mutation

- One has the opportunity to enhance a test set T after having assessed its adequacy.

- Step 1: If the mutation score (MS) is 1, then some other technique, or a different set of mutants, needs to be used to help enhance T.

- Step 2: If the mutation score (MS) is less than 1, then there exist live mutants that are not equivalent to P. Each live mutant needs to be distinguished from P.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test enhancement using mutation [2]

- Step 3: Hence a new test t is designed with the objective of distinguishing at least one of the live mutants; let us say this mutant is m.

- Step 4: If t does not distinguish m then another test t' needs to be designed to distinguish m. Suppose that t does distinguish m.

- Step 5: It is possible that t also distinguishes other live mutants.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test enhancement using mutation [3]

- Step 6: Add t to T and re-compute the mutation score (MS).
- Repeat the enhancement process from Step 1.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation

- As with any test enhancement technique, there is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P. Nevertheless, empirical studies have found to be the most powerful of all formal test enhancement techniques.

- The next simple example illustrates how test enhancement using mutation detects errors.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [2]

- Consider the following function foo that is required to return the sum of two integers x and y. Clearly foo is incorrect.

```
int foo(int x, y){
return (x-y);
}
```

← This should be return (x+y)

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [3]

- Now suppose that foo has been tested using a test set T that contains two tests:

  T={ t1: <x=1, y=0>, t2: <x=-1, y=0>}

- First note that foo behaves perfectly fine on each test in, i.e. foo returns the expected value for each test case in T. Also, T is adequate with respect to all control and data flow based test adequacy criteria.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [4]

Let us evaluate the adequacy of T using mutation. Suppose that the following three mutants are generated from foo.

M1:

```
int foo(int x, y){
return (x+y);
}
```

M2:

```
int foo(int x, y){
return (x-0);
}
```

M3:

```
int foo(int x, y){
return (0+y);
}
```

- Note that M1 is obtained by replacing the - operator by a + operator, M2 by replacing y by 0, and M3 by replacing x by 0.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [4]

Next we execute each mutant against tests in T until the mutant is distinguished or we have exhausted all tests. Here is what we get.

T={ t1: <x=1, y=0>, t2: <x=-1, y=0>}

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	Killed

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [5]

After executing all three mutants we find that two are live and one is distinguished. Computation of mutation score requires us to determine of any of the live mutants is equivalent.

In class exercise: Determine whether or not the two live mutants are equivalent to foo and compute the mutation score of T.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [6]

Let us examine the following two live mutants.

```
M1:  int foo(int x, y){          M2:  int foo(int x, y){
     return (x+y);                    return (x-0);
     }                                }
```

Let us focus on M1. A test that distinguishes M1 from foo must satisfy the following condition:

$x-y \neq x+y$ implies $y \neq 0$.

Hence we get t3: <x=1, y=1>

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Error detection using mutation [7]

Executing foo on t3 gives us foo(t3)=0. However, according to the requirements we must get foo(t3)=2. Thus, t3 distinguishes M1 from foo and also reveals the error.

M1:

```
int foo(int x, y){
return (x+y);
}
```

M2:

```
int foo(int x, y){
return (x-0);
}
```

In class exercise: (a) Will any test that distinguishes also reveal the error? (b) Will any test that distinguishes M2 reveal the error?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Guaranteed error detection

Sometimes there exists a mutant P' of program P such that any test t that distinguishes P' from P also causes P to fail. More formally:

> Let P' be a mutant of P and t a test in the input domain of P. We say that P' is an error revealing mutant if the following condition holds for any t.
>
> P'(t) ≠P(t) and P(t) ≠R(t), where R(t) is the expected response of P based on its requirements.

Is M1 in the previous example an error revealing mutant? What about M2?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Distinguishing a mutant

A test case t that distinguishes a mutant m from its parent program P program must satisfy the following three conditions:

Condition 1: Reachability: t must cause m to follow a path that arrives at the mutated statement in m.

Condition 2: Infection: If S_{in} is the state of the mutant upon arrival at the mutant statement and S_{out} the state soon after the execution of the mutated statement, then $S_{in} \neq S_{out}$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Distinguishing a mutant [2]

Condition 3: Propagation: If difference between S_{in} and S_{out} must propagate to the output of m such that the output of m is different from that of P.

Exercise: Show that in the previous example both t1 and t2 satisfy the above three conditions for M3.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Equivalent mutants

- The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable. Hence there is no way to fully automate the detection of equivalent mutants.

- The number of equivalent mutants can vary from one program to another. However, empirical studies have shown that one can expect about 5% of the generated mutants to the equivalent to the parent program.

- Identifying equivalent mutants is generally a manual and often time consuming--as well as frustrating--process.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

A misconception

There is a widespread misconception amongst testing educators, researchers, and practitioners that any “coverage” based technique, including mutation, will not be able to detect errors due to missing path. Consider the following programs.

Program under test

```
int foo(int x, y){
int p=0;
if(x<y)                ←    Missing else
        p=p+1;
return(x+p*y)
}
```

Correct program

```
int foo(int x, y){
int p=0;
if(x<y)
        p=p+1;
else
        p=p-1;
return(x+p*y)
}
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

A misconception [2]

(a) Suggest at least one mutant M of foo that is guaranteed to reveal the error; in other words M is an error revealing mutant.

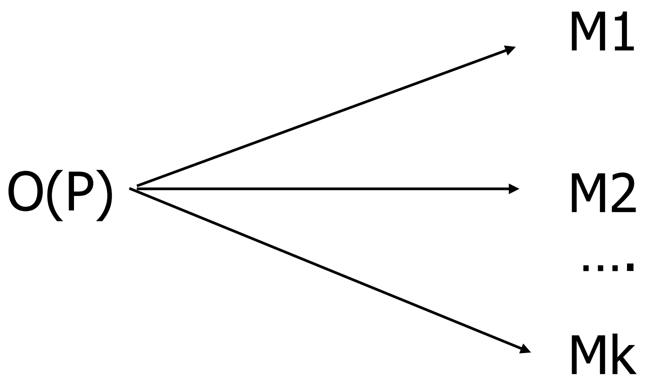
(b) Suppose T is decision adequate for foo. Is T guaranteed to reveal the error?

(c) Suppose T is def-use adequate for foo. Is T guaranteed to reveal the error?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutant operators

- A mutant operator O is a function that maps the program under test to a set of k (zero or more) mutants of P.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutant operators [2]

- A mutant operator creates mutants by making simple changes in the program under test.

- For example, the “variable replacement” mutant operator replaces a variable name by another variable declared in the program. An “relational operator replacement” mutant operator replaces relational operator wirh another relational operator.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutant operators: Examples

Mutant operator	In P	In mutant

Variable replacement	z=x*y+1;	x=x*y+1;
		z=x*x+1;

Relational operator replacement	if (x<y)	if(x>y)
		if(x<=y)

Off-by-1	z=x*y+1;	z=x*(y+1)+1;
		z=(x+1)*y+1;

Replacement by 0	z=x*y+1;	z=0*y+1;
		z=0;

Arithmetic operator replacement	z=x*y+1;	z=x*y-1;
		z=x+y-1;

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mutants: First order and higher order

- A mutant obtained by making exactly “one change” is considered first order.
- A mutant obtained by making two changes is a second order mutant. Similarly higher order mutants can be defined. For example, a second order mutant of z=x+y; is x=z+y; where the variable replacement operator has been applied twice.
- In practice only first order mutants are generated for two reasons: (a) to lower the cost of testing and (b) most higher order mutants are killed by tests adequate with respect to first order mutants. [See coupling effect later.]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mutant operators: basis

- A mutant operator models a simple mistake that could be made by a programmer

- Several error studies have revealed that programmers--novice and experts--make simple mistakes. For example, instead of using x<y+1 one might use x<y.

- While programmers make "complex mistakes" too, mutant operators model simple mistakes. As we shall see later, the "coupling effect" explains why only simple mistakes are modeled.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mutant operators: Goodness

- The design of mutation operators is based on guidelines and experience. It is Thus, evident that two groups might arrive at a different set of mutation operators for the same programming language. How should we judge whether or not that a set of mutation operators is “good enough?”

- Informal definition:
    - Let S1 and S2 denote two sets of mutation operators for language L. Based on the effectiveness criteria, we say that S1 is superior to S2 if mutants generated using S1 guarantee a larger number of errors detected over a set of erroneous programs.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mutant operators: Goodness [2]

- Generally one uses a small set of highly effective mutation operators rather than the complete set of operators.
- Experiments have revealed relatively small sets of mutation operators for C and Fortran. We say that one is using "constrained" or "selective" mutation when one uses this small set of mutation operators.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutant operators: Language dependence

- For each programming language one develops a set of mutant operators.
- Languages differ in their syntax thereby offering opportunities for making mistakes that duffer between two languages. This leads to differences in the set of mutant operators for two languages.
- Mutant operators have been developed for languages such as Fortran, C, Ada, Lisp, and Java. [See the text for a comparison of mutant operators across several languages.]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Competent programmer hypothesis (CPH)

- CPH states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.
- An extreme interpretation of CPH is that when asked to write a program to find the account balance, given an account number, a programmer is unlikely to write a program that deposits money into an account. Of course, while such a situation is unlikely to arise, a devious programmer might certainly write such a program.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Competent programmer hypothesis (CPH) [2]

- A more reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program.

- The CPH assumes that the programmer knows of an algorithm to solve the problem at hand, and if not, will find one prior to writing the program.

- It is Thus, safe to assume that when asked to write a program to sort a list of numbers, a competent programs knows of, and makes use of, at least one sorting algorithm. Mistakes will lead to a program that can be corrected by applying one or more first order mutations.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Coupling effect

- The coupling effect has been paraphrased by DeMillo, Lipton, and Sayward as follows: “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors”
- Stated alternately, again in the words of DeMillo, Lipton and Sayward ``..seemingly simple tests can be quite sensitive via the coupling effect."

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Coupling effect [2]

- For some input, a non-equivalent mutant forces a slight perturbation in the state space of the program under test. This perturbation takes place at the point of mutation and has the potential of infecting the entire state of the program.
- It is during an analysis of the behavior of the mutant in relation to that of its parent that one discovers complex faults.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Tools for mutation testing

- As with any other type of test adequacy assessment, mutation based assessment must be done with the help of a tool.

- There are few mutation testing tools available freely. Two such tools are Proteum for C from Professor Josè Maldonado and muJava for Java from Professor Jeff Offutt. We are not aware of any commercially available tool for mutation testing. See the textbook for a more complete listing of mutation tools.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Tools for mutation testing: Features

- A typical tool for mutation testing offers the following features.

  - A selectable palette of mutation operators.
  - Management of test set T.
  - Execution of the program under test against T and saving the output for comparison against that of mutants.
  - Generation of mutants.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Tools for mutation testing: Features [2]

- Mutant execution and computation of mutation score using user identified equivalent mutants.
- Incremental mutation testing: i.e. allows the application of a subset of mutation operators to a portion of the program under test.
- Mothra, an advanced mutation tool for Fortran also provided automatic test generation using DeMillo and Offutt' s method.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Mutation and system testing

- Adequacy assessment using mutation is often recommended only for relatively small units, e.g. a class in Java or a small collection of functions in C.

- However, given a good tool, one can use mutation to assess adequacy of system tests.

- The following procedure is recommended to assess the adequacy of system tests.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutation and system testing [2]

- Step 1: Identify a set U of application units that are critical to the safe and secure functioning of the application. Repeat the following steps for each unit in U.
- Step 2: Select a small set of mutation operators. This selection is best guided by the operators defined by Eric Wong or Jeff Offutt. [See the text for details.]
- Step 3: Apply the operators to the selected unit.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutation and system testing [3]

- Step 4: Assess the adequacy of T using the mutants so generated. If necessary, enhance T.

- Step 5: Repeat Steps 3 and 4 for the next unit until all units have been considered.

    - We have now assessed T, and perhaps enhanced it. Note the use of incremental testing and constrained mutation (i.e., use of a limited set of highly effective mutation operators).

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Mutation and system testing [4]

- Application of mutation, and other advanced test assessment and enhancement techniques, is recommended for applications that must meet stringent availability, security, safety requirements.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary

- Mutation testing is the most powerful technique for the assessment and enhancement of tests.
- Mutation, as with any other test assessment technique, must be applied incrementally and with assistance from good tools.
- Identification of equivalent mutants is an undecidable problem--similar the identification of infeasible paths in control or data flow based test assessment.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary [2]

- While mutation testing is often recommended for unit testing, when done carefully and incrementally, it can be used for the assessment of system and other types of tests applied to an entire application.

- Mutation is a highly recommended technique for use in the assurance of quality of highly available, secure, and safe systems.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Chapter 9

Test Selection, Minimization, and Prioritization for Regression Testing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Updated: July 17, 2013

Contents

Learning Objectives

What is regression testing?

How to select a subset of tests for regression testing?

How to select or minimize a set of tests for regression testing?

How to prioritize a set of tests for regression testing?

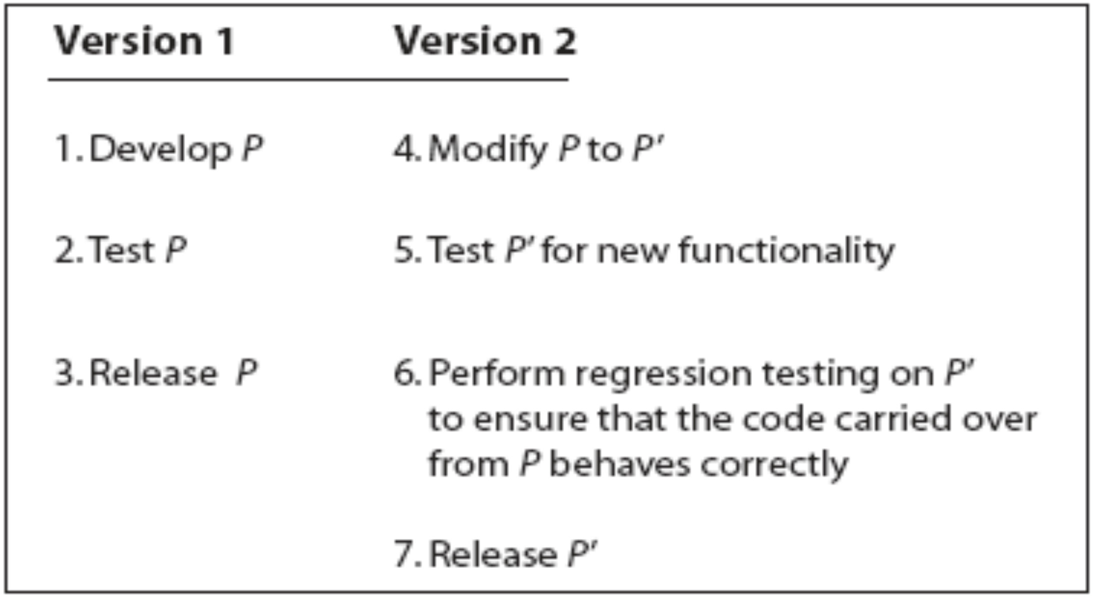
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

9.1. What is regression testing?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Regression testing



Version 1	Version 2
1. Develop *P*	4. Modify *P* to *P′*
2. Test *P*	5. Test *P′* for new functionality
3. Release *P*	6. Perform regression testing on *P′* to ensure that the code carried over from *P* behaves correctly
	7. Release *P′*

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

What tests to use?

Idea 1:

All valid tests from the previous version and new tests created to test any added functionality. [This is the TEST-ALL approach.]

What are the strengths and shortcomings of this approach?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

The test-all approach

The test-all approach is best when you want to be certain that the the new version works on all tests developed for the previous version and any new tests.

But what if you have limited resources to run tests and have to meet a deadline? What if running all tests as well as meeting the deadline is simply not possible?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test selection

Idea 2:

Select a subset Tr of the original test set T such that successful execution of the modified code P' against Tr implies that all the functionality carried over from the original code P to P 'is intact.

Finding Tr can be done using several methods. We will discuss two of these known as test minimization and test prioritization.

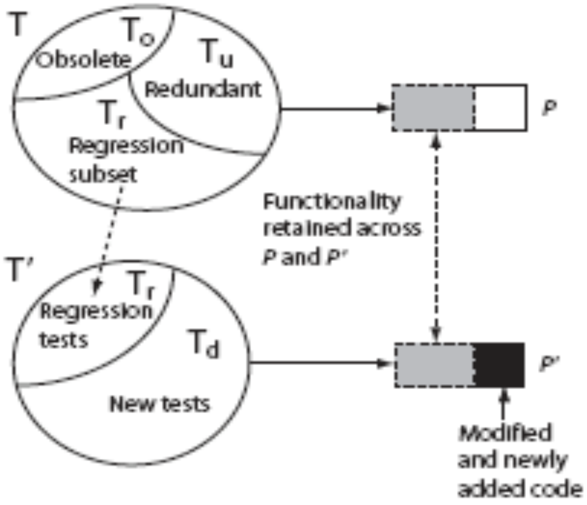
Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

9.3. Regression test selection: The problem

Contents

Regression Test Selection problem



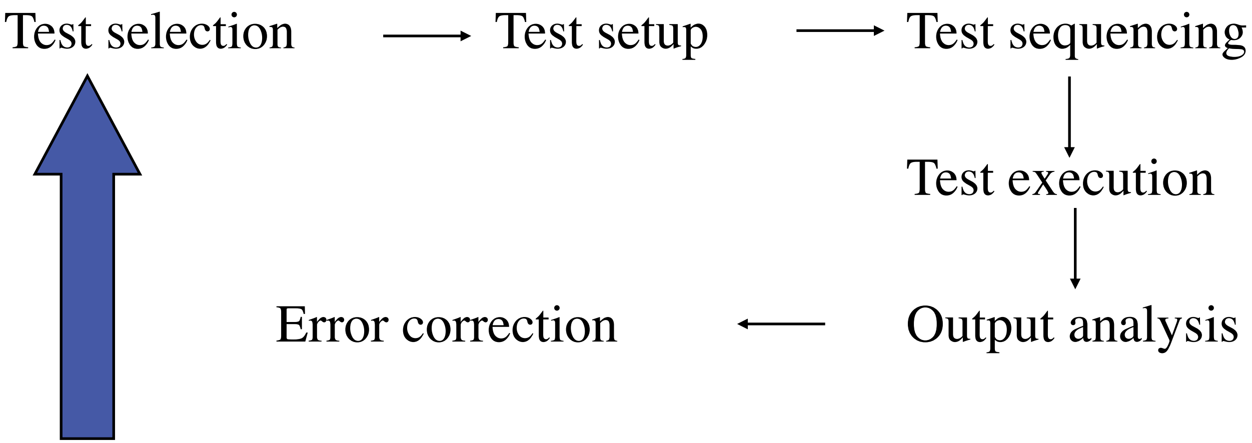
Given test set T, our goal is to determine Tr such that successful execution of P' against Tr implies that modified or newly added code in P' has not broken the code carried over from P.

Note that some tests might become obsolete when P is modified to P' . Such tests are not included in the regression subset Tr. The task of identifying such obsolete tests is known as test revalidation.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Regression Test Process

Now that we know what the regression test selection problem is, let us look at an overall regression test process.



Test selection

Test setup

Test sequencing

Test execution

Output analysis

Error correction

In this chapter we will learn how to select tests for regression testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

9.5. Test selection using execution trace

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Overview of a test selection method

Step 1: Given P and test set T, find the execution trace of P for each test in T.

Step 2: Extract test vectors from the execution traces for each node in the CFG of P

Step 3: Construct syntax trees for each node in the CFGs of P and P' . This step can be executed while constructing the CFGs of P and P'.

Step 4: Traverse the CFGs and determine the a subset of T appropriate for regression testing of P' .

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Execution Trace [1]

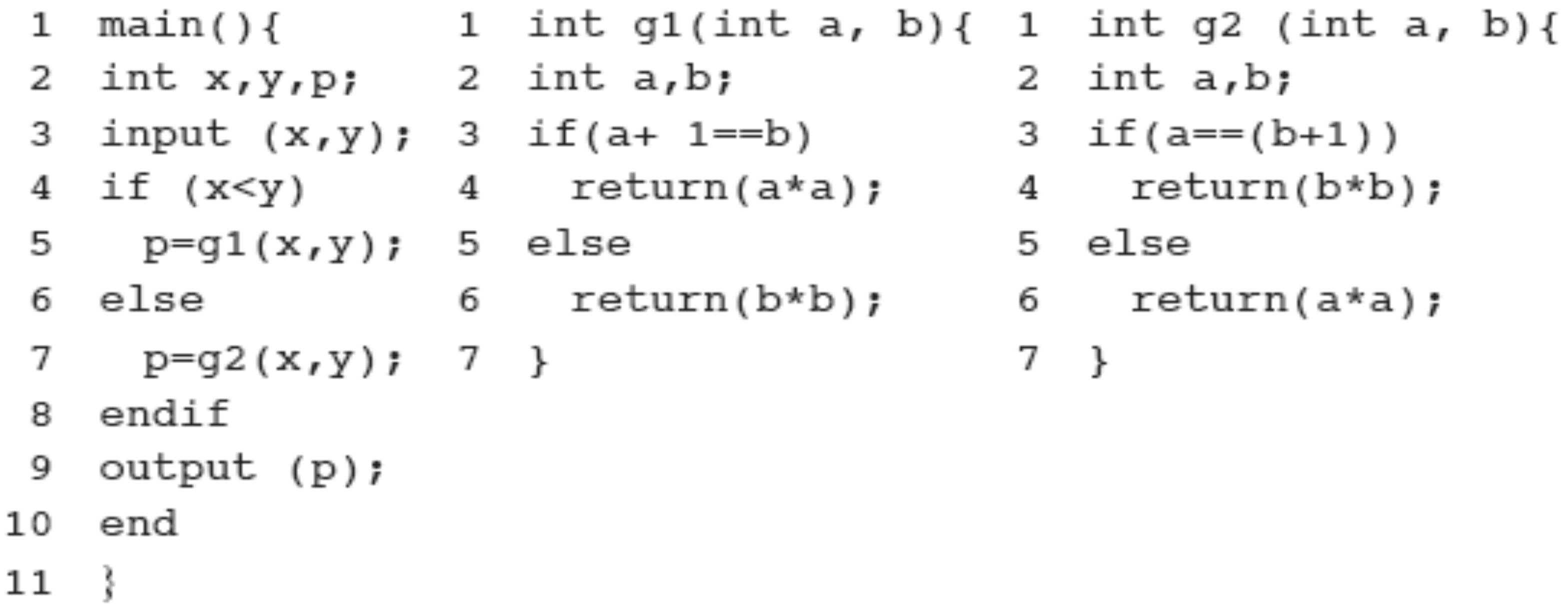
Let G=(N, E) denote the CFG of program P. N is a finite set of nodes and E a finite set of edges connecting the nodes. Suppose that nodes in N are numbered 1, 2, and so on and that Start and End are two special nodes as discussed in Chapter 1.

Let T_{no} be the set of all valid tests for P'. Thus, T_{no} contains only tests valid for P'. It is obtained by discarding all tests that have become obsolete for some reason.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Execution Trace [2]

An execution trace of program P for some test t in T_{no} is the sequence of nodes in G traversed when P is executed against t. As an example, consider the following program.



```
main(){
int x,y,p;
input (x,y);
if (x<y)
  p=g1(x,y);
else
  p=g2(x,y);
endif
output (p);
end
}
```

```
int g1(int a, b){
int a,b;
if(a+ 1==b)
  return(a*a);
else
  return(b*b);
}
```

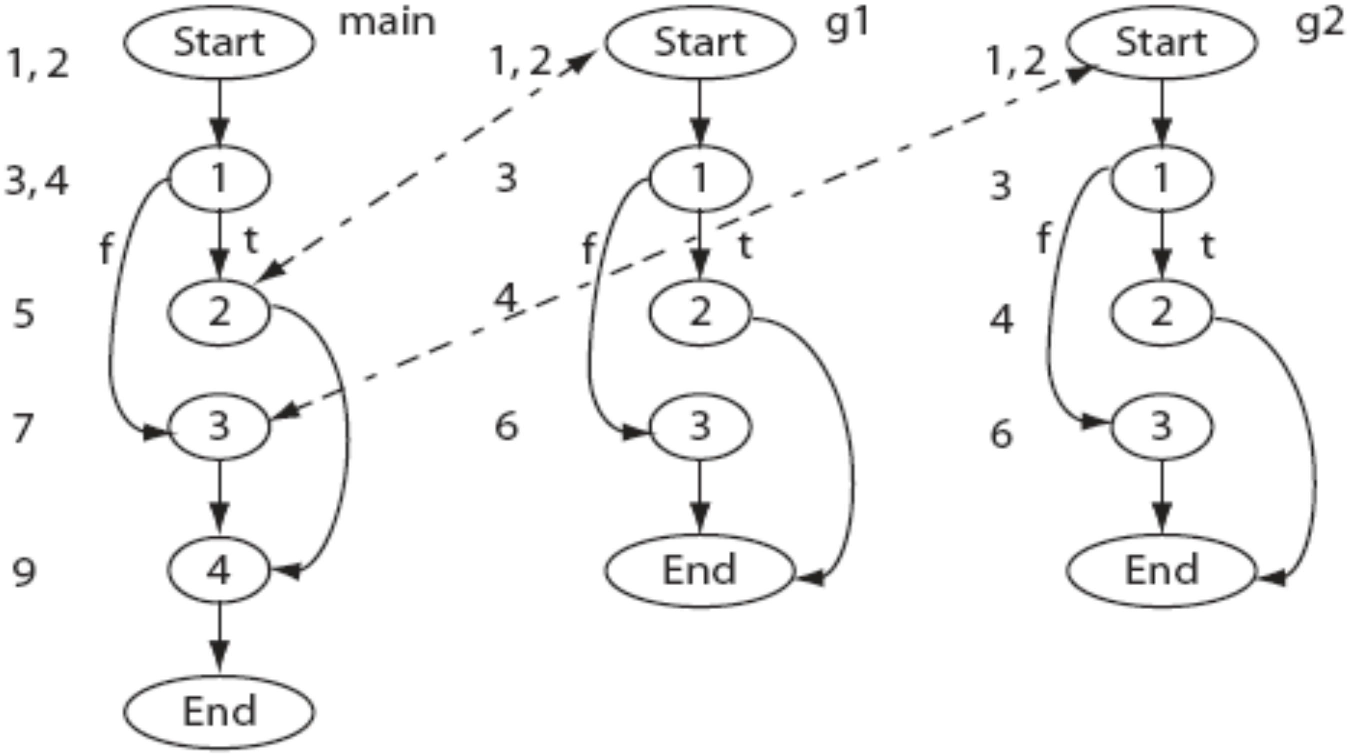
```
int g2 (int a, b){
int a,b;
if(a==(b+1))
  return(b*b);
else
  return(a*a);
}
```

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Execution Trace [3]

Here is a CFG for our example program.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Execution Trace [4]

Now consider the following set of three tests and the corresponding trace.

$$T = \left\{ \begin{array}{l} t_1 :< x = 1, y = 3 > \\ t_2 :< x = 2, y = 1 > \\ t_3 :< x = 3, y = 1 > \end{array} \right\}$$

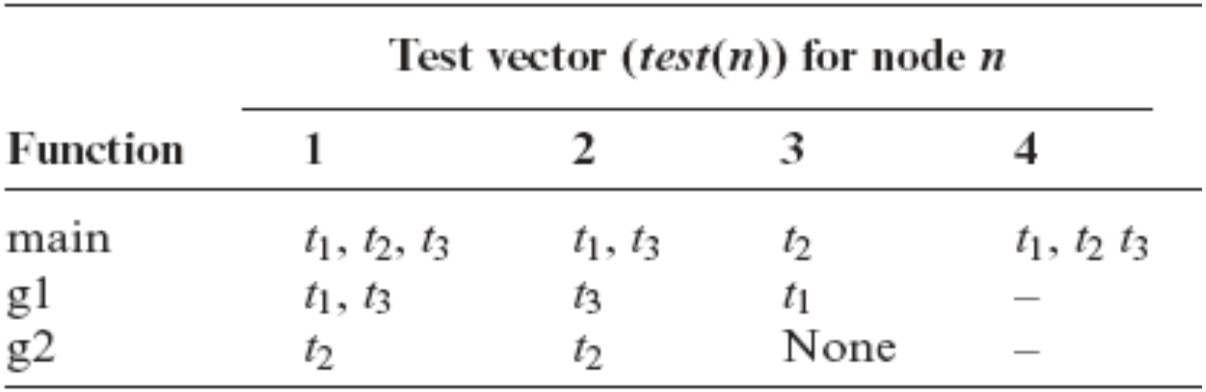
Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test vector

A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG. For program P we obtain the following test vectors.



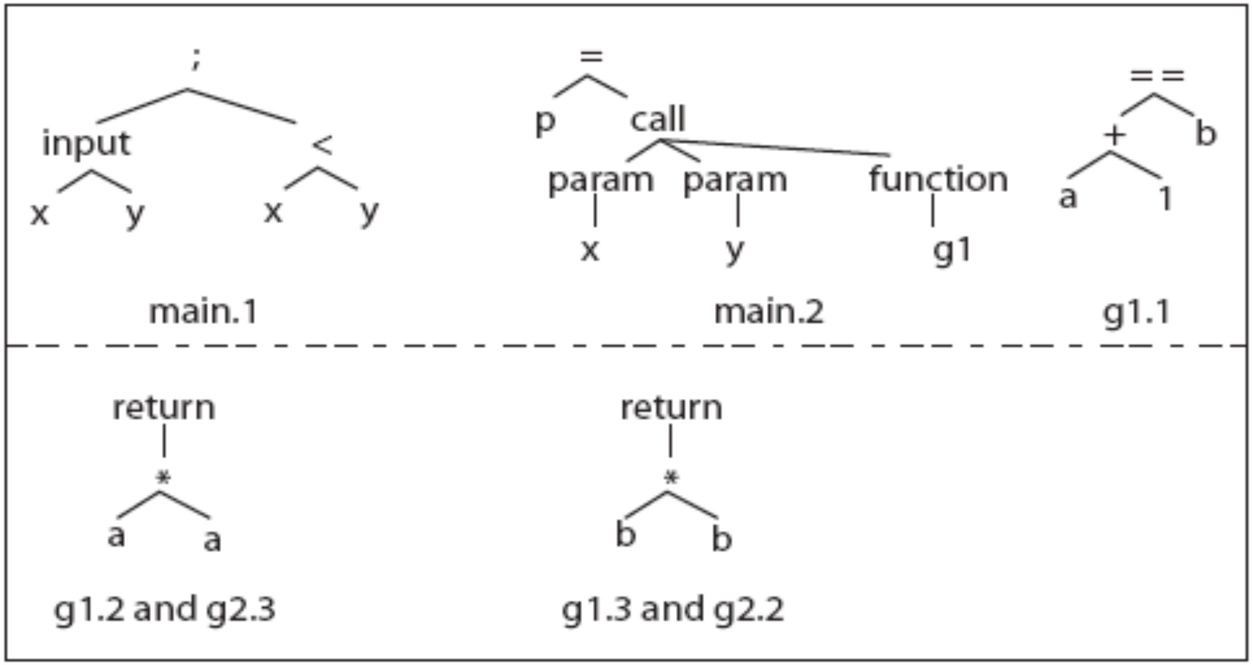
	Test vector (*test*(*n*)) for node *n*			
Function	**1**	**2**	**3**	**4**
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2 t_3
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Syntax trees

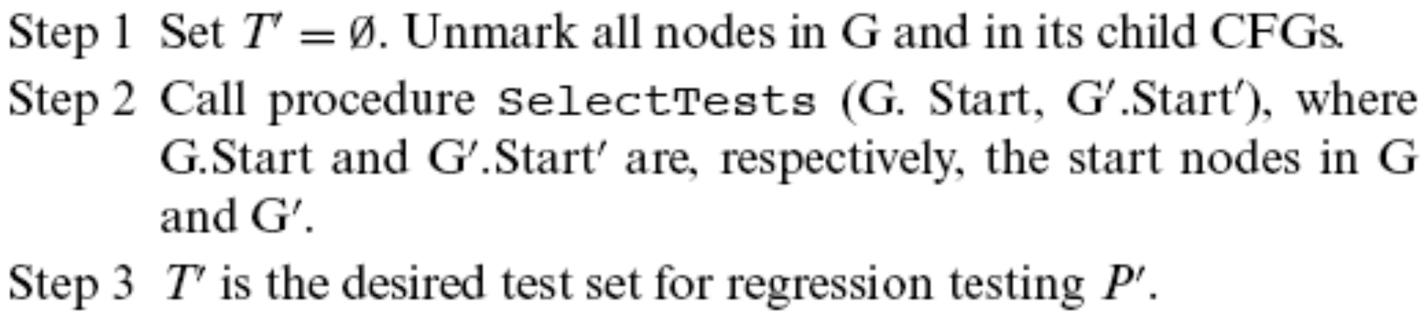
A syntax tree is constructed for each node of CFG(P) and CFG(P’). Recall that each node represents a basic block. Here sample syntax trees for the example program.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test selection [1]

Given the execution traces and the CFGs for P and P’ , the following three steps are executed to obtain a subset T’ of T for regression testing of P’ .



Step 1 Set $T' = \emptyset$. Unmark all nodes in G and in its child CFGs.

Step 2 Call procedure `SelectTests` (G. Start, G′.Start′), where G.Start and G′.Start′ are, respectively, the start nodes in G and G′.

Step 3 T' is the desired test set for regression testing P'.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test selection [2]

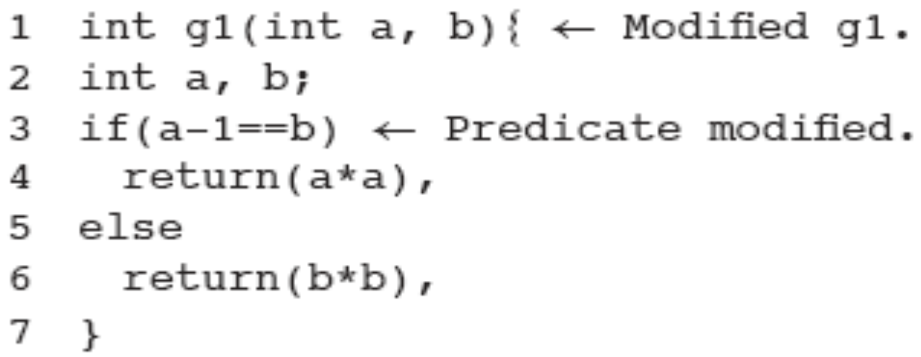
The basic idea underlying the SelectTests procedure is to traverse the two CFGs from their respective START nodes using a recursive descent procedure.

The descent proceeds in parallel and the corresponding nodes are compared. If two two nodes N in CFG(P) and N' in CFG(P') are found to be syntactically different, all tests in test (N) are added to T' .

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test selection example

Suppose that function g1 in P is modified as follows.



```
int g1(int a, b){ ← Modified g1.
int a, b;
if(a-1==b) ← Predicate modified.
  return(a*a),
else
  return(b*b),
}
```

Try the SelectTests algorithm and check if you get T’ ={t1, t3}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Issues with SelectTests

Think:

What tests will be selected when only, say, one declaration is modified?

Can you think of a way to select only tests that correspond to variables in the modified declaration?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

9.6. Test selection using dynamic slicing

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Dynamic slice

Let L be a location in program P and v a variable used at L.

Let trace(t) be the execution trace of P when executed against test t.

The dynamic slice of P with respect to t and v, denoted as DS(t, v, L), is the set of statements in P that (a) lie in trace(t) and (b) effected the value of v at L.

Question: What is the dynamic slice of P with respect to v and t if L is not in trace(t)?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Dynamic dependence graph (DDG)

The DDG is needed to obtain a dynamic slice. Here is how a DDG G is constructed.

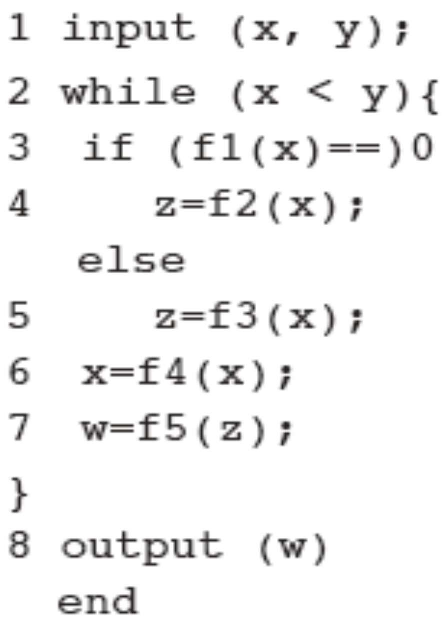
Step 1: Initialize G with a node for each declaration. There are no edges among these nodes.

Step 2: Add to G the first node in trace(t).

Step 3: For each successive statement in trace(t) a new node n is added to G. Control and data dependence edges are added from n to the existing nodes in G. [Recall from Chapter 2 the definitions of control and data dependence edges.]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Construction of a DDG: Example [1]



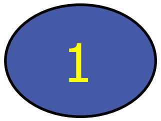
```
1 input (x, y);
2 while (x < y){
3  if (f1(x)==)0
4      z=f2(x);
   else
5      z=f3(x);
6  x=f4(x);
7  w=f5(z);
}
8 output (w)
  end
```

Let t: <x=2, y=4>

Assume successive values of x to be 2, 0 and 5, and for these values f1(x) is 0, 2, and 3 respectively.

trace(t)={1, 2, 3, 4, 6, 7, 2, 3, 5, 6, 7, 2, 8}

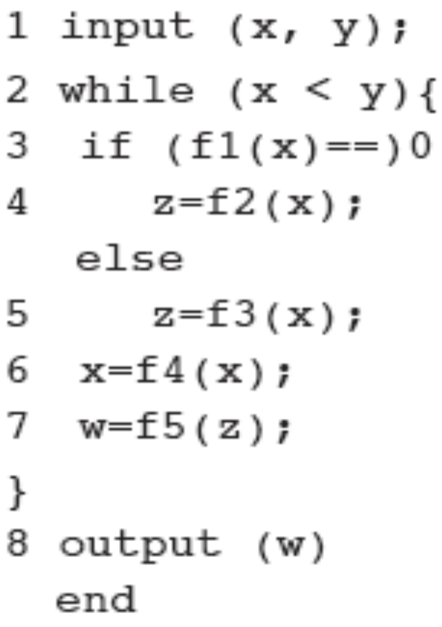
Ignore declarations for simplicity. Add a node to G corresponding to statement 1.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construction of a DDG: Example [2]



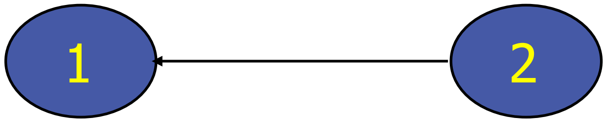
```
1 input (x, y);
2 while (x < y){
3  if (f1(x)==)0
4      z=f2(x);
   else
5      z=f3(x);
6  x=f4(x);
7  w=f5(z);
}
8 output (w)
  end
```

should be…

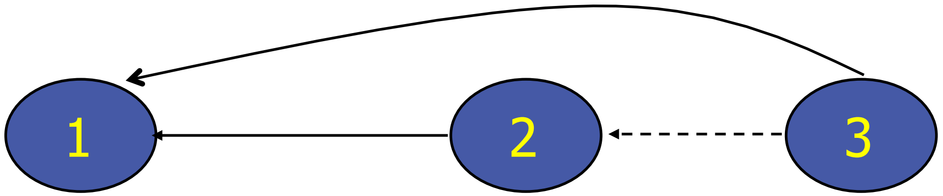
3 if(f1(x)==0)

trace(t)={1, 2, 3, 4, 6, 7, 2, 3, 5, 6, 7, 2, 8}

Add another node corresponding to statement 2 in trace(t). Also add a data dependence edge from 2 to 1 as statement 2 is data dependent on statement 1.



Add yet another node corresponding to statement 3 in trace(t). Also add a data dependence edge from node 3 to node 1 as statement 3 is data dependent on statement 1 and a control edge from node 3 to 2.



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Construction of a DDG: Example [3]

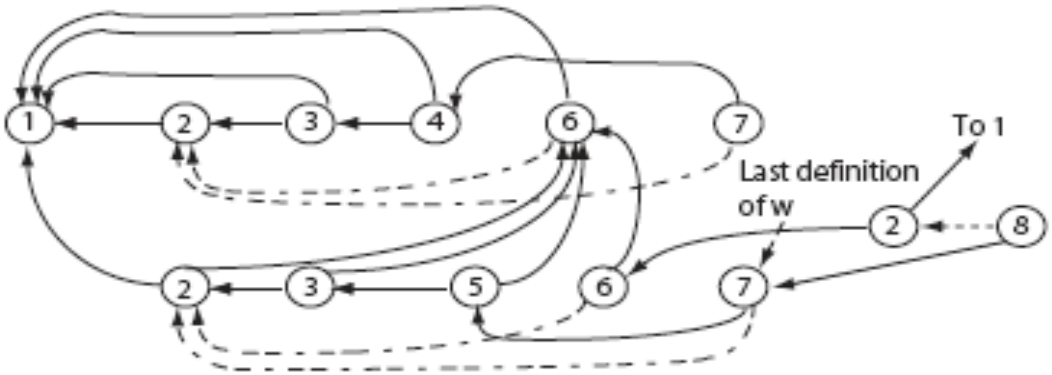
```
1 input (x, y);
2 while (x < y){
3  if (f1(x)==)0
4      z=f2(x);
   else
5      z=f3(x);
6  x=f4(x);
7  w=f5(z);
  }
8 output (w)
  end
```

should be…

3 if(f1(x)==0)

trace(t)={1, 2, 3, 4, 6, 7, 2, 3, 5, 6, 7, 2, 8}

Continuing this way we obtain the following DDG for program P and trace(t).



Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Obtaining dynamic slice (DS)

Step 1: Execute P against test t and obtain trace(t).

Step 2: Construct the dynamic dependence graph G from P and trace(t).

Step 3: Identify in G node n labeled L that contains the last assignment to v. If no such node exists then the dynamic slice is empty, other wise execute Step 4.

Step 4: Find in G the set DS(t, v, n) of all nodes reachable from n, including n. DS(t, v, n) is the dynamic slice of P with respect to v at location L and test t.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Obtaining dynamic slice: Example

Suppose we want to compute the dynamic slice of P with respect to variable w at line 8 and test t shown earlier.

We already have the DDG of P for t.

First identify the last definition of w in the DDG. This occurs at line 7 as marked.

Traverse the DDG backwards from node 7 and collect all nodes reachable from 7. This gives us the following dynamic slice: $\{1, 2, 3, 5, 6, 7, 8\}$.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test selection using dynamic slice

Let T be the test set used to test P. P' is the modified program. Let n1, n2, ..nk be the nodes in the CFG of P modified to obtain P'. Which tests from T should be used to obtain a regression test T' for P'?

Find DS(t) for P. If any of the modified nodes is in DS(t) then add t to T'.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

In class exercise

Suppose line 4 in the example program P shown earlier is modified to obtain P' .

(a) Should t be included in T' ?

(b) Will t be included in T' if we were to use the execution slice instead of the dynamic slice to make our decision?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Teasers [1]

You may have noticed that a DDG could be huge, especially for large programs. How can one reduce the size of the DDG and still obtain the correct DS?

The DS contains all statements in trace(t) that had an effect on w, the variable of interest. However there could be a statement s in trace(t) that did not have an effect but could affect w if changed. How can such statements be identified? [Hint: Read about potential dependence.]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Teasers [2]

Suppose statement s in P is deleted to obtain P' ? How would you find the tests that should be included in the regression test suite?

Suppose statement s is added to P to obtain P' ? How would you find the tests that should be included in the regression test suite?

In our example we used variable w to compute the dynamic slice. While selecting regression tests, how would you select the variable for which to obtain the dynamic slice?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

9.8 Test selection using test minimization

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test minimization [1]

Test minimization is yet another method for selecting tests for regression testing.

To illustrate test minimization, suppose that P contains two functions, main and f. Now suppose that P is tested using test cases t1 and t2. During testing it was observed that t1 causes the execution of main but not of f and t2 does cause the execution of both main and f.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test minimization [2]

Now suppose that P' is obtained from P by making some modification to f.

Which of the two test cases should be included in the regression test suite?

Obviously there is no need to execute P' against t1 as it does not cause the execution of f. Thus, the regression test suite consists of only t2.

In this example we have used function coverage to minimize a test suite {t1, t2} to a obtain the regression test suite {t2}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test minimization [3]

Test minimization is based on the coverage of testable entities in P.

Testable entities include, for example, program statements, decisions, def-use chains, and mutants.

One uses the following procedure to minimize a test set based on a selected testable entity.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

A procedure for test minimization

Step 1: Identify the type of testable entity to be used for test minimization. Let e1, e2, ..ek be the k testable entities of type TE present in P. In our previous example TE is function.

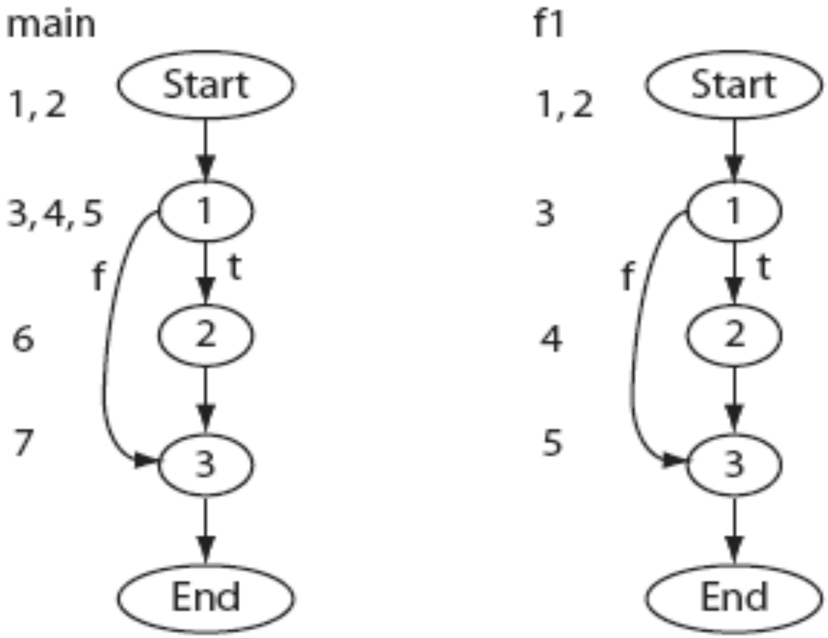
Step 2: Execute P against all elements of test set T and for each test t in T determine which of the k testable entities is covered.

Step 3: Find a minimal subset T' of T such that each testable entity is covered by at least one test in T'.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test minimization: Example

Step 1: Let the basic block be the testable entity of interest. The basic blocks for a sample program are shown here for both main and function f1.



Step 2: Suppose the coverage of the basic blocks when executed against three tests is as follows:

t1: main: 1, 2, 3. f1: 1, 3

t2: main: 1, 3. f1: 1, 3

t1: main: 1, 3. f1: 1, 2, 3

Step3: A minimal test set for regression testing is {t1, t3}.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Test minimization: Teasers

Is the minimal test set unique? Why or why not?

Is test minimization NP hard? How is the traditional set cover problem in mathematics related to the test minimization problem?

What criteria should be used to decide the kind of testable entity to be used for minimization?

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

9.9 Test selection using test prioritization

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Test prioritization

Note that test minimization will likely discard test cases. There is a small chance that if P' were executed against a discarded test case it would reveal an error in the modification made.

When very high quality software is desired, it might not be wise to discard test cases as in test minimization. In such cases one uses test prioritization.

Tests are prioritized based on some criteria. For example, tests that cover the maximum number of a selected testable entity could be given the highest priority, the one with the next highest coverage m the next higher priority and so on.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

A procedure for test prioritization

Step 1: Identify the type of testable entity to be used for test minimization. Let e_1, e_2, ..e_k be the k testable entities of type TE present in P. In our previous example TE is function.

Step 2: Execute P against all elements of test set T and for each test t in T. For each t in T compute the number of distinct testable entities covered.

Step 3: Arrange the tests in T in the order of their respective coverage. Test with the maximum coverage gets the highest priority and so on.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Using test prioritization

Once the tests are prioritized one has the option of using all tests for regression testing or a subset. The choice is guided by several factors such as the resources available for regression testing and the desired product quality.

In any case test are discarded only after careful consideration that does not depend only on the coverage criteria used.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

9.10. Tools

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Tools for regression testing

Methods for test selection described here require the use of an automated tool for all but trivial programs.

xSuds from Telcordia Technologies can be used for C programs to minimize and prioritize tests.

Many commercial tools for regression testing simply run the tests automatically; they do not use any of the algorithms described here for test selection. Instead they rely on the tester for test selection. Such tool are especially useful when all tests are to be rerun.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary [1]

Regression testing is an essential phase of software product development.

In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.

In such situations one can make use of sophisticated technique for selecting a subset of all tests and hence reduce the time for regression testing.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Summary [2]

Test selection for regression testing can be done using any of the following methods:

Select only the modification traversing tests [based on CFGs].

Select tests using execution slices [based on execution traces].

Select tests using dynamic slices [based on execution traces and dynamic slices].

Select tests using code coverage [based on the coverage of testable entities].

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Summary [3]

Select tests using a combination of code coverage and human judgment [based on amount of the coverage of testable entities].

Use of any of the techniques mentioned here requires access to sophisticated tools. Most commercially available tools are best in situations where test selection is done manually and do not use the techniques described in this chapter.

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Contents

Chapter 10

Unit Testing

[Under Construction]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd

Chapter 11

Integration Testing

[Under Construction]

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd