

Testarea Sistemelor Software

Prof. univ. dr. Tudor Bălănescu

Specificare

- Specificare: descrierea detaliilor structurale și de comportament ale unui produs.
 - Sistemele mari de programe sunt descompuse în *subsisteme*.
 - Pentru ca subsistemele să poată fi *dezvoltate independent*, este necesară definirea *interfețelor de comunicare* între subsisteme.
 - Exemplu: specificarea printr-un set de clase de obiecte, stabilite de comun acord cu toți dezvoltatorii de subsisteme

Formalism vs. limbaj natural

- Specificare neformalizată (mai mare grad de ambiguitate, incompletitudine și inconsistență)
- Specificare formalizată: (neambiguă și mai completă)
 - Completează (însoțește) specificarea neformalizată
 - Facilitează analiza proprietăților de consistență și de neambiguitate ale specificării neformalizate
- Tipuri de specificări formale:
 - Algebrică
 - Logică
 - Fundamentată pe teoria mulțimilor
 - Vienna Development Method
 - Notația (limbajul) Z
 - etc.

Verificare și Validare

- Procesul de detectare și de eliminare a defectiunilor din programe are două componente: *verificare și validare (V & V)*.
- V&V, parte a aceluiași proces prin care se urmărește:
 - 1) descoperirea defectelor programului;
 - 2) certificarea faptului că programul va funcționa corect în condiții reale de exploatare.
- Termenii verificare și validare sunt utilizati deseori cu semnificații ambigue și sunt confundați.
- Diferența dintre cei doi termeni poate fi exprimată succint astfel: *prin verificare se urmărește dacă programul a fost construit bine; prin validare se controlează dacă programul este bun.*
- Prin verificare se atestă compatibilitatea programului cu specificația inițială.
- Prin validare se controlează dacă programul îtrunește calitățile cerute de utilizator. Cele două aspecte nu coincid întotdeauna.

Exemplu, V&V

- Să presupunem că se dorește realizarea unui program după executarea căruia valorile vectorului v să fie ordonate crescător (la terminare, valorile lui v trebuie să aibă proprietatea: " $v[i] \leq v[i+1]$ dacă $1 \leq i \leq n-1$ "
- Posibilă soluție:
for $i := 1$ **to** n **do** $v[i] := 0$.
- Acesta va rezista etapelor de verificare deoarece îndeplinește cerințele din specificația inițială, în care s-a omis să se precizeze că valorile componentelor lui v nu pot fi modificate decât prin operații de interschimbare.
- Totuși el nu poate fi validat deoarece cerința utilizatorului a fost cu totul alta.

Comportamentul instabil al Sistemelor software

- V&V nu sunt specifice [GheJaMa 91] activității de realizare a produselor software, ci sunt întâlnite în toate procesele tehnologice, unde se controlează atât desfășurarea corectă a procesului de fabricație (verificare) cât și calitățile produsului finit (validare).
Exemplu: construcția unui pod, testarea cu camioane
- procesul de realizare a unei construcții fizice are o anumită proprietate de *stabilitate (continuitate în comportament)*, care lipsește sistemelor software
Exemplu: **while** $x <> 0$ **do** $x := x - 2$

V&V, dinamic, static

- Tehnicile de verificare și validare pot fi dinamice sau statice.
- Primele constă în executarea programului în condiții asemănătoare cu cele din exploatarea reală pentru a vedea dacă intrunește calitățile așteptate. Acest experiment este cunoscut sub numele de testare și se bazează pe executarea efectivă a programului.
- Celelalte tehnici, de natură statică, implică examinarea și analiza textului programului, a documentației, a metodelor de proiectare etc. spre a deduce funcționarea corectă a produsului final. Executarea programului nu este necesară, ea este cel mult imaginată. În această categorie este inclusă:
 - inspectarea codului sursă pentru depistarea eventualelor anomalii precum și
 - metodele formalizate de verificare a programelor.
- Cele două categorii de tehnici se dovedesc a fi complementare și este recomandat să fie aplicate împreună.

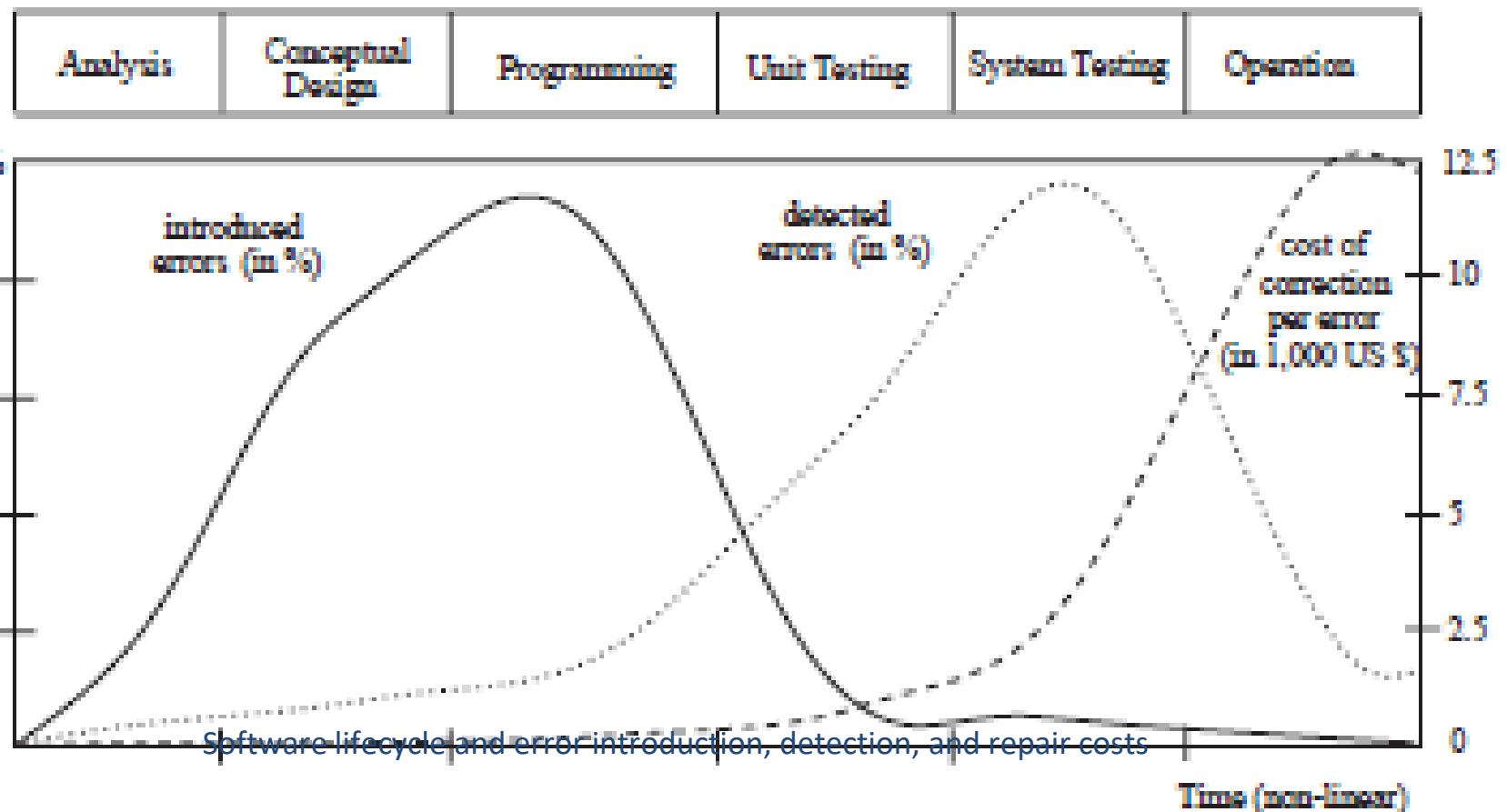
Correctness vs. reliability (corectitudine vs. siguranță în exploatare)

- În [DaHoDij 72], Dijkstra: tehniciile dinamice pot fi utilizate doar pentru a **semnală** prezența defectelor unui program, fără a garanta absența lor; erori încă nedepistate pot exista chiar după teste foarte cuprinzătoare.
- Testarea programelor, care se bazează pe executarea în situații considerate reprezentative pentru operarea reală, este prin natura sa **incompletă** deoarece în general nu pot fi experimentate toate cazurile posibile ale funcționării reale.
- Din acest motiv, se consideră că un test este **concludent** numai dacă prin el este semnalată o eroare. În caz contrar, testul este considerat pur și simplu nerelevant și presupus că nu a solicitat programul într-o măsură suficientă pentru a-i expune defectele.

Testare vs. Demonstrare

- Pe de altă parte, procesul de examinare și analiză, oricât de formalizat ar fi, este el însuși predispus la erori, ca orice activitate umană;
- se limitează la corespondența dintre program și specificația sa, fără a garanta că produsul este operațional.
- Aplicarea metodelor matematice de verificare este însă condiționată de existența unei definiții precise a semanticii limbajului de programare în care este scris algoritmul și de specificarea cerințelor programului într-o notație adecvată metodei de verificare.
- Există însă multe limbaje de utilizare largă ce nu dispun de definiții semantice riguroase sau ale căror definiții semantice săn exprimate prin mecanisme greu de înțeles de către utilizatori. În consecință, pentru multe programe este greu de realizat o argumentare matematică a funcționării lor, testarea rămînînd tehnica predominantă a procesului de verificare și validare.
- Acest echilibru între formal și neformal poate fi realizat numai pe baza unei bune familiarizări atât cu **tehnicele de testare** cât și cu **metodele matematice de verificare**.

Depistare erori: cu cat mai devreme, cu atat mai bine



Tehnici de testare a programelor

- Prin testare se înțelege de fapt executarea programului cu intenția declarată de a descoperi cel puțin o eroare (aspectul *demolator*)
- Ea se bazează pe imaginarea unor *eșantioane* de date de intrare care să conducă la depistarea erorilor într-un timp cât mai scurt și cu efort cât mai mic.
- În cele ce urmează, prin *test* se înțelege chiar această *mulțime finită de eșantioane*.
- Test case:

Obiectivele testării

- Să indice prezența erorilor
- Să faciliteze localizarea erorilor (în vederea depanării)
- Să fie repetabil: la repetarea experimentului, să se obțină aceleași rezultate
 - Nerepetabil, din cauza contextului (environment)
 - Nerepetabil, din cauza nedeterminismului (paralelismului)

Testare și absența proprietății de “continuitate”

- Testul (set of test cases) exercează un număr finit de comportamente: fiecare corespunde unui eșantion de date (test case)
- Comportamentul în cazul testelor nu poate fi “extrapolat” către o concluzie privind comportamentul, în general, al sistemului
- Nu există continuitate comportamentală: poate funcționa bine într-o infinitate de cazuri, dar poate să funcționeze defectuos în altele

- **Observație.** Numărul eșantioanelor dintr-un test influențează relevanța testului, dar nu este un element determinant. Esențială este ***calitatea*** datelor alese;
- Exemplu: de pildă, în cazul instrucțiunilor

if $x > y$ **then**

$m := x$

else

$m := x;$

{cea mai mare valoare dintre x și y este m }

- testul cu 3 eșantioane
 $x = 2, y = 1; x = 1, y = 0; x = 1, y = 1$
este neconcludent
- testul cu un eșantion
 $x = 0, y = 1$
este concludent.

Test Process Maturity Level in an Organisation

(Moa Johansson, Wolfgang Ahrendt, Vladimir Klebanov: Testing, Debugging, and Verification)

- Level 0: There is no difference between testing and debugging.
- Level 1: Purpose of testing: show correctness.
- Level 2: Purpose of testing: show that the software does not work.
- Level 3: Purpose of testing: reduce the risk of using the software.
- Level 4: Testing is a mental discipline helping IT professionals to develop higher quality software.

Level 0: There is no difference between testing and debugging.

Testing is the same as debugging

- Does not distinguish between incorrect behaviour and defects in the program
- Does not help develop software that is reliable or safe

Level 1: Purpose of testing: show correctness.

Purpose: showing correctness

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

Level 2: Purpose of testing: show that the software does not work.

Purpose: showing failures

- Looking for failures is a negative activity
- Puts testers and developers into an adversarial relationship
- What if there are no failures?

This describes most software companies.

Level 3: Purpose of testing: reduce the risk of using the software.

Purpose: reduce risk

- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and the consequences catastrophic
- *Testers and developers work together to reduce risk*

This describes a few “enlightened” software companies.

Level 4: Purpose of testing: improve ability of developers to produce high quality software

Testing, a mental discipline that increases quality

- Testing only one way to increase quality
- *Test engineers can become technical leaders of the project*
- Primary responsibility to measure and improve software quality
- Their expertise should help developers

This is the way “traditional” engineering works.

Activities of Test Engineer

Test engineer: IT professional in charge of test activities, including:

- designing test inputs
- running tests
- analysing results
- reporting results to developers and managers
- *automating any of the above*

Developer = or /= Test Engineer?

Should testing be done by the developers (of the same software)?

Different takes on this:

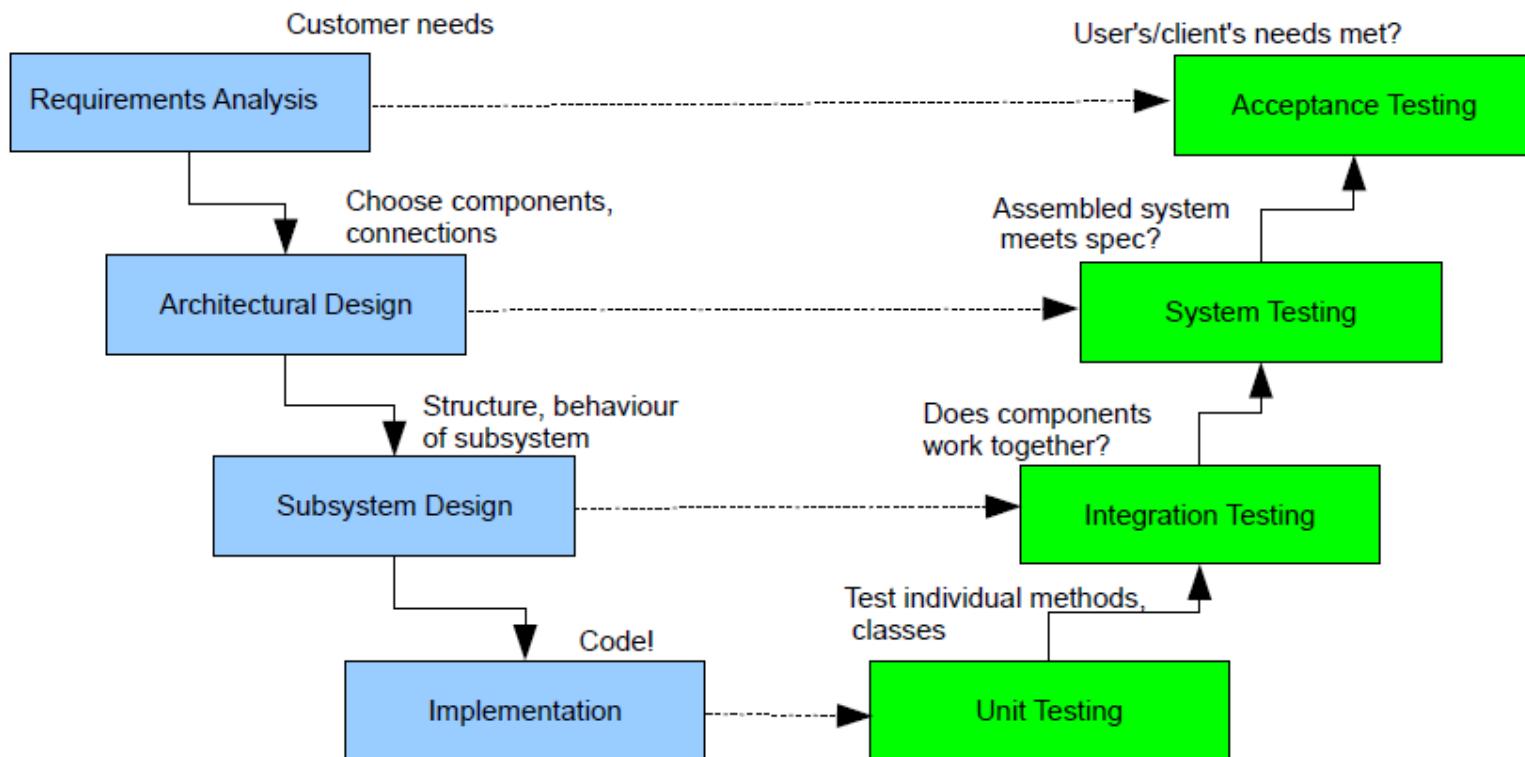
- Contra: Test Principles in [Myers]
 - Principle 2: Programmer should avoid testing his/her own program.
(misunderstanding of specs carry over to testing)
 - Principle 3: A programming organization should not test its own programs.
- Pro: Extreme Testing (ET)
 - Principle: Developers create tests first
 - Principle: Developers re-run test on all incremental changes

Tool: JUnit designed for ET

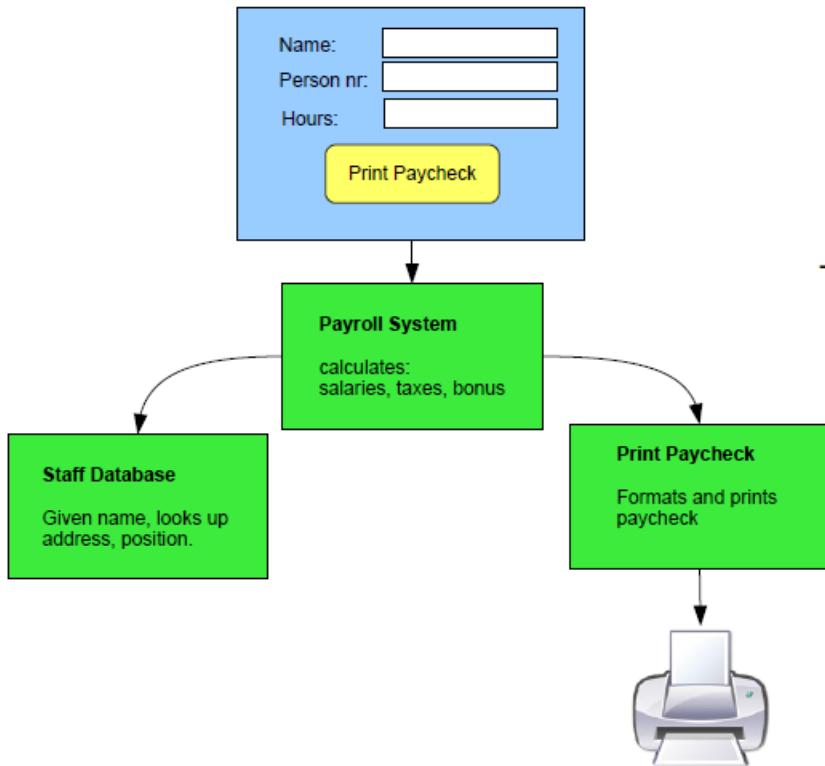
Testing Levels Based on Software Activity

- Acceptance Testing: assess software with respect to user requirements
- System Testing: assess software with respect to system-level specification.
 - Testing system against specification of externally observable behavior
- Integration Testing: assess software with respect to high-level design;
 - testing interaction between modules
- Unit Testing: assess software with respect to low-level unit design; testing individual units of a system
 - traditionally: unit = procedure
 - in object-orientation (Java): unit = method

Life cycle & Testing level Model



Example: System Testing, Integration Testing



- System Test
 - Enter data in GUI, does it print the correct paycheck, formatted as expected *by specification?*
 - (Acceptance Test: does ... *by user?*)
- Integration Tests, e.g.
 - Payroll asks database for staff data, are values what's expected?
 - Maybe there are special characters (unexpected!).
 - Are paychecks formatted correctly for different kinds of printers?
- Unit Tests, e.g.
 - Does payroll system compute correct tax-rate, bonus etc?
 - Does the Print Paycheck button react when clicked?

Regression Testing

(Orthogonal to the above testing levels)

- Regression Testing: Testing that is done after changes in the software.
- Purpose:
- gain confidence that the change(s) did not cause (new) failures.

It is a standard part of the *maintenance phase* of software development.

- E.g. Suppose Payroll subsystem is updated. Need to re-run tests (which ones?).

Theoretical foundations

(Carlo Ghezzi)

- Definitions
- P (program), D (input domain), R (output domain)
 - $P: D \rightarrow R$ (may be partial)
- Correctness defined by a test oracle $OR \subseteq D \times R$
 - $P(d)$ correct (w.r.t OR) if $\langle d, P(d) \rangle \in OR$
 - P correct (w.r.t OR) if all $P(d)$ are correct
- Test case t
 - an element of D
- Test set T
 - a **finite** subset of D, and
 - *an oracle OR* (vital for automated evaluation of test)
- Test t is successful if $P(t)$ is correct ($\langle t, P(t) \rangle \in OR$)
- Test set T successful if P correct for all t in T

Ideal Test

- *Ideal test set T*: if P is incorrect, there is an element of T such that $P(d)$ is incorrect
 - *if an ideal test set exists for any program, we could prove program correctness by testing*
-

$$P \text{ correct (w.r.t OR)} \Leftrightarrow (\text{by def}) \forall d \in D \langle d, P(d) \rangle \in OR$$

$$T \text{ successful (on } P) \Leftrightarrow (\text{by def}) \forall t \in T \langle t, P(t) \rangle \in OR$$

$$P \text{ incorrect (w.r.t OR)} \Leftrightarrow (\text{by def}) \exists d \in D \langle d, P(d) \rangle \notin OR$$

↑ ↓(only for ideal T)

$$T \text{ fails(on } P) \Leftrightarrow (\text{by def}) \exists t \in T \langle t, P(t) \rangle \notin OR$$

Hence, for ideal T : T fails(on P) \Leftrightarrow P incorrect (w.r.t . OR). i.e

T successful (on P) \Leftrightarrow P correct (w.r.t . OR).

Note: Finding ideal tests is **undecidable**

Test criterion

- A criterion C defines finite subsets of D (test sets)
 - $C \subseteq 2^D$
- A test set T satisfies C if it is an element of C

Example

$$C = \{ \langle x_1, x_2, \dots, x_n \rangle \mid n \geq 3 \wedge \exists i, j, k, (x_i < 0 \wedge x_j = 0 \wedge x_k > 0) \}$$

$\langle -5, 0, 22 \rangle$ is a test set that satisfies C

$\langle -10, 2, 8, 33, 0, -19 \rangle$ also does

$\langle 1, 3, 99 \rangle$ does not

Properties of criteria (1)

- C is consistent
 - for any pairs T₁, T₂ satisfying C, T₁ is successful iff T₂ is successful
 - so either of them provides the “same” information
- C is complete
 - if P is incorrect, there is a test set T of C that is not successful
- C is complete and consistent
 - identifies an ideal test set
 - allows correctness to be proved!

Properties of criteria (2)

- C1 is finer than C2
 - for any program P
 - for any T1 satisfying C1 there is a subset T2 of T1 which satisfies C2

Properties of definitions

- None is effective, i.e., no algorithms exist to state if a program, test set, or criterion has that property
- In particular, there is no algorithm to derive a test set that would prove program correctness
 - there is no constructive criterion that is consistent and complete

Empirical testing principles

- Attempted compromise between the impossible and the inadequate
- Find strategy to select significant test cases
 - significant=has high potential of uncovering presence of error

Complete-Coverage Principle

- Try to group elements of D into subdomains D_1, D_2, \dots, D_n where any element of each D_i is likely to have similar behavior
 - $D = D_1 \cup D_2 \cup \dots \cup D_n$
- Select one test as a representative of the subdomain
- If $D_j \cap D_k = \emptyset$ for all j, k (partition), any element can be chosen from each subdomain
- Otherwise choose representatives to minimize number of tests, yet fulfilling the principle

Tehnici de testare a programelor

- Există o mare varietate de metode de elaborare a unor astfel de eșantioane, care pun la dispoziția programatorului căi de abordare sistematică a activității de testare, ce asigură o probabilitate ridicată de descoperire a erorilor. Aceste metode se bazează mai mult sau mai puțin pe una din următoarele strategii:
 - 1) *testare funcțională* sau *metoda cutiei negre (black box)* : cunoscînd funcțiile pe care trebuie să le îndeplinească programul, eșantioanele sănt astfel concepute încît să se asigure că fiecare funcție este pe deplin realizată;
 - 2) *testare structurală* sau *metoda cutiei transparente (white box)* : cunoscînd structura (instrucțiunile) algoritmului, eșantioanele sunt astfel concepute încît să se asigure că sunt testate convingător toate părțile programului.

Metoda cutiei negre (black box)

Testare funcțională (functional testing)

- Testele sănt elaborate pe baza unei specificări funcționale a programului.
- Programul este văzut ca o cutie neagră a cărei comportare este determinată prin *prelucrarea* unor date de intrare și *observarea* rezultatelor obținute.
- Compararea rezultatelor observate cu cele așteptate este făcută pe baza unui asa numit *oracol*. (un simplu tabel, un alt program care raspunde cu *yes* sau *no*, un sistem cu interfață grafică, roșu pentru testele concludente etc.)
- Elaborarea testelor depinde în mare măsură de îndemînarea și experiența celui care face testarea, dar există și idei cu aplicabilitate mai generală care pot să ghidzeze acțiunea de selectare a eșantioanelor de test după o strategie riguroasă.
- În anumite cazuri, procesul de elaborare a testelor poate fi *automatizat*.

Example: The Triangle Problem

The aim of this program is to classify triangles. The program accepts three positive integers as lengths of the sides of a triangle. The program classifies the triangle into one of the following groups:

- *Equilateral*: all the sides have equal lengths (return 1)
- *Isosceles*: two sides have equal length, but not all three (return 2)
- *Scalene*: all the lengths are unequal (return 3)
- *Impossible*: the three lengths cannot be used to form a triangle, or form only a flat line (return 4)

Adapted from

(appears in Myers' book) [Testing docs 2015\Software Testing.ppt](#)

- [Testing docs 2015\Testing1.pdf](#) ((Moa Johansson, Wolfgang Ahrendt, Vladimir Klebanov: Testing, Debugging, and Verification))

Clase de echivalență a datelor de intrare

- domeniul datelor de intrare va fi *partiționat în clase de echivalență*, astfel determinate încît pentru datele din aceeași clasă programul testat să se comporte la fel. Dacă această condiție este îndeplinită, atunci testul constă în executarea programului utilizând un set de date ce conține *cîte un eşantion din fiecare clasă*.
- Exemplu. Căutare binară.
 - Specificare funcțională:
 - $\text{binar}(x, v, n)$
 - intrare:
 - vector $v = (v[1], \dots, v[n])$ de numere întregi, ordonat crescător,
 - n este cel puțin egal cu 2 și inferior unei valori fixe d_{\max} .
 - x , valoarea căutată
 - ieșire: $\text{binar}(x, v, n) = \text{true}$ dacă $x \in v$, $\text{binar}(x, v, n) = \text{false}$ dacă $x \notin v$.

- Considerăm, la început, două clase de echivalentă definite prin:
 - binar $(x, v, n) = \text{true}$
 - binar $(x, v, n) = \text{false}$.
- În această situație, se poate utiliza un test alcătuit din două eșantioane:

$$1) \quad x = 10;$$

$$v = (10, 20, 30, 40);$$

$$n = 4;$$

$$2) \quad x = 0;$$

$$v = (10, 20, 20, 40);$$

$$n = 4.$$

- Testul anterior, foarte probabil neconcludent; continuă partitionarea De pildă în cazul 1 putem distinge situația în care x este primul element al vectorului de situațiile în care este ultimul sau este în interiorul vectorului.
- Se ajunge astfel la o partiție cu 4 clase de echivalentă:
 - 1.a) $\{(x, v, n) \mid \text{binar}(x, v, n) = \text{true} \text{ și } v[n] = x\}$
 - 1.b) $\{(x, v, n) \mid \text{binar}(x, v, n) = \text{true} \text{ și } v[1] = x\}$
 - 1.c) $\{(x, v, n) \mid \text{binar}(x, v, n) = \text{true} \text{ și } v[n] \neq x \neq v[1]\}$
 - 2) $\{(x, v, n) \mid \text{binar}(x, v, n) = \text{false}\}.$
- Continuare rafinare, n par sau impar? Partiție cu 8 clase de echivalentă: 1.a.p; 1.a.i; 1.b.p; 1.b.i; 1.c.p; 1.c.i; 2.p; 2.i (prin p s-a notat faptul că n este număr par).

1.a.p.1 (*)

$x = 20;$

$v = (10, 20);$

$n = 2;$

1.a.i (*)

$x = 30;$

$v = (10, 20, 30);$

$n = 3;$

1.b.p.1

$x = 10;$

$v = (10, 20);$

$n = 2;$

1.b.i

$x = 10;$

$v = (10, 20, 30);$

$n = 3;$

1.a.p.2 (*)

$x = 40;$

$v = (10, 20, 30, 40)$

$n = 4;$

1.b.p.2

$x = 10;$

$v = (10, 20, 30, 40);$

$n = 4;$

1.c.p (*)

x = 20;

v = (10, 20, 30, 40);

n = 4;

1.c.i

x = 20;

v = (10, 20, 30);

n = 3;

2.p.1

x = 0;

v = (10, 20);

n = 2;

2.i

x = 0;

v = (10, 20, 30);

n = 3.

2.p.2

x = 0;

v = (10, 20, 30, 40);

n = 4;

- În situația ideală ar trebui ca rezultatele testelor să fie aceleași, indiferent de eșantionul ales dintr-o clasă de echivalentă. Prin această calitate se legitimează faptul că eșantionul este reprezentativ pentru clasa din care face parte. În realitate, criteriile de definire a claselor de echivalentă și finețea partitiei obținute influențează reprezentativitatea eșantioanelor. Obținerea unei partitii corespunzătoare este un proces euristic care se desfășoară gradual, dirijat și de rezultatele testelor anterioare.

Exercițiu

Exercițiu. Programul următor adaugă și extrage numere întregi dintr-o coadă cu cel mult n_{max} elemente, unde $0 \leq n_{max} \leq 100$. Pentru adăugare se răspunde prin caracterul **a**, pentru extragere prin **e** și pentru oprire prin **o**. Să se testeze prin metoda cutiei negre și să se depaneze acest program.

R. Testul constă în realizarea unor operații de adăugare (**a**) și extragere (**e**) în diverse combinații, urmate de operația (**o**). Un test poate fi reprezentat prin expresia $n_1a\ m_1e, \dots, n_k a\ m_k e$, o dacă s-au făcut n_1 adăugări urmate de m_1 extrageri etc. Se pot distinge cazurile:

- a) $n_{max} = 0$;
- b) $1 \leq n_{max} \leq 100$.

Această partitie poate fi rafinată considerînd situațiile:

- 1) $n_1 + \dots + n_k < n_{max}$;
- 2) $n_1 + \dots + n_k = n_{max}$;
- 3) $n_1 + \dots + n_k > n_{max}$.

Se obțin 5 clase de echivalentă: **a.2, a.3, b.1, b.2 și b.3**, deoarece **a.1** conduce la $n_1 + \dots + n_k < 0$.

- La fiecare din cele 5 cazuri putem distinge situațiile:
 - a) extragerile solicitate sunt totdeauna posibile, adică pentru orice i , $1 \leq i \leq k$ avem $n_1 + \dots + n_i \geq m_1 + \dots + m_i$;
 - b) la un moment dat se solicită o extragere în situația cînd nu mai există elemente în coadă, adică există i astfel încît $n_1 + \dots + n_i < m_1 + \dots + m_i$. Se obțin clasele de echivalență $a.2.a$, $a.2.b$ etc.

- considerarea valorilor *netipice* ale domeniului datelor de intrare, numite deseori valori "de la marginea" domeniului. În cazul analizat este de presupus că algoritmul ar putea să nu funcționeze corect atunci când $n = 2$, adică pe vectori de lungime minimă. Distingând acest caz, clasele de echivalentă $1.a.p$, $1.b.p$ și $2.p$ sunt împărțite în două, obținând o partitie cu 11 clase: $1.a.p.1$; $1.a.p.2$; $1.a.i$; $1.b.p.1$ etc. Alegând cîte un eşantion din fiecare clasă, se alcătuieşte următorul test:

Exercițiu

- Se consideră o funcție care verifică dacă sirurile x și y sunt identice. Să se elaboreze un test utilizând clase de echivalență.
- R. Criteriile de organizare a claselor de echivalență pot fi: lungimi egale sau diferite; siruri identice sau diferite; lungime minimă (1) sau oarecare; siruri distințe care sfîrșesc cu același caracter. Rezultă 6 clase de echivalență și se poate considera următorul test:
1. x = 'a'; y = 'a';
 2. x = 'abc'; y = 'abc';
 3. x = 'a'; y = 'b';
 4. x = 'ab'; y = 'ac';
 5. x = 'ab'; y = 'cb'; *
 6. x = 'ab'; y = 'abc'.

Ghid pentru partitionarea datelor de intrare

(2.3.5, [Testing docs 2015\Mathur Foundations of Software Testing.pdf](#))

- Clase de echivalență asociate variabilelor

Guidelines for generating equivalence classes for variables:
range and strings

		Example	
Kind	Equivalence classes	Constraint	Class representatives ^a
Range	One class with values inside the range and two with values outside the range	$speed \in [60 \dots 90]$	$\{\{50\}\downarrow, \{75\}\uparrow, \{92\}\downarrow\}$
		$area: float;$ $area \geq 0$ $age: int;$ $0 \leq age \leq 120$ $letter: char;$ $fname: string$	$\{\{-1.0\}\downarrow, \{15.52\}\uparrow\}$ $\{\{-1\}\downarrow, \{56\}\uparrow, \{132\}\downarrow\}$ $\{\{J\}\uparrow, \{3\}\downarrow\}$ $\{\{real\}\uparrow, \{small\}\downarrow\}$
String	At least one		

String At least one containing all legal strings and one containing all illegal strings.
Legality is determined based on constraints on the length and other semantic features of the string

fname: string; $\{\{\epsilon\}\downarrow, \{Sue\}\uparrow,$
 $\{Sue2\}\downarrow, \{Too$
 $Long\ a\ name\}\downarrow\}$

vname: string; $\{\{\epsilon\}\downarrow, \{shape\}\uparrow,$
 $\{address1\}\uparrow\},$
 $\{Long$
 $variable\}\downarrow$

^aSymbols following each equivalence class: \downarrow , Representative from an equivalence class containing illegal inputs; \uparrow , representative from an equivalence class containing legal inputs.

Guidelines for generating equivalence classes for variables: Enumeration and arrays

Example ^a			
Kind	Equivalence classes	Constraint	Class representatives ^b
Enumeration	Each value in a separate class	<i>auto_color</i> ∈ {red, blue, green} <i>up</i> : boolean	{ {red}↑, {blue}↑, {green}↑ } { {true}↑, {false}↑ }
Array	One class containing all legal arrays, one containing only the empty array, and one containing arrays larger than the expected size	<i>Java array</i> : int[] <i>aName</i> = new int[3];	{ { [] }↓, { [-10, 20] }↑, { [-9, 0, 12, 15] }↓ }

Unidimensional versus multidimensional partitioning

- Unidimensional: consider one input variable at a time
 - The number of partitions=number of input variables
 - + Traditionally used in test case selection, due to its simplicity and scalability
- Multidimensional: consider the product of all input variables
 - Produces one partition,
 - -usually with a too large number of classes, difficult to manage manually
 - -many of the classes might be infeasible (does not satisfy the constraints among input variables)
 - + offers an increased variety of tests

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 unidimensional partitioning, we apply the partitioning guidelines

6 classes for multidimensional partitioning a) b)

$$E1: x < 3 \quad E2: 3 \leq x \leq 7 \quad E3: x > 7$$

$$E4: y < 5 \quad E5: 5 \leq y \leq 9 \quad E6: y > 9$$

9 classes for multidimensional partitioning c)

$$E1: x < 3, y < 5 \quad E2: x < 3, 5 \leq y < 9 \quad E3: x < 3, y > 9$$

$$E4: 3 \leq x \leq 7, y < 5 \quad E5: 3 \leq x \leq 7, 5 \leq y \leq 9 \quad E6: 3 \leq x \leq 7, y > 9$$

$$E7: x > 7, y < 5 \quad E8: x > 7, 5 \leq y \leq 9 \quad E9: x > 7, y > 9$$

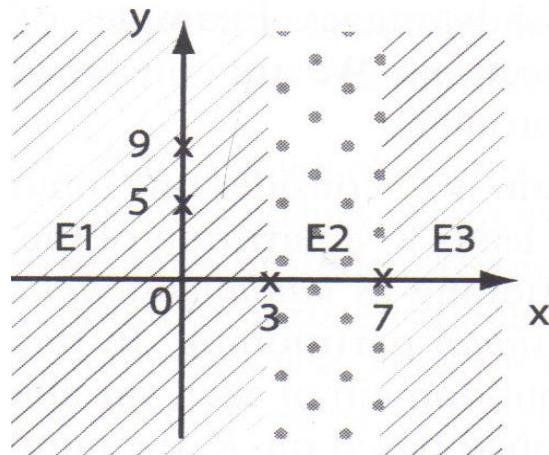
From test selection perspective:

-unidimensional, 6 representative tests, one for each class

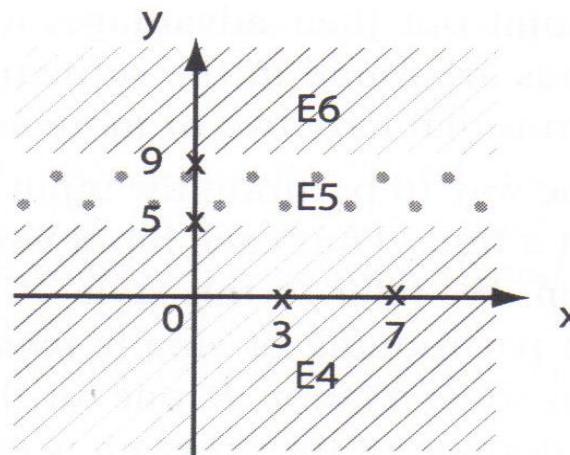
{ ($x=-100, y=0$), ($x=4, y=0$), ($x=100, y=0$), ($x=0, y=0$), ($x=0, y=8$), ($x=0, y=100$) }

-multidimensional, 9 : { ($x=-100, y=0$), ($x=-100, y=6$), ($x=-100, y=100$), ($x=5, y=-100$)...}

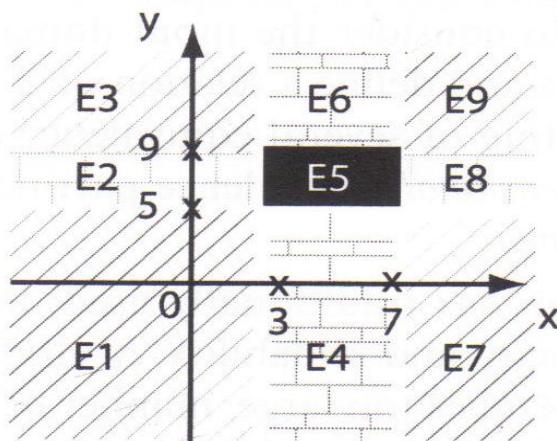
-which is better: depends on application



(a)



(b)



(c)

Geometric representation of equivalence classes derived using unidimensional partitioning based on x and y in (a) and (b), respectively, and using multidimensional partitioning as in (c).

A systematic procedure for equivalence partitioning (1/2)

1. Identify the ***input domain***
 1. Read requirements and identify all ***input and output variables***
(environment variables of operating systems are input variables)
 2. Determine their ***types*** and ***constraints***
 3. Determine the ***set*** each variable can assume
 4. An ***approximation*** of the input domain is the ***product of*** these sets
 5. According to the constraints, add some values from approximation and obtain ***input domain***, as a superset.
2. ***Equivalence classing***: partition the set of values for each variable (unidimensional). Ex. For x and y, $\{X_1, X_2\} \{Y_1, Y_2, Y_3\}$

A systematic procedure for equivalence partitioning (2/2)

3. **Combine** equivalence classes (multidimensional)

$$E = \{X_1 \times Y_1, X_1 \times Y_2, X_1 \times Y_3, X_2 \times Y_1, X_2 \times Y_2, X_2 \times Y_3\}$$

Note: exponential explosion: this step often avoided

4. **Discard infeasible equivalence** classes (that contains one combination of input data that can not be generated during the test, **untestable**):

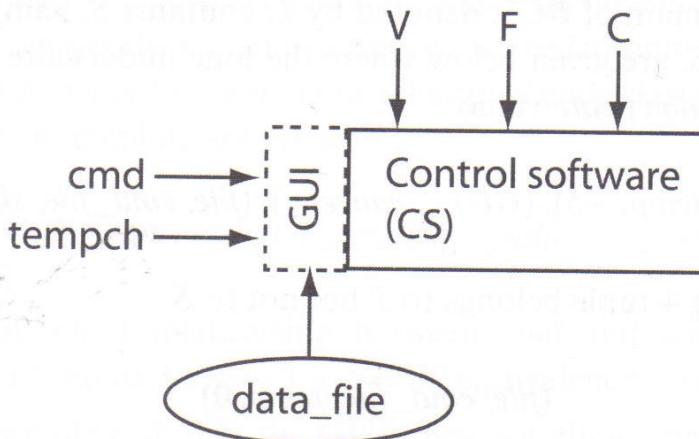
1. This might happen due the filtering of the invalid input combinations by a GUI (choice lists for stations Mers tren etc.)
2. Some classes are wholly infeasible; usually, a class is a mixture of infeasible and testable data.

5. **Test selection.** The resulting partition is then used for *selection of tests*.

Example , Boiler Control System (BCS)

(Adytia P. Mathur)

- One of the selectable Options: C (control), used to give one of 3 commands:
 - temp (set temperature),
selection of temp: BCS ask operator for tempch, values -10,-5, 5, 10
 - shut,
 - cancel
- Selection of C: where from comes the command? GUI? File?
 $V=\{\text{GUI, File}\}$ is another option
- Where is the name of the file? Option F.
- GUI filters tempch (validation)



Inputs for the boiler-control software. V and F are environment variables. Values of cmd (command) and $tempch$ (temperature change) are input via the GUI or a data file depending on V . F specifies the data file.

BCS, step 1

1. Identify input domain: V, F- environment variables

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

Approximation:

$$S = V \times F \times cmd \times tempch.$$

Input domain I: contains S ($S \sqsubseteq I$)
 $(GUI, "x.txt", temp, 5)$ in S (and I)
 $(GUI, "x.txt", temp, 100)$ not in S (but in I)

BCS, step 2

2. Equivalence classing

Variable (name of partition)	Partition
V	$\{\{\text{GUI}\}, \{\text{FILE}\}, \text{undefined}\}$
F	$\{f_valid, f_invalid\}$
cmd	$\{\{\text{TEMP}\}, \{\text{SHUT}\}, \{\text{CANCEL}\}, c_invalid\}$
tempch	$\{\{-10\}, \{-5\}, \{5\}, \{10\}, t_invalid\}\}$

where f_valid = the set of names of the existing files, $f_invalid$, nonexistent
 $c_invalid$, the set of nonexistent commands, stop for instance
 $t_invalid$, the set of not allowed temperature values {0, -1, 10.5 etc.}
 $undefined$ is a singleton indicating that the environment is not defined

Example. Combining (Product of) partitions.

$V \times F = \{\{\text{GUI}\} \times f_valid, \{\text{GUI}\} \times f_invalid, \dots, \text{undefined} \times f_invalid\}$ has 6 classes
and something like
(GUI, “f.txt”), (GUI, “?!!!”), ..., (START, “--!!!”)) are representatives of classes of
 $V \times F$.

BCS, steps 3,4

3. **Combine.** Leads to $3 \times 2 \times 4 \times 5 = 120$ classes !

Like

$\{\text{GUI}\} \times f_{\text{valid}} \times \{\text{TEMP}\} \times t_{\text{invalid}}$, (a class) a.s.o.

Note. Some classes are infinite (f_{valid} is an infinite set of names)

4. **Discard infeasibles.**

-tempch is needed only for command temp.

$V \times F \times \text{cmd} \setminus \{\text{TEMP}\} \times \text{tempch}$ ← 90 infeasible classes, **30 remained**

-GUI filters invalid temperatures

$\{\text{GUI}\} \times f_{\text{valid}} \times \{\text{TEMP}\} \times t_{\text{invalid}}$, ← 2 more Infeasible
 $\{\text{GUI}\} \times f_{\text{invalid}} \times \{\text{TEMP}\} \times t_{\text{invalid}}$,

28 remained

BCS, steps 3,4, cont

- *tempch* not required for *f_invalid*
 $\{\text{FILE}\} \times \{\text{f_invalid}\} \times \{\text{TEMP}\} \times \text{tempch}$  More 5 infeasible
23 left
- *V undefined*, cmd and tempch not allowed
 $\{(undefined, _, \text{temp}, t_valid \cup t_invalid)\}$ More 5 infeasible
18 left a.s.o.
- We discard 102 classes, 18 left:

$\{(GUI, f_valid, \text{temp}, t_valid)\}$	four equivalence classes.
$\{(GUI, f_invalid, \text{temp}, t_valid)\}$	four equivalence classes.
$\{(GUI, _, \text{cancel}, NA)\}$	two equivalence classes.
$\{(file, f_valid, \text{temp}, t_valid \cup t_invalid)\}$	five equivalence classes.
$\{(file, f_valid, shut, NA)\}$	one equivalence class.
$\{(file, f_invalid, NA, NA)\}$	one equivalence class.
$\{(undefined, NA, NA, NA)\}$	one equivalence classes.

NA=data can not be input, GUI does not ask for it

_ = data can be input, but is not used by the control system

5. Tests selection

ID	Equivalence class ^a $\{(V, F, cmd, tempch)\}$	Test data ^b $(V, F, cmd, tempch)$
E1	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, -10)$
E2	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, -5)$
E3	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, 5)$
E4	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, 10)$
E5	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E6	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E7	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E8	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E9	$\{(GUI, _, cancel, NA)\}$	$(GUI, a_file, cancel, -5)$
E10	$\{(GUI, _, cancel, NA)\}$	$(GUI, no_file, cancel, -5)$
E11	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, -10)$
E12	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, -5)$
E13	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, 5)$
E14	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, 10)$
E15	$\{(file, f_valid, temp, t_invalid)\}$	$(file, a_file, temp, -25)$
E16	$\{(file, f_valid, temp, NA)\}$	$(file, a_file, shut, 10)$
E17	$\{(file, f_invalid, NA, ,NA)\}$	$(file, no_file, shut, 10)$
E18	$\{(undefined, _, NA, NA)\}$	$(undefined, no_file, shut, 10)$

^a don't care; NA = input not allowed

^b don't care; NA = input not allowed

Avoiding classes explosion

- *Covering of each equivalence class of each variable by a small number of tests.*
- A test *covers a class E for a variable V* if it *contains a reprezentative of E*
- Thus, one test may cover multiple equivalence classes, one for each input variable.
- Example for BCS: 5 tests cover 14 classes (3+2+4+5, each equivalent classes for each variable).

```
T = { (GUI, a_file, temp, -10), (GUI, no_file, temp, -5),  
      (file, a_file, temp, 5), (file, a_file, cancel, 10),  
      (undefined, a_file, shut, -10)  
 }
```

- Weakness: it fails to consider the semantic relation between different variables
- Example. What happens when *shut* and *V undefined?*

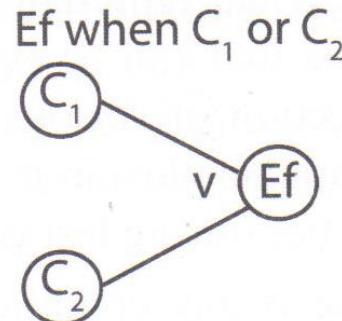
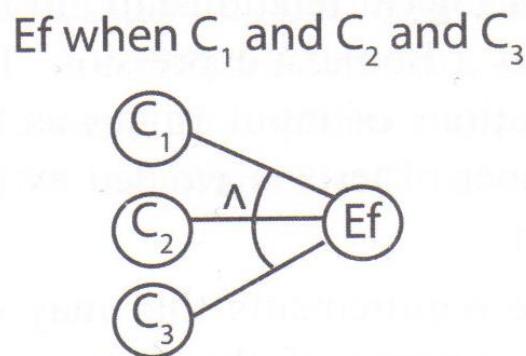
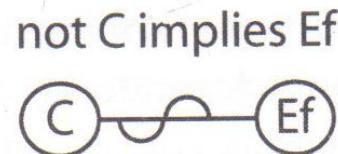
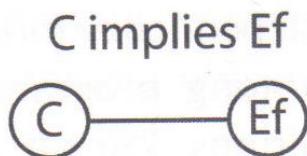
Test generation from requirements: Cause-Effect Graphing

(Mathur, 2.6,[Testing docs 2015\Mathur Foundations of Software Testing.pdf](#))

- Focuses on modeling dependency relationships among program input conditions (*causes*) and output conditions (*effects*), using a *graph*.
- The cause-effect graph allows selection of various combinations of input values as *tests*.
- Steps:
 1. Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
 2. Express the relationship between causes and effects using a cause–effect graph.
 3. Transform the cause–effect graph into a limited entry decision table, hereafter referred to simply as decision table.
 4. Generate tests from the decision table.

Notations used in cause-effect graphing Relations

C, C_1, C_2 , and C_3 denote causes, and Ef denotes an effect.



C implies Ef :

$\text{if}(C) \text{ then } Ef;$

not C implies Ef :

$\text{if } (\neg C) \text{ then } Ef;$

Ef when C_1 and C_2 and C_3 :

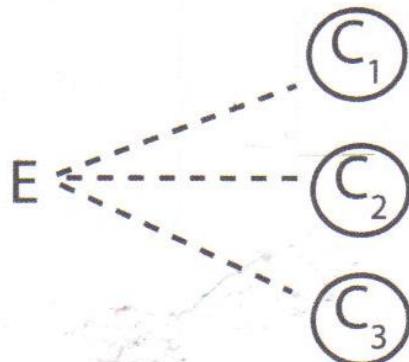
$\text{if}(C_1 \And C_2 \And C_3) \text{ then } Ef;$

Ef when C_1 or C_2 :

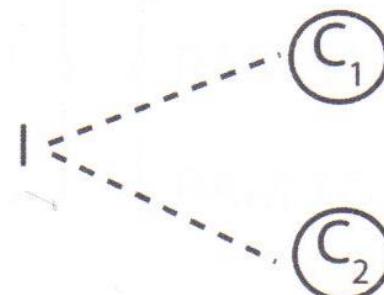
$\text{if}(C_1 \Or C_2) \text{ then } Ef;$

Constraints among causes

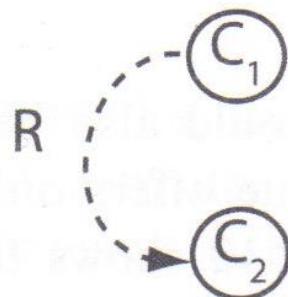
Exclusive: either C_1 or C_2 or C_3



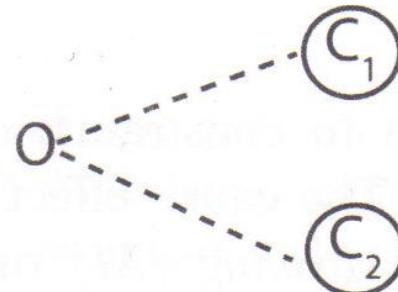
Inclusive: at least C_1 or C_2



C_1 requires C_2



One, and only one, of C_1 and C_2

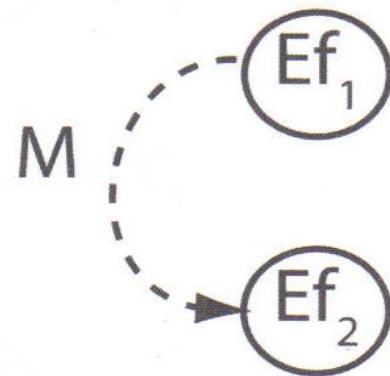


Possible values of constraints

Constraint	Arity	Possible values		
		C_1	C_2	C_3
$E(C_1, C_2, C_3)$	$n > 1$	0	0	0
		1	0	0
		0	1	0
		0	0	1
$I(C_1, C_2)$	$n > 1$	1	0	-
		0	1	-
		1	1	-
$R(C_1, C_2)$	$n > 1$	1	1	-
		1	1	-
		0	0	-
		0	1	-
$O(C_1, C_2, C_3)$	$n > 1$	1	0	0
		0	1	0
		0	0	1

Masking Constraint among Effects

Ef_1 masks Ef_2



Example

Ef_1 : Generate “Shipping invoice”.

Ef_2 : Generate an “Order not shipped” regret letter.

However, Ef_2 is masked by Ef_1 for the same order, that is both effects cannot occur for the same order.

Creating Cause-Effect graphs

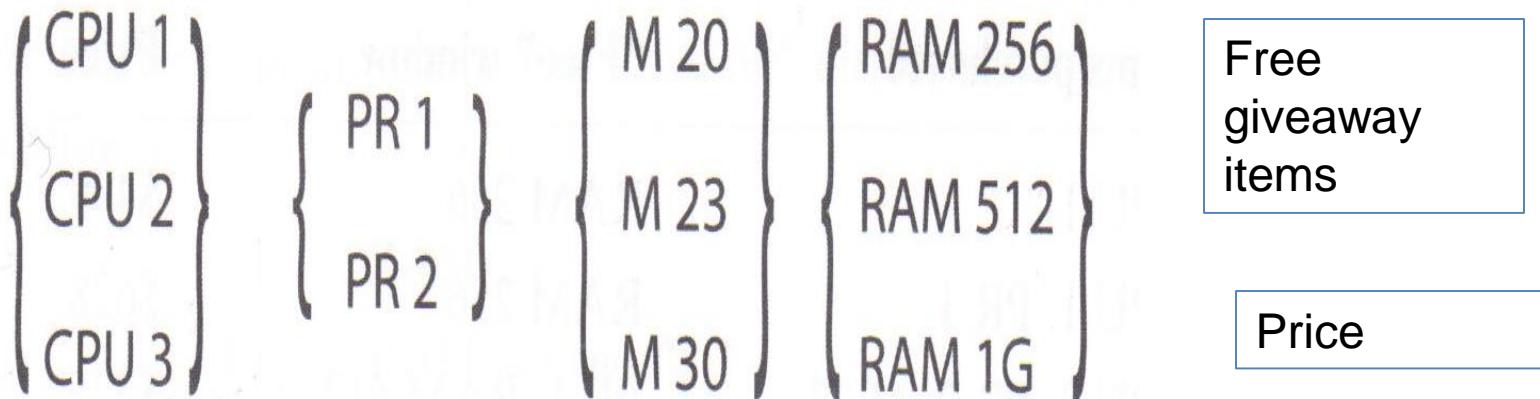
Step 1. Make a list of causes and effects

First, the causes and effects are identified by a careful examination of the requirements. This process also exposes the relationships among various causes and effects as well as constraints among the causes and effects. Each cause and effect is assigned a unique identifier for ease of reference in the cause–effect graph.

Step 2. Causes-Effects graph construction

In the second step, the cause–effect graph is constructed to express the relationships extracted from the requirements. When the number of causes and effects is large, say over 100 causes and 45 effects, it is appropriate to use an incremental approach. An illustrative example follows:

Example. Test generation for a GUI-based computer purchase system



- Four windows for choosing computers, printers, monitors, memory
- One windows for free giveaway items, one for price
- For simplicity: Only one unit of each item can be purchased

Requirements

When a buyer selects a CPU, the contents of the printer and monitor windows are updated. Similarly, if a printer or a monitor is selected, contents of various windows are updated. Any free printer and RAM available with the CPU selection is displayed in a different window marked “Free”. The total price, including taxes, for the items purchased is calculated and displayed in the “Price” window. Selection of a monitor could also change the items displayed in the “Free” window. Sample configurations and contents of the “Free” window are given below.

Items purchased	“Free” window	Price
CPU 1	RAM 256	\$499
CPU 1. PR 1	RAM 256	\$628
CPU 2. PR 2. M 23	PR 1, RAM 512	\$2257
CPU 3. M 30	PR 2, RAM 1G	\$3548

Step 1, List of causes-Effects, Example

Causes

- C 1: Purchase CPU 1.
- C 2: Purchase CPU 2.
- C 3: Purchase CPU 3.
- C 4: Purchase PR 1.
- C 5: Purchase PR 2.
- C 6: Purchase M 20.
- C 7: Purchase M 23.
- C 8: Purchase M 30.

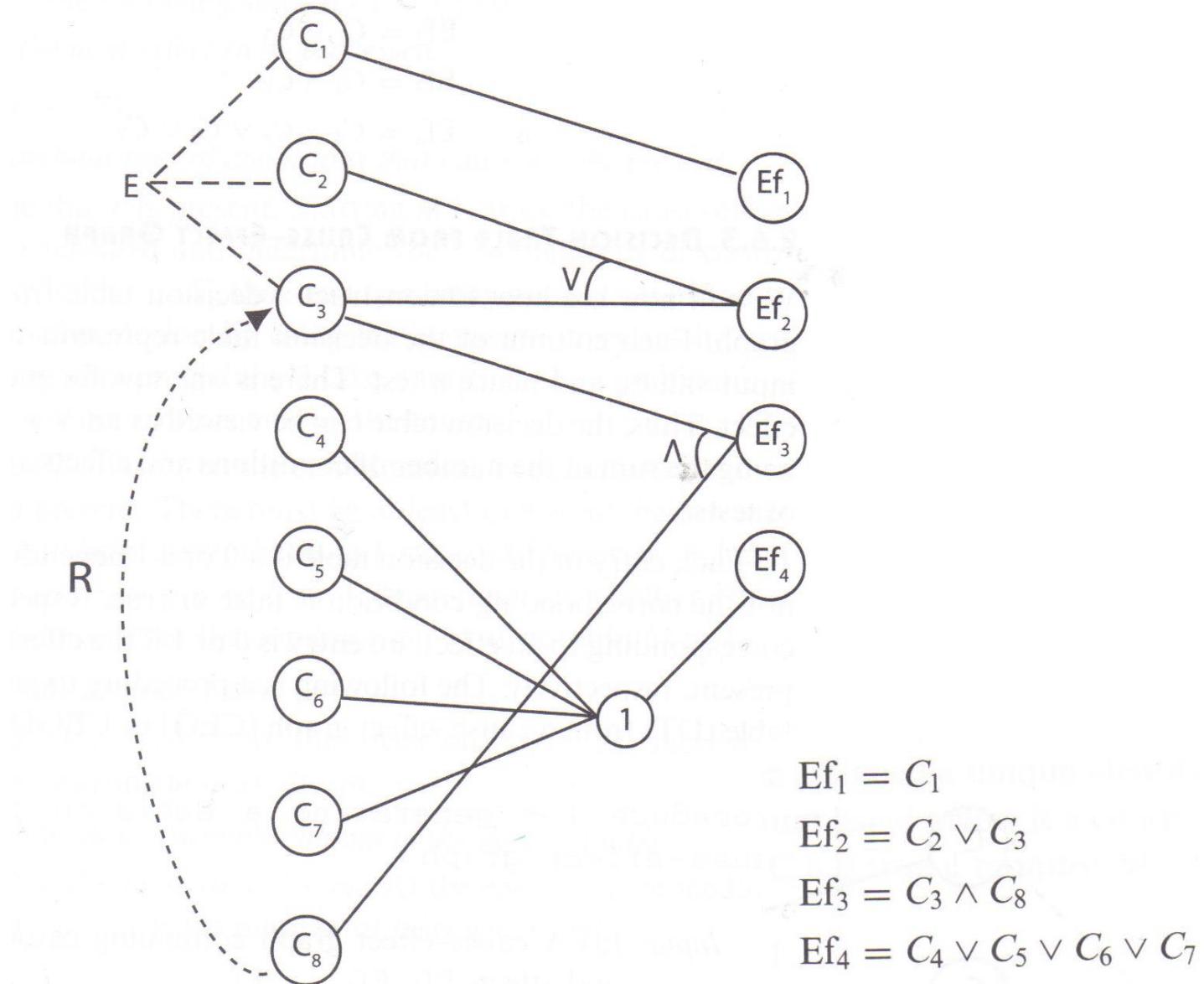
Effects

- Ef₁ : RAM 256.
- Ef₂ : RAM 512 and PR 1.
- Ef₃ : RAM 1G and PR 2.
- Ef₄ : No giveaway with this item.

For simplicity, we ignore
the price-related cause and effect.

Note that while it is possible to order any of the items listed above, the GUI will update the selection available depending on what CPU or any other item is selected. For example, if CPU 3 is selected for purchase, then monitors M 20 and M 23 will not be available in the monitor-selection window. Similarly, if monitor M 30 is selected for purchase, then CPU 1 and CPU 2 will not be available in the CPU window.

Step 2, causes-effects graph, Example



Decision table from cause-effect graph

Procedure for generating a decision table from a cause–effect graph.

Input: (a) A cause–effect graph containing causes C_1, C_2, \dots, C_p and effects Ef_1, Ef_2, \dots, Ef_q .

Output: A decision table DT containing $N = p + q$ rows and M columns, where M depends on the relationship between the causes and effects as captured in the cause–effect graph.

/* i is the index of the next effect to be considered.

$next_dt_col$ is the next empty column in the decision table.

V_k : a vector of size $p + q$ containing 1s and 0s.

V_j , $1 \leq j \leq p$, indicates the state of condition C_j and $V_{l+p}, 1 \leq l \leq p + q$, indicates the presence or absence of effect Ef_{l-p} .

*/

Step 1 Initialize DT to an empty decision table.

next_dt_col = 1.

Step 2 Execute the following steps for $i = 1$ to q .

2.1 Select the next effect to be processed.

Let $e = \text{Ef}_i$.

2.2 Find combinations of conditions that cause e to be present.

Assume that e is present. Starting at e , trace the cause-effect graph backward and determine the combinations of conditions C_1, C_2, \dots, C_p that lead to e being present. Avoid combinatorial explosion by using the heuristics given in the text following this procedure. Make sure that the combinations satisfy any constraints among the causes.

Let V_1, V_2, \dots, V_{m_i} be the combinations of causes that lead to e being present. There must be at least one combination that makes e to be present, that is in 1-state, and hence $m_i \geq 1$. Set $V_k(l)$, $p < l \leq p + q$ to 0 or 1 depending on whether effect Ef_{l-p} is present for the combination of all conditions in V_k .

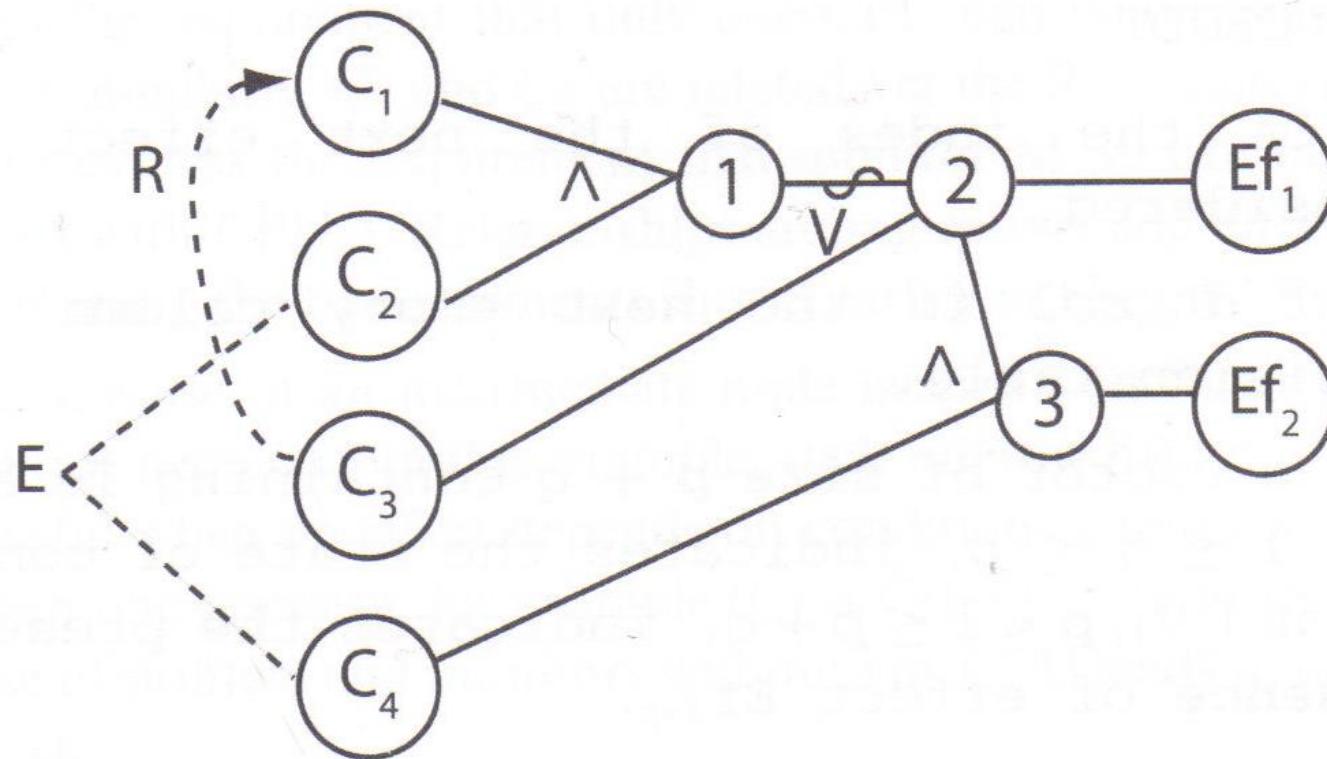
2.3 Update the decision table.

Add V_1, V_2, \dots, V_{m_i} to the decision table as successive columns starting at next_dt_col .

2.4 Update the next available column in the decision table.

$\text{next_dt_col} = \text{next_dt_col} + m_i$. At the end of this procedure, $\text{next_dt_col} - 1$ is the number of tests generated.

Example, a cause-effect graph



The obtained decision table

	1	2	3	4	5	6	7	8
C_1	1	1	1	0	0	1	1	0
C_2	0	1	0	1	0	0	0	0
C_3	1	1	0	0	0	1	0	0
C_4	0	0	0	0	0	1	1	1
Ef_1	1	1	1	1	1	1	1	1
Ef_2	0	0	0	0	0	1	1	1

Heuristics to avoid combinatorial explosion

- Brute force:
 n causes related to an effect: 2^{**n} combination of causes
- Combinatorial explosion can be avoided by using simple heuristics related to OR / AND nodes.
- Heuristics are based on the assumption that certain types of errors are less likely to occur than others.
 - Beneficial: reduction in the number of generated tests
 - Drawbacks: it may also discard tests that would have revealed an error
- Hence, one must apply the heuristics with care and only when the total number of tests is too large to be used in practice
- Bases: Myers , Marick:
“it is unlikely that skipping the ALL-TRUE case in an OR node will miss predicate coding faults”

Myers: Each-Conditions/All-Conditions heuristic

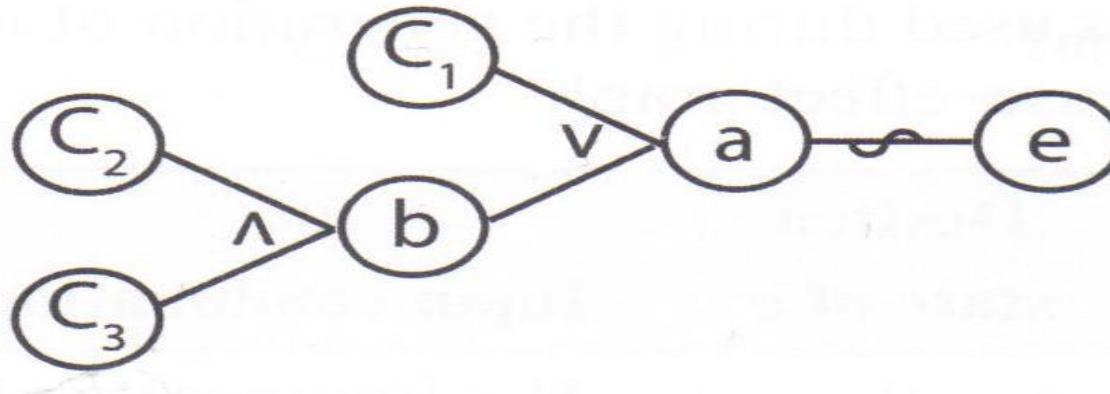
- The *test suite* is composed of *test cases* such that:
 - Each variable is made TRUE ONCE, with all other being FALSE

x	y	z	w
<hr/>			
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

- Add the *test case*
 - ALL TRUE, for AND node
 - x y z w
1 1 1 1
 - ALL FALSE, for OR node
 - 0 0 0 0

- For n conditions: n+1 test cases (instead of 2^{**n})
(good limit but, it is likely that relevant test cases are missing)

Example: Each-Conditions/All-Conditions

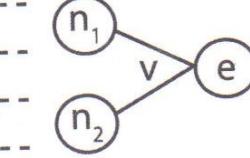
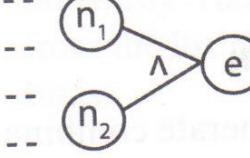


- OR (node a):

C1	b	Hence:	C1	C2	C3
0	1		0	1	1
1	0		0	0	1
0	0		0	1	0
- AND (node b):

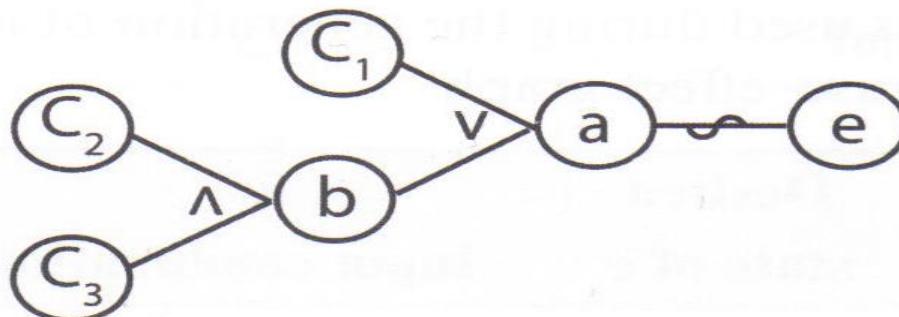
C2	C3		1	0	1
0	1		1	1	0
1	0				
1	1	(C1=1, only with b=0, we discarded: 1 1 1)			

Heuristics oriented to the “desired state”

Node type	Desired state of e	Input combinations
	0	H ₁ : Enumerate all combinations of inputs to n_1 and n_2 such that $n_1 = n_2 = 0$
	1	H ₂ : Enumerate all combinations of inputs to n_1 and n_2 other than those for which $n_1 = n_2 = 1$
	0	H ₃ : Enumerate all combinations of inputs to n_1 and n_2 such that each of the possible combinations of n_1 and n_2 appears exactly once and $n_1 = n_2 = 1$ does not appear. Note that for two nodes, as in the figure on the left, there are three such combinations: (1, 0), (0, 1), and (1, 0). In general, for k nodes added to form e , there are $2^k - 1$ such combinations
	1	H ₄ : Enumerate all combinations of inputs to n_1 and n_2 such that $n_1 = n_2 = 1$

Total number is still large,
but reduced for:
FALSE, in OR case
TRUE, in AND

Example: desired state heuristic



$H_1(a=0)$: C1 b
 0 0

$H_3(b=0)$: C2 C3
 0 0
 0 1
 1 0

The reduced set of combinations:

C1	C2	C3
0	0	0
0	0	1
0	1	0

Test Generation from Predicates

(Mathur, 2.7)

- Predicates might arise from requirements or might be embedded in the program
 - if(**printer_status is ON and paper_tray not empty**) then send_doc to printer (from requirements)
 - if(***p==1 || x!=0**) then y=1;
- A programmer might code a predicate correctly or might make an error, creating a fault.
- *Predicate testing*= testing to assure that there are no error in the *implementation of predicates*.
- Objectives:
 - to **generate** test cases **from predicates** such that
 - **any fault** from **a given class**
 - **are guaranteed** to be detected.

Fault model for Predicate testing: BOR, Boolean Operator

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 a Boolean variable. Four incorrect codings of this condition, each containing a Boolean operator fault, are given below.

$(a < b) \wedge (c > d) \wedge e$	Incorrect Boolean operator
$(a < b) \vee \neg(c > d) \wedge e$	Incorrect negation operator
$(a < b) \wedge (c > d) \vee e$	Incorrect Boolean operators
$(a < b) \vee (c > d) \wedge f$	Incorrect Boolean variable (f instead of e).

Note that a predicate might contain a single or multiple faults. The third example above is a predicate containing two faults.

Predicate constraints (1/2)

- $\text{BR}=\{\text{t,f}, <, =, >\}$, set of Boolean & Relational *constraints on boolean variables or relational expressions*.
- A *predicate constraints* for a predicate p with n boolean operators \wedge, \vee is a sequence of $n+1$ BR symbols.
Example. $p= b \wedge r < s \vee u \geq v$
 $C=(t,=,>)$ is a *constraint* on p
- A *test case* t satisfies a constraint if each component of p satisfies the corresponding constraints.
Example:
 - $t1=<\text{b=true, r=1, s=1, u=1, v=0}>$ satisfies the above C .
 - $t2=<\text{b=true, r=1, s=1, u=1, v=2}>$ does not satisfy C .
- If $t1, t2$ satisfies C , then $p(t1)=p(t2)$. We denote by $p(C)$ that value ($p(C)=\text{true}$, in our example).
- $p(C)=\text{true}$, C is called a *true constraint*.
 $p(C)=\text{false}$, C is called a *false constraint*.

Predicate constraints (2/2)

- We partition a set S of constraints for a given predicate p , $S = S_t \cup S_f$.

Example. $p = a < b \wedge c > d$.

Let us consider $S = \{\langle t, t \rangle, \langle t, f \rangle\}$.

We have $S_t = \{\langle t, t \rangle\}$, $S_f = \{\langle t, f \rangle\}$,

If $S = \{\langle f, t \rangle, \langle t, f \rangle, \langle f, f \rangle\}$, $S_t = \emptyset$.

BOR predicate testing Criterion (1/2)

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} .

- Example. $p = a < b \wedge c > d$

Let S denote a set of constraints on p_r ; $S = \{(\texttt{t}, \texttt{t}), (\texttt{t}, \texttt{f}), (\texttt{f}, \texttt{t})\}$. The following test set T satisfies constraint set S and the BOR-testing criterion.

$$T = \{ \begin{aligned} t_1: & \langle a = 1, b = 2, c = 1, d = 0 \rangle; \\ & \text{Satisfies } (\texttt{t}, \texttt{t}), \\ t_2: & \langle a = 1, b = 2, c = 1, d = 2 \rangle; \\ & \text{Satisfies } (\texttt{t}, \texttt{f}), \\ t_3: & \langle a = 1, b = 0, c = 1, d = 0 \rangle; \\ & \text{Satisfies } (\texttt{f}, \texttt{t}). \end{aligned} \}$$

BOR example, (2/2)

Predicate	t_1	t_2	t_3
$a < b \wedge c > d$	true	false	false
Single Boolean operator fault			
1 $a < b \vee c > d$	true	true	true
2 $a < b \wedge \neg c > d$	false	true	false
3 $\neg a < b \wedge c > d$	false	false	true
Multiple Boolean operator faults			
4 $a < b \vee \neg c > d$	true	true	false
5 $\neg a < b \vee c > d$	true	false	true
6 $\neg a < b \wedge \neg c > d$	false	false	false
7 $\neg a < b \vee \neg c > d$	true	true	true

Generating BOR adequate tests

$$A \times B = \{ (a, b) \mid a \in A, b \in B \}$$

For finite sets A and B , $A \otimes B$ is a minimal set of pairs (u, v) such that $u \in A$, $v \in B$, and each element of A appears at least once as u and each element of B appears at least once as v . Note that there are several ways to compute $A \otimes B$ when both A and B contain two or more elements.

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

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

$$A \otimes B = \{(t, f), (=, <), (>, <)\}; \text{ First possibility.}$$

$$A \otimes B = \{(t, <), (=, f), (>, <)\}; \text{ Second possibility.}$$

$$A \otimes B = \{(t, f), (=, <), (>, f)\}; \text{ Third possibility.}$$

$$A \otimes B = \{(t, <), (=, <), (>, f)\}; \text{ Fourth possibility.}$$

$$A \otimes B = \{(t, <), (=, f), (>, f)\}; \text{ Fifth possibility.}$$

$$A \otimes B = \{(t, f), (=, f), (>, <)\}; \text{ Sixth possibility.}$$

Procedure for generating a (minimal) BOR-constraints set of a predicate (1/2)

Procedure for generating a minimal BOR-constraint set from an abstract syntax tree of a predicate p_r

Input: An abstract syntax tree for predicate p_r , denoted by $AST(p_r)$. p_r contains only singular expressions.

Output: BOR-constraint set for p_r attached to the root node of $AST(p_r)$.

Procedure: BOR-CSET

Step 1 Label each leaf node N of $AST(p_r)$ with its constraint set $S(N)$. For each leaf $S_N = \{t, f\}$

Step 2 Visit each nonleaf node in $AST(p_r)$ in a bottom up manner. Let N_1 and N_2 denote the direct descendants of node N , if N is an AND or an OR-node. If N is a NOT-node, then N_1 is its direct descendant. S_{N_1} and S_{N_2} are the BOR-constraint sets for nodes N_1 and N_2 , respectively. For each nonleaf node N , compute S_N as follows:

Procedure for generating a (minimal) BOR-constraints set of a predicate (2/2)

2.1 N is an OR-node:

$$\begin{aligned} S_N^f &= S_{N_1}^f \otimes S_{N_2}^f \\ S_N^t &= (S_{N_1}^t \times \{f_2\}) \cup (\{f_1\} \times S_{N_2}^t) \\ \text{where } f_1 &\in S_{N_1}^f \text{ and } f_2 \in S_{N_2}^f \end{aligned}$$

2.2 N is an AND-node:

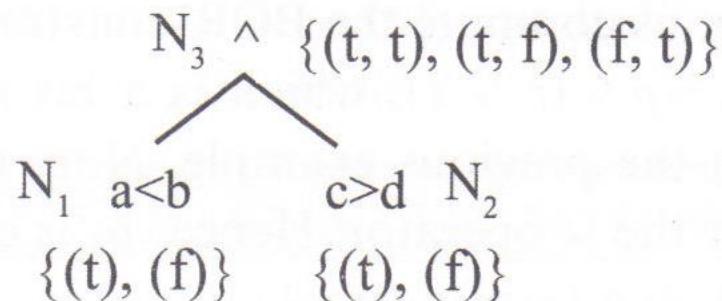
$$\begin{aligned} S_N^t &= S_{N_1}^t \otimes S_{N_2}^t \\ S_N^f &= (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f) \\ \text{where } t_1 &\in S_{N_1}^t \text{ and } t_2 \in S_{N_2}^t \end{aligned}$$

2.3 N is NOT-node:

$$\begin{aligned} S_N^t &= S_{N_1}^f \\ S_N^f &= S_{N_1}^t \end{aligned}$$

Step 3 The constraint set for the root of $AST(p_r)$ is the desired BOR-constraint set for p_r .

Example 1. $p = a < b \wedge c > d$



Remark. For **single** Boolean operator fault, **any two** constraints are sufficient

Remark. For **multiple** Boolean operator fault, three constraints are necessary.

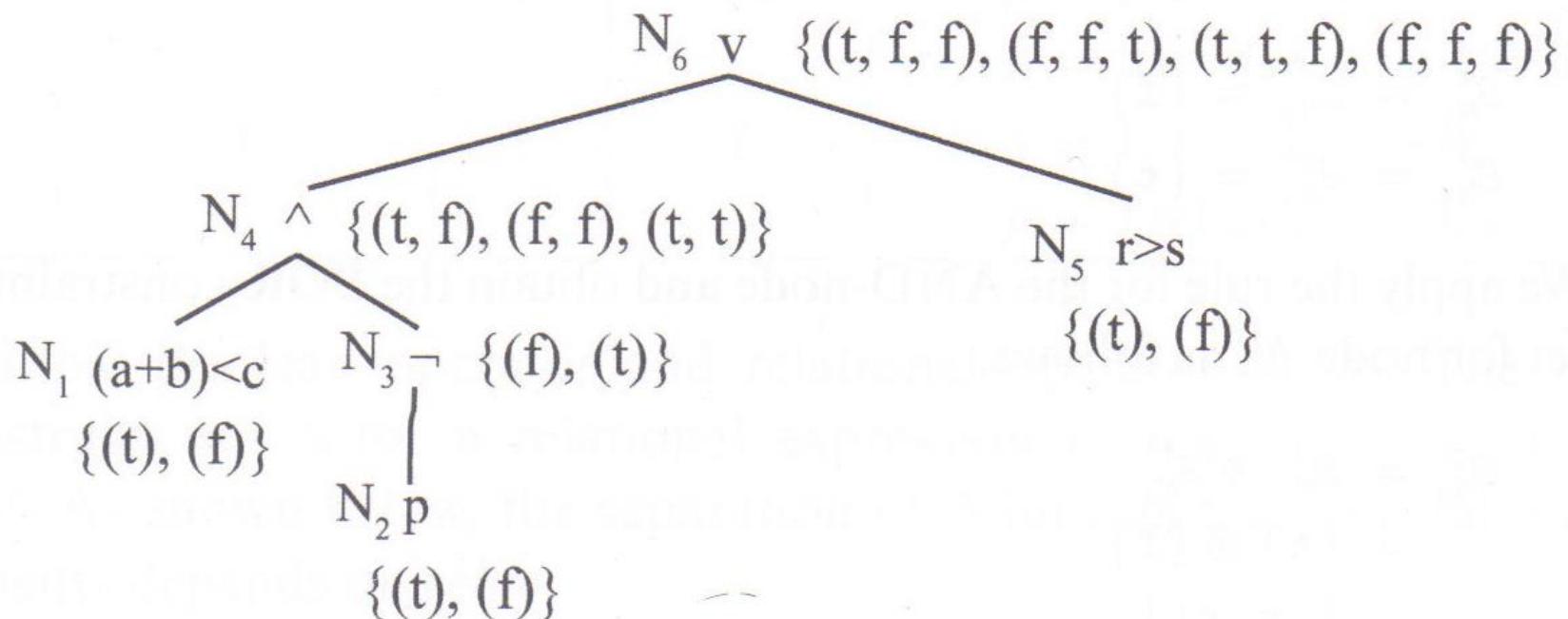
Example 1. $p = a < b \wedge c > d$ (continued)

	(t,t)	(f,t)	(t,f)
$p \wedge q$	t	f	f
$p \vee q$	t	t	t
$\neg p \wedge q$	f	t	f
$P \wedge \neg q$	f	f	t
Two are sufficient for single BOR fault			
Three are necessary for multiple BOR fault			
$\neg p \wedge \neg q$	f	f	f
$\neg p \vee q$	t	t	t
$p \vee \neg q$	t	f	t

Criterii de alcătuire a testelor

- Alegerea eșantioanelor ce vor face parte dintr-un test este guvernată de un ansamblu de condiții stabilite de cel ce face testarea (*criteriu* de alcătuire a testelor) și definește o mulțime (finită sau infinită) de teste.
- În anumite situații este convenabil să asimilăm un criteriu cu mulțimea pe care o definește. De pildă, în cazul unei partiții a datelor de intrare în clasele de echivalență c_1, \dots, c_n se poate lua drept criteriu produsul cartezian $C = c_1 \times \dots \times c_n$; acesta se numește criteriu *indus* de partiție.

Example 2. $p = (a + b < c) \wedge \neg p \vee (r > s)$



Criteriu compatibil

- Un criteriu C (multime de teste) se numește ***compatibil*** dacă testele din C sînt sau toate concludente sau toate neconcludente.
- În cazul unui criteriu compatibil este deci suficientă considerarea unui singur test
- nu există un algoritm general pentru a stabili dacă un criteriu este sau nu compatibil (problema compatibilităii criteriilor este ***nedecidabilă***, ca și alte probleme importante precum terminarea programelor, echivalența lor, corectitudinea etc.).

Relații între criterii

- Un criteriu C este ***mai tare*** (***mai fin***) decât alt criteriu D dacă pentru orice test (multime de eșantioane) $T \in C$ există un test $U \in D$ astfel că $U \subseteq T$; cu alte cuvinte, orice test din C poate fi adaptat, prin eliminarea unor eșantioane, astfel încât să satisfacă criteriul D.

Observație. Dacă o partiție c_1, \dots, c_n este o rafinare a partiției d_1, \dots, d_m atunci criteriul $c_1 \times \dots \times c_n$ este mai tare decât $d_1 \times \dots \times d_m$.

- Criteriul C este ***mai bun*** decât D dacă este ***mai fin*** decât D și există un test $U \in D$ neconcludent și un test $T \in C$ concludent cu proprietatea $U \subseteq T$; cu alte cuvinte, testul U devine concludent prin adăugarea de eșantioane noi, cerute de criteriul C.

Metoda cutiei transparente

Testare structurală

- Aceasta este o strategie de testare complementară metodei cutiei negre.
- prin această metodă testul se elaborează examinînd detaliile programului: instrucțiuni, date etc.
- Ea poate fi utilizată independent sau împreună cu metoda cutiei negre, pentru a rafina o partitură deja obținută.

(De pildă, în implementările funcției *binar* componenta *v[mijloc]* joacă un rol important. Aceasta poate sugera un criteriu nou de rafinare și anume considerarea cazurilor în care valoarea căutată *x* este egală, mai mică sau mai mare decât componenta *v[mijloc]*)

- Dificultățile în aplicarea acestei strategii sunt legate de prezența instrucțiunilor de decizie (*if*, *case* etc.), a celor iterative (*while*, *repeat*) sau a celor de transfer (*goto*). Acestea determină apariția unui mare număr de combinații în care instrucțiunile elementare ale programului (atribuiri, instrucțiuni de intrare/ieșire, apeluri de proceduri) pot fi executate.
- Ideal ar fi ca testul să solicite executarea fiecărei combinații posibile cel puțin o dată. Din punct de vedere practic acest lucru nu este realizabil, chiar în cazul programelor de dimensiune relativ mică.

Exemplu. În cazul

while a do

if b then M

else N

dacă se execută n iterații, sunt posibile 2^n combinații în care se execută instrucțiunile M și N. În testare exhaustivă ar trebui ca setul de date să conțină 2^n eșantioane. Presupunând că prelucrarea unui eșantion ar dura doar o milisecundă, în cazul $n = 40$ ar fi necesari aproape 30 de ani (??) de lucru neîntrerupt.

Criterii de acoperire

- În realitate nu se testează decât un număr limitat de astfel de combinații, considerate mai importante. În stabilirea acestui număr de combinații se utilizează diferite **criterii de acoperire**, dintre care cele mai importante sînt:
 - 1) *acoperirea tuturor instrucțiunilor*;
 - 2) *acoperirea tuturor arcelor*;
 - 3) *acoperirea tuturor condițiilor simple*;
 - 4) *acoperirea tuturor drumurilor*.

Acoperirea tuturor instrucțiunilor (INSTR)

- Acest criteriu se bazează pe observația evidentă că o eroare nu va fi descoperită dacă nu se execută deloc instrucțiunile ce o conțin. Prin urmare, testul trebuie să fie astfel ales încât *pentru fiecare instrucțiune elementară a programului să existe un eșantion de date care să provoace execuțarea sa*.
- Criteriul nu garantează semnalarea erorii.

$$m = \max(x, y);$$

$m := x;$

if $y \leq x$ **then** $m := y$

{ $m = \max(x, y)$ }

Testul $\{(x = 2, y = 2)\}$ provoacă execuțarea ambelor instrucțiuni de atribuire iar rezultatul obținut este corect (testul este neconcludent).

Dar algoritmul este incorrect deoarece se calculează de fapt

$$m = \min(x, y).$$

Testele ce satisfac acest criteriu sunt în general neconcludente deoarece același eşantion de date poate provoca executarea mai multor instrucțiuni; este deci foarte probabil să putem minimiza numărul eşantioanelor din test, menținând criteriul de acoperire a instrucțiunilor.

if $x \geq 0$ **then**

 writeln ('x pozitiv') {w1}

else writeln ('x negativ'); {w2}

if $y \geq 0$ **then**

 writeln ('y pozitiv') {w3}

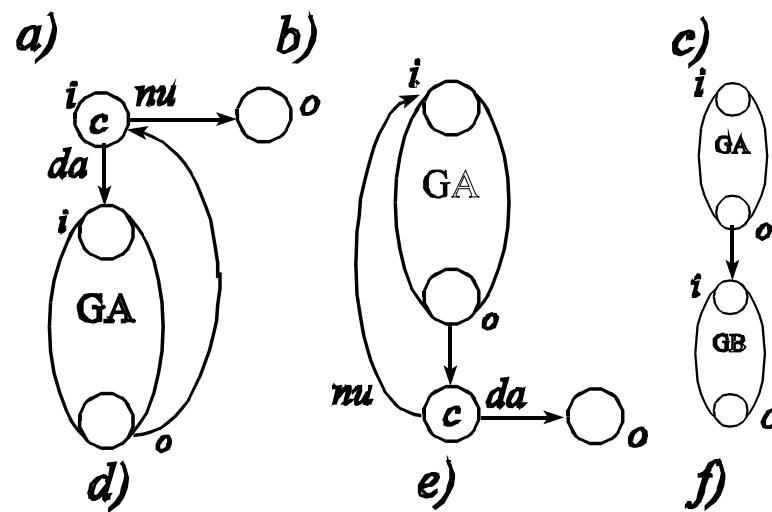
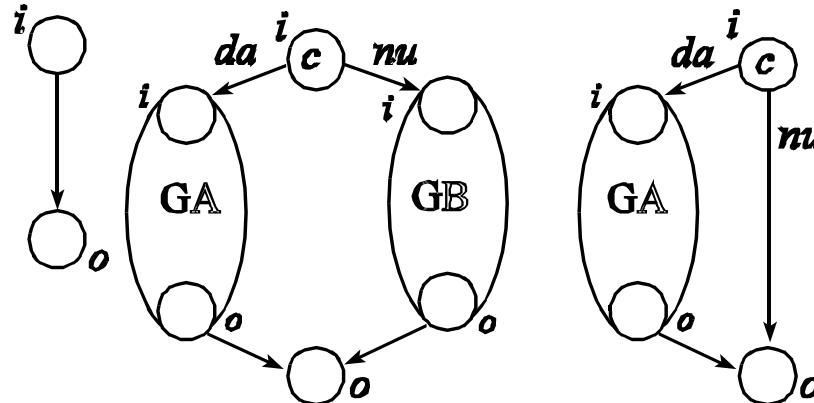
else writeln ('y negativ'); {w4}

- pentru executarea instrucțiunilor w1 și w2 sunt suficiente clasele de echivalență $\{(x, y) \mid x \geq 0\}$ și $\{(x, y) \mid x < 0\}$.
- Rafinând această partitură spre a asigura executarea lui w3 și a lui w4, obținem o partitură cu 4 clase de echivalență ce corespund celor 4 cadrane ale planului cartezian. Deci un test care conține 4 eşantioane (cîte unul din fiecare clasă) îndeplinește criteriul de acoperire a instrucțiunilor.
- În realitate este suficient un test cu doar două eşantioane: $\{(x = 2, y = 2), (x = -2, y = -2)\}$, care este mai slab decât primul.

Acoperirea tuturor arcelor(condițiilor): ARCE(COND)

- **graf de control** al programului (construit inductiv pentru programe ce conțin instrucțiuni elementare (adică atribuiri, instrucțiuni de intrare/ieșire, apeluri de proceduri), instrucțiuni de decizie (**if - then - else, if - then**) și instrucțiuni iterative (*while, repeat* etc.).
- Instrucțiuni elementare: 1a; arcul reprezintă instrucțiunea; dacă este necesar, se etichetează univoc;
- instrucțiune de decizie de forma
if c then A else B: 1b
if c then A: 1c
GA și GB grafurile asociate lui A, B; în vîrful de intrare se poate înscrie condiția c;
- instrucțiunea iterativă
while c do A: 1d
repeat A until c : 1e.
- Compunere secvențială, A ; B : 1f

Grafuri asociate instrucțiunilor



Exemple

- Instrucțiunilor următoare
`read(x);
if x = 0 then y := 0;
writeln(x);`
li se asociază graful de control din figura 2a
- **Observație.** Secvențele de forma celor din figura 3a cu proprietatea că singurele arce care intră sau ies din vîrfurile n_2, \dots, n_{k-1} unde $k \geq 3$ sănt cele din figură se pot înlocui prin secvențe de forma celor din figura 3b
(este posibil ca graful să devină un multigraf:pot exista mai multe muchii distincte care să lege aceeași pereche de vîrfuri)
- instrucțiunilor din exemplul 1 li se asociază multigraful din figura 2b
- Instrucțiunilor următoare:
`read(x);
while x > 0 do x := x – 1`
li se asociază multigraful din figura 4

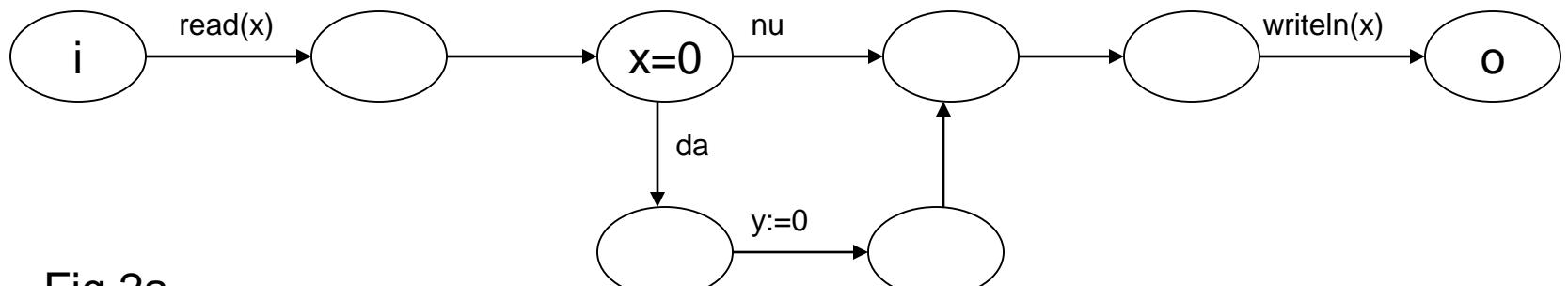


Fig 2a

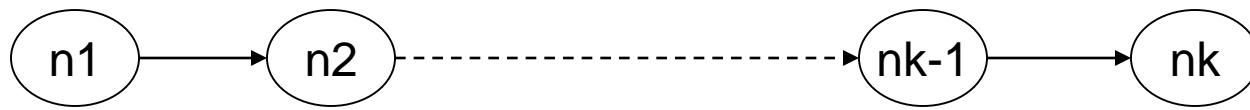


Fig 3a

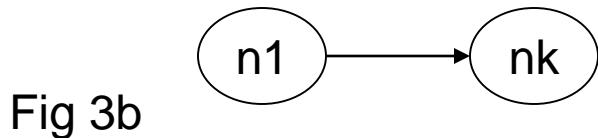


Fig 3b

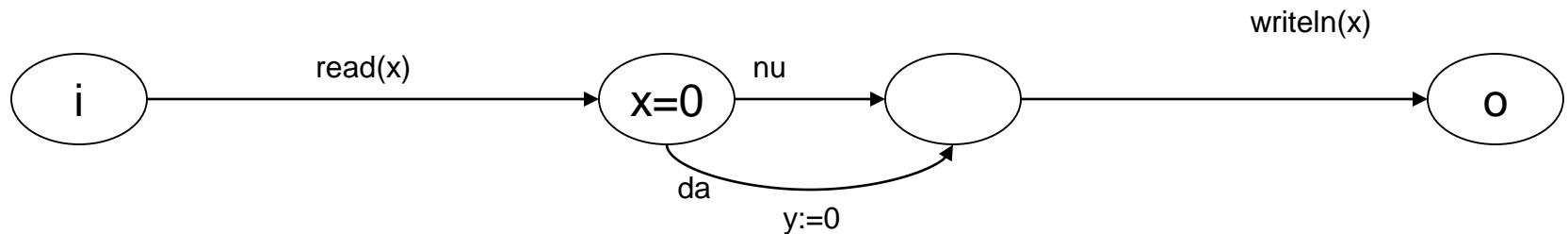


Fig 2b

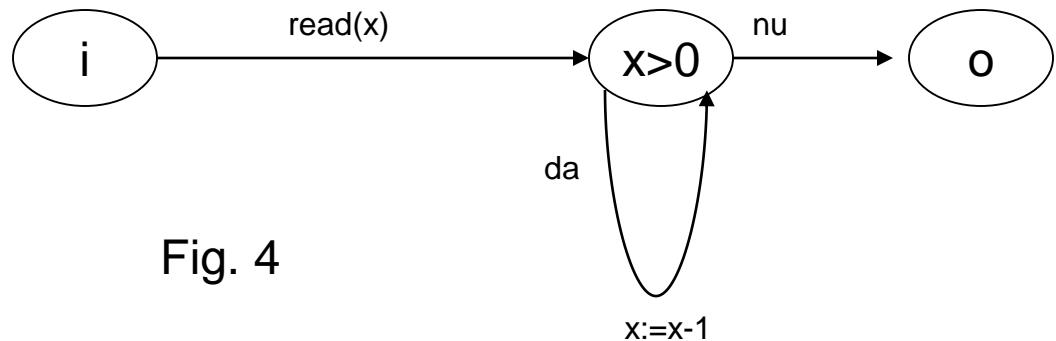


Fig. 4

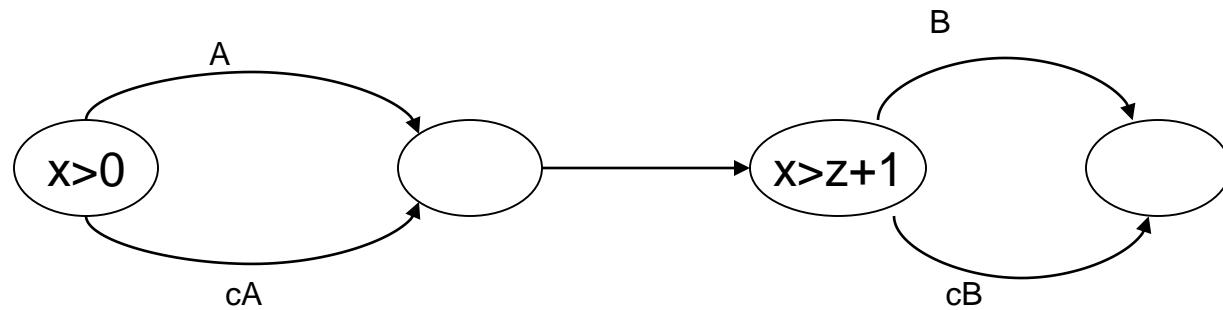


Fig. 6

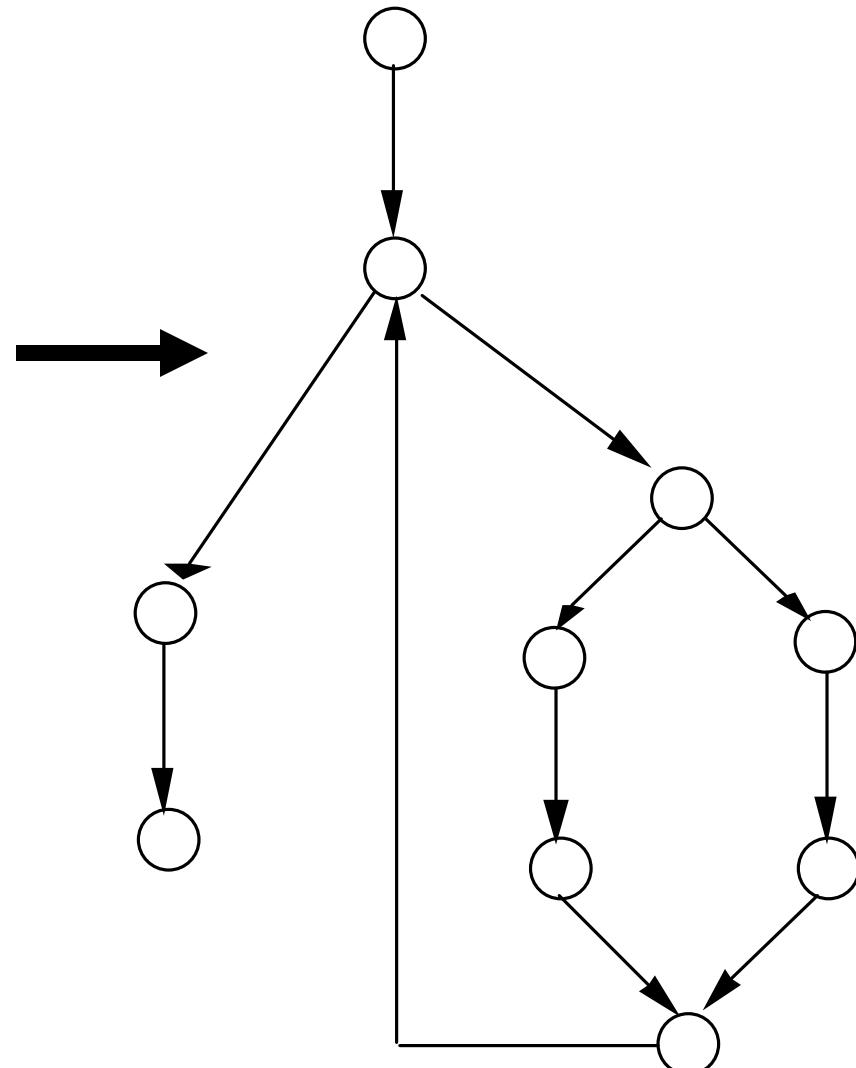
Exemple: Euclid's algorithm

begin

```
    read (x); read (y);
    while x ≠ y loop
        if x > y then
            x := x - y;
        else
            y := y - x;
        end if;
    end loop;
    gcd := x;

```

end;



- Criteriul de acoperire a tuturor arcelor: testul să fie astfel ales încât pentru orice arc din graf (multigraf) să existe eșantioane care să provoace parcurgerea arcului la executarea programului.

Exemplu: Pentru instrucțiunile din exemplul 1.setul de date $\{(x = 0); (x = 1)\}$ îndeplinește acest criteriu.

- **Observație.** *Criteriul de acoperire a tuturor arcelor este mai tare decât criteriul de acoperire a tuturor instrucțiunilor.*
Această relație este strictă; pentru instrucțiunile din exemplul 1 testul $\{x = 0\}$ asigură acoperirea tuturor instrucțiunilor, deși arcul corespunzînd situației $x \neq 0$ nu este parcurs.
- **Observație.** Criteriul de acoperire a tuturor arcelor are calitatea că impune alegerea unui test care să provoace evaluarea fiecărei condiții cel puțin o dată atât cu rezultatul true cât și cu rezultatul false.

Exemplu

- **Exercițiu.** Fie instrucțiunile

```
if x > z then y := 3  
else y := 2;  
if x > z + 1 then w := 3  
else w := 2
```

Găsiți un test cu număr minim de eșantioane pentru acoperirea tuturor arcelor. Variabilele sunt de tipul integer.

- R. Fie $A = \{(x,z) \mid x > z\}$; $cA = \{(x,z) \mid x \leq z\}$;

$B = \{(x,z) \mid x > z + 1\}$; $cB = \{(x,z) \mid x \leq z + 1\}$.

Orice test care conține câte un eșantion din fiecare mulțime este suficient pentru acoperirea tuturor arcelor (a se vedea figura 6). De pildă, se poate lua testul:

$x = 1, z = 0; x = 1, z = 1; x = 2, z = 0; x = 0, z = 0$.

- Aceste multimi nu sunt disjuncte. O partiție se poate obține considerînd următoarele clase de echivalență:

$$A \cap B = \{(x,z) \mid x > z + 1\};$$

$$A \cap cB = \{(x,z) \mid x = z + 1\};$$

$$cA \cap cB = \{(x,z) \mid x \leq z\}$$

($cA \cap B$ este vid_).

- Este suficient deci un test cu 3 eșantioane (cîte unul din fiecare clasă de echivalență). Se poate lua de pildă

$$x = 2, z = 0;$$

$$x = 2, z = 1;$$

$$x = 0, z = 1.$$

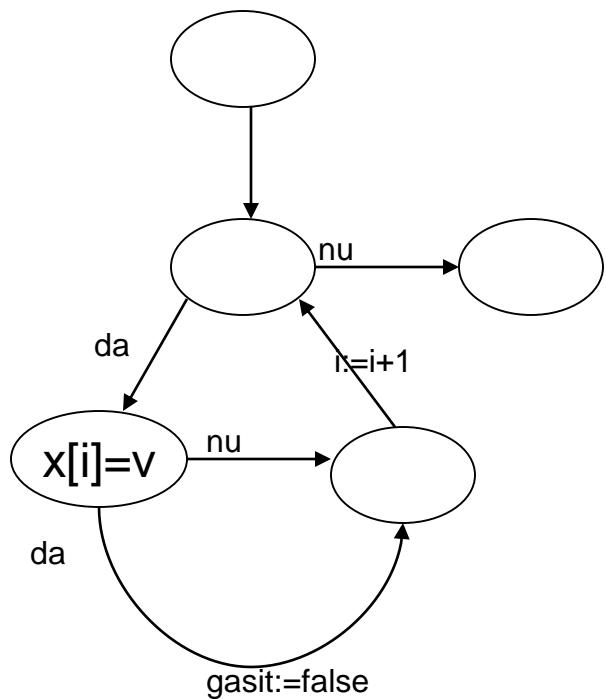
- Există însă și teste cu două eșantioane $(x_1, z_1) \in A \cap B, (x_2, z_2) \in cA \cap cB$.
- Teste cu un singur eșantion și care să satisfacă criteriul de acoperire a tuturor arcelor nu există deoarece orice eșantion nu poate asigura decît parcurgerea unui singur arc al instrucțiunii de decizie.

Exemplu

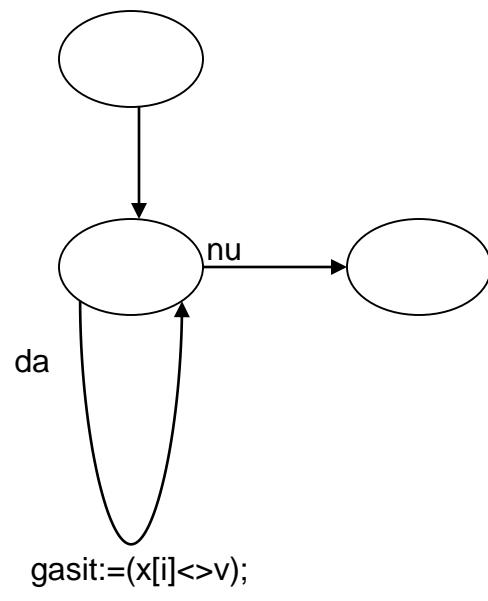
- Fie următorul program de căutare secvențială a elementului v în sirul x_1, \dots, x_n , unde $1 \leq n \leq 10$.

```
const dmax = 10;
var      x: array[1..dmax] of integer;
i,n: integer;
v: integer;
gasit: boolean;
begin
  read(n);
  for i := 1 to n do read(x[i]);
  read(v);
  gasit := false;
  i := 1;
  while (not gasit) and (i <= n) do begin
    if x[i] = v then gasit := true; {?}
    i := i + 1
  end;
end.
```

- Se consideră o variantă a acestui program, în care instrucțiunea de decizie din instrucțiunea iterativă se înlocuiește cu instrucțiunea de atribuire
 $\text{gasit} := (\text{x}[i] \neq v)$
 Arătați că aplicarea criteriului de acoperire a tuturor arcelor conduce la teste diferite în cele două cazuri.
R. Grafurile asociate sănt prezentate în figura 5.
 - În primul caz sănt necesare două eșantioane, pentru cazul $\text{x}[i] = v$ și respectiv $\text{x}[i] \neq v$. Se poate lua testul :
 $n = 1, x[1] = 10, v = 10;$
 $n = 1, x[1] = 10, v = 0.$
 Acest test este concludent (primul eșantion).
 - În al doilea caz este suficient un singur eșantion pentru acoperirea tuturor arcelor. Oricare din cele două eșantioane considerate anterior semnalează eroarea.
 - Algoritmul corect se obține prin înlocuirea instrucțiunii $\text{gasit} := \text{false}$ cu $\text{gasit} := \text{true}$. Instrucțiunea de decizie poate fi înlocuită prin $\text{gasit} := (\text{x}[i] = v)$.



a)

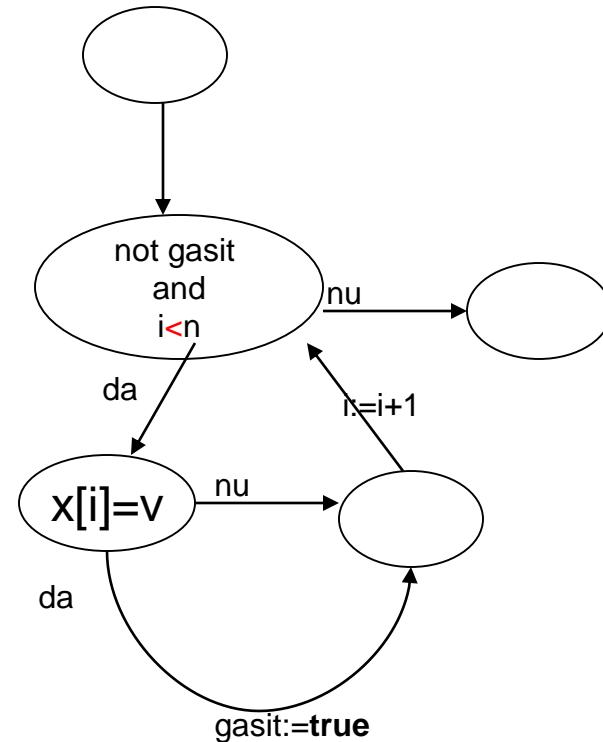


b)

Fig. 5

Acoperirea arcelor, criteriu incompatibil

Considerăm altă variantă ($i < n$):



$T_1 = \{(x=\{1,2\}, v=1), (x=\{1,2\}, v=3)\}$ acoperă arcele,
dar neconcludent

$T_2 = \{(x=\{1,2\}, v=1), (x=\{1,2\}, v=2)\}$ acoperă arcele,
concludent

Acoperirea tuturor condițiilor simple ½

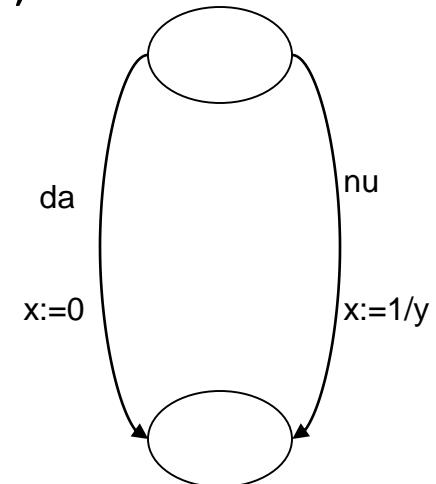
(SIMPL-COND+ARCE(COND))

- În cazul când în program există condiții compuse de forma **c and d** sau **c or d**, acoperirea tuturor arcelor nu conduce totdeauna la evaluarea condițiilor **c și d** atât cu rezultatul false cât și cu rezultatul true.
- Exemplul 1.§.1.2.2.3.** Fie instrucțiunile următoare:

if ($x \neq 0$) **and** ($y \neq 0$) **then**

$z := 0$

else $z := 1 / y$



- Testul $\{(x=1, y=1); (x=0, y=1)\}$ asigură acoperirea celor două arce. Se observă însă că situația $y = 0$, care ar fi condus la descoperirea unei erori nu a fost necesară pentru îndeplinirea acestui criteriu.

Acoperirea tuturor condițiilor simple 2/2

- Este util să considerăm un criteriu mai tare decât cel al acoperirii tuturor arcelor, prin care se cere ca testul: 1. să acopere toate arcele și 2. să asigure evaluarea tuturor condițiilor simple atât cu rezultatul true cât și cu rezultatul false.
- Acest criteriu este echivalent cu acoperirea tuturor arcelor, dacă programul conține doar condiții simple, în care nu apar operatorii logici **and** sau **or**.
- Evaluarea tuturor condițiilor simple nu asigură și acoperirea arcelor (false and true, true and false, același arc).
- La rescrierea instrucțiunilor ce conțin condiții compuse prin instrucțiuni cu condiții simple se va ține seama că

if c and d then

A

else B

if c then

if d then

A

și

else B

else B

sînt echivalente doar dacă operatorul **and** este evaluat *prin scurtcircuitare*.

În caz contrar rezultatele sînt diferite, după cum se poate observa în cazul cînd c este $x \neq 0$, d este $1/x > 0$ iar x are valoarea 0.

Acoperire multiplă a tuturor condițiilor simple (SIMPL-COND-MULT)

- testul trebuie să *asigure evaluarea condițiilor simple cu rezultatul true și false în toate combinațiile* cerute de structura condiției compuse în care ele apar. De pildă, în cazul c **and d** se obține pentru perechea de condiții simple (c, d) patru combinații: (false, false), (false, true) etc.
- acest criteriu este *strict mai tare* decât acoperirea tuturor condițiilor simple.

Exemplu

- Fie instrucțiunile

if ($x \neq 0$) **and** ($y \neq 0$) **then**

$z := 1$

else $z := 1 / (xy + x + y)$

- Testul $\{(x = 1, y = 0); (x = 0, y = 1); (x = 1, y = 1)\}$ acoperă toate condițiile simple dar nu provoacă evaluarea (false, false).
- El este de altfel neconcludent. Eroarea este descoperită chiar prin eşantionul $(x = 0, y = 0)$ care adăugat la testul considerat acoperă multiplu toate condițiile simple. Pentru acest program, acest criteriu este mai bun decât acoperirea tuturor condițiilor simple.

Acoperirea tuturor atribuirilor logice: LOG-ATR

- Se consideră un program P în care executarea este dirijată doar prin expresii logice (condiții); alte mecanisme (precum instrucțiunea **case**) sănătate sunt excluse.
- Fie c_1, \dots, c_n toate condițiile care apar în P. Se consideră toate sistemele de valori logice $w = (v_1, \dots, v_n)$, unde $v_i \in \{\text{true}, \text{false}\}$ pe care le pot lua aceste condiții.
- Se formulează următorul criteriu, numit **acoperirea tuturor atribuirilor logice**: testul trebuie să asigure evaluarea condițiilor astfel încât să se obțină toate sistemele w .
- acest criteriu este strict mai tare decât acoperirea tuturor arcelor.

Exemplu

- Fie instrucțiunile
if $x = 0$ **then** $z := 0$ **else** $z := 1;$
if $y = 0$ **then** $z := 0$ **else** $z := 1 / (x + y)$
- Testul $\{(x = 0, y = 1); (x = 1, y = 0)\}$ acoperă toate arcele dar nu acoperă toate atribuirile logice; sistemele (true, true) și (false, false) nu sînt evaluate.
- Testul trebuie completat cu alte două eșantioane, de pildă $(x=0, y=0)$ și $(x=1, y=1)$. Primul face ca testul să devină concludent (criteriul este mai bun).

Exemplu

- Să se arate că în cazul instrucțiunilor următoare

if $x <> 0$ **then** $y := 5$

else $z := z - x;$

if $z > 1$ **then** $z := z / x$

else $z := 0$

criteriul acoperirii tuturor atribuirilor logice asigură semnalarea erorii.

- R. Vom arăta că dacă testul este neconcludent atunci nu acoperă toate atribuirile logice. Fie x_0, z_0 valorile lui x și y înainte de prima instrucțiune și x_1, y_1 valorile după executarea acesteia. Dacă testul este neconcludent, din $x_1 = 0$ rezultă $z_1 \leq 1$. Dar $x_1 = x_0$; rezultă de aici că atribuirea (false, true) nu poate fi realizată.

Criteriul acoperirii multiple a condițiilor simple și criteriul atribuirilor logice sănăt incomparabile.

- R. Fie instrucțiunile următoare:

if ($x \neq 0$) **and** ($y \neq 0$) **then** {condiția C}

$a := 0$

else $a := 1;$

if ($z \neq 0$) **and** ($t \neq 0$) **then** {condiția D}

$a := 0$

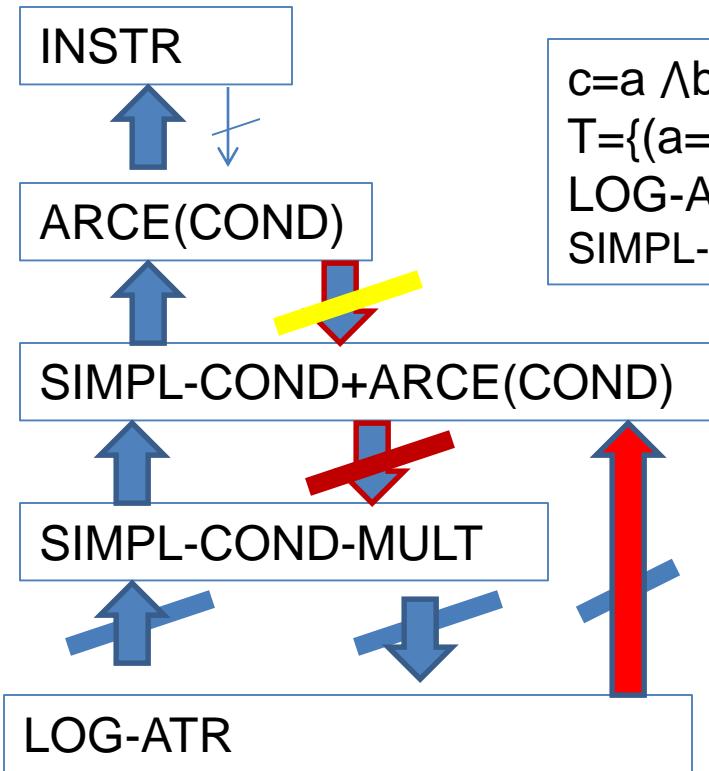
else $a := 1$

- Eșantioanele sănăt sisteme de valori (x, y, z, t).

- Testul $\{(0,0,0,0); (0,0,1,1); (1,1,0,0); (1,1,1,1)\}$ acoperă toate atribuirile logice ale condițiilor C și D dar nu acoperă multiplu condițiile simple din care acestea sănăt formate.

- Testul $\{(0,0,0,0); (0,1,0,1); (1,0,1,0); (1,1,1,1)\}$ acoperă multiplu condițiile simple dar nu acoperă toate atribuirile logice.

Relații între criterii, sinteză



$c=a \wedge b$ (o condiție)
 $T=\{(a=1,b=0),(a=1,b=1)\}$
LOG-ATR, dar nu
SIMPL-COND+ARCE(COND)



$C_1=a \wedge b, C_2=c \wedge d$
a b c d
0 0 0 0 LOG-ATR (si SIMPLE-COND+ARCE(COND))
0 1 1 1 dar nu
1 1 0 0 SIMPL-COND-MULT
1 1 1 1



PATHS
- strict mai tare decât toate

$C_1=a \wedge b, C_2=c \wedge d$
a b c d
0 0 0 0 SIMPL-COND-MULT
0 1 1 0 dar nu
1 0 0 0 LOG-ATR
1 1 1 1



Acoperirea tuturor drumurilor: PATHS

- În conformitate cu acest criteriu, *pentru fiecare drum al grafului de control testul conține eșantioane prin prelucrarea cărora executarea programului să urmeze drumul ales.*
- Acest criteriu este strict mai tare decât cele prezentate înainte.
- Fie instrucțiunile următoare:

if $x \neq 0$ **then** $z := 0;$

else $z := 1;$

if $y \neq 0$ **then** $z := 2$

else $z := 1/(x + y)$

Graful asociat este cel din figura 1.§.1.2.2.4.

- Testul $\{(x=0, y=1); (x=1, y=0)\}$ acoperă toate condițiile simple. El nu acoperă însă decât două drumuri din cele 4 posibile, fiind neglijat chiar cel care ar fi dus la semnalarea unei erori (cazul $x = 0$ și $y = 0$).
- Un test care satisface acest criteriu este, de exemplu
 $\{(x = 0, y = 1); (x = 1, y = 0);$
 $(x = 0, y = 0); (x = 1, y = 1)\}$

- acest criteriu exprimă un deziderat greu de realizat în practică datorită instrucțiunilor iterative.
- Criteriul poate fi însă aplicat în forme mai slabe, selectând pentru testare doar anumite drumuri considerate critice.
- Deși această selecție este dependentă de problemă, ea poate fi totuși ghidată de sugestia ca datele de test să provoace parcurgerea unor drumuri ce corespund:
 - numărului minim de iterații (adică nici o iterație în cazul *while* și o iterație în cazul *repeat*);
 - numărului maxim de iterații;
 - numărului mediu de iterații, dacă acesta poate fi stabilit într-un fel oarecare; în caz contrar se acceptă o execuție cu un număr semnificativ de iterații.

Exemplu

Să considerăm cazul unui algoritm de căutare secvențială a unei valori x printre componentele $v[1], \dots, v[n]$ ale vectorului v . Se presupune $0 \leq n \leq d_{\max}$.

```
const      dmax = 100;
type       index = 1..dmax;
           numar_componete = 0..dmax;
           element = integer;
           vector = array[index] of element;
function  secential (x : element; v : vector;n : numar_componete ) : boolean;
var      gasit : boolean;
         pozitie : index;
begin
  gasit := false; pozitie := 1;
  while  (not gasit)      and (pozitie < n) do begin
    gasit := (v[pozitie] = x);
    pozitie := pozitie + 1
  end;
  secential := gasit
end;
```

- Graful asociat acestui algoritm este prezentat în figura 2.§.1.2.2.4.
- Este important ca setul de date de test să asigure:
 - parcurgerea unui drum în care instrucțiunea *while* se execută în zero iterații ($n = 1$ sau $n=0$);
 - parcurgerea unui drum în care instrucțiunea *while* se execută în cîteva iterații, distingînd situațiile în care x face sau nu parte din secvența $v[1],...v[n]$.
 $n=2, n=dmax$.

Vom considera deci testul:

1.

$x = 10;$

$v = ();$

$n = 0;$

2. (*)

$x = 10;$

$v = (10);$

$n = 1;$

3.

$x = 10;$

$v = (10, 20, 30);$

$n = 3$

4.

$x = 20;$

$v = (10, 20, 30);$

$n = 3$

5. (*)

$x = 30$

$v = (10, 20, 30);$

$n = 3$

6.

$x = 0;$

$v = (10, 20, 30);$

$n = 3$

7.

$x = 0;$

$v = (10);$

$n = 1.$

Acesta este un test concludent, deoarece în cazurile 2 și 5 este semnalată o eroare. Aceste eşantioane au valoarea x pe ultima poziție a vectorului. Depanarea algoritmului constă în înlocuirea condiției pozitie $< n$ prin pozitie $\leq n$.

Aplicarea integrată a criteriilor de testare

- *Algoritmul de sortare rapidă (QuickSort)*. Acest algoritm a fost elaborat de C.A.R. Hoare [Ho 62]. El face parte din algoritmii bazați pe aplicarea metodei de programare *divide et impera* la problemele de sortare. În consecință, sortarea unui sir S care are cel puțin două elemente este concepută ca un proces în trei pași:
 - 1) **partitionarea** sirului S în două subșiruri (nevide) S_1 și S_2 ;
 - 2) **sortarea** sirurilor S_1 și S_2 , avînd ca rezultat sirurile R_1 și R_2 ;
 - 3) **combinarea** sirurilor R_1 și R_2 pentru obținerea sirului sortat R .

Algoritmul poate fi reprezentat prin următoarea procedură recursivă:

```
procedure Sortare (S : sir; var R : sir);  
  var S1, S2, R1, R2 : sir;  
begin  
  if S are cel puțin două elemente then begin  
    Partitionare (S, S1, S2);  
    Sortare (S1, R1);  
    Sortare (S2, R2);  
    Combinare (R1, R2, R)  
  end  
end;
```

- Această procedură conduce la plasarea algoritmilor de sortare în două categorii:
 - algoritmi în care **partiționarea este dificilă**, dar combinarea ușoară;
 - algoritmi în care partiționarea este ușoară, dar **combinarea este dificilă**.
- Exemplele reprezentative ale acestor categorii sunt algoritmii *QuickSort* și respectiv sortarea *prin interclasare*.
 - *QuickSort* procedura de partiționare este dificilă, sortarea prin interclasare împarte pur și simplu lista în două.
 - În compensație, la algoritmul *QuickSort* combinarea se realizează prin simpla alăturare a listelor sortate în timp ce la cealaltă metodă se face mai greu, prin procedeul de *interclasare*.

Testare funcțională, QuickSort

- Presupunem ca dimensiunea maxima este $d_{max}=4$.
- Metoda cutiei negre conduce la considerarea unor eșantioane care să cuprindă:
 - șiruri vide; (din afara domeniului)
 - șiruri cu un singur element; (marginea domeniului)
 - șiruri de dimensiune maxima; (marginea domeniului)
 - șiruri cu câteva elemente (două) elemente;

Apoi, rafinare:

- șiruri constante;
- șiruri strict descrescătoare;
- șiruri strict crescătoare.
- Avem deci 11 clase de echivalență și se poate utiliza testul următor:

1. k=0; x=();
2. k=1; x=(10);
3. k=2; x=(10,10);
4. k=2; x=(10,20);
5. k=2; x=(20,10);
6. k=3; x=(10,10,10);
7. k=3; x=(10,20,30);
8. k=3; x=(30,20,10);
9. k=4; x=(10,10,10,10);
10. k=4; x=(10,20,30,40);
11. k=4; x=(40,30,20,10).

Testare structurală, QuickSort

```
program Q_sort;
  const kmax = 10; {numarul maxim de elemente ale sirului}
  type   indice = 0 .. kmax+1;
         element = integer;
  var    x: array[indice] of element;
         k: indice; {numarul real de elemente ale sirului}

  procedure Partitionare      (m,n : indice; var i,j :indice);
    var t, pivot: element;
begin
  i:= m; j:= n; pivot:= x[ (m+n) div 2 ];
  while i <= j do begin
    while x[i] < pivot do i:= i+1;
    while x[j] > pivot do j:= j-1;
    if i <= j then begin
      t:= x[i]; x[i]:= x[j]; x[j]:=t;
      i:= i+1; j:= j-1
    end
  end
end;
end;
```

```
procedure QuickSort (m,n: indice );
    var i,j: indice;
begin
    if m < n then begin
        Partitionare( m,n,i,j);
        QuickSort (m, j);
        QuickSort (i, n)
    end
end;

var i:indice;
begin
    write(' dati numarul de elemente ale sirului ');
    readIn(k);
    write(' dati elementele sirului ');
    for i:= 1 to k do read (x[i]);
    QuickSort( 1,k );
    writeln; write( 'sirul ordonat este: ');
    for i:= 1 to k do write ( x[i] : 5 );
    writeln
end.
```

- Textul procedurii *Partitionare*, preluat din [AlAr 78] este atât de bine conceput încît aproape orice modificare îi afectează corectitudinea.
- De pildă, s-ar putea încerca înlocuirea condiției $x[i] < \text{pivot}$ prin $x[i] \leq \text{pivot}$ și a condiției $x[j] > \text{pivot}$ prin $x[j] \geq \text{pivot}$; această tentativă este justificată prin intenția de a " sări" și peste elementele egale cu pivotul pentru a accelera procesul de parcurgere. Spre exemplu, în cazul sirului
 $1, 2, 2, 2, 2, 2, 2, 5$
s-ar ajunge la situația finală
 $1^j, 2, 2, 2, 2, 2, 2, 5_i$
într-o singură iterație.
(în forma inițială, procedura *Partitionare* ajunge la forma finală
 $1, 2, 2, 2^j, 2_i, 2, 2, 5$
în trei iterării.)

- Se poate constata însă că procedura astfel modificată nu funcționează corect, existând posibilitatea ca indicii i și j să depășească domeniul de valori permise.
- De pildă, în cazul cînd $x = (1, 1, 1)$ are doar trei elemente, modificarea propusă ar conduce la obținerea valorii $i=4$ și deci expresia $x[4]<1$ ar fi nedefinită.
- această eroare este semnalată prin eșantioanele 3, 4, 5, 6, 9 și 10 ale testului anterior. Prin urmare, $x[i]<\text{pivot}$ ar putea fi înlocuită prin $x[i] \leq \text{pivot}$ numai în cazul $i \leq n$ iar $x[i] > \text{pivot}$ prin $x[i] = \text{pivot}$ numai în cazul $m \leq j$. Aceasta ar dubla însă numărul de teste realizate în instrucțiunile iterative.

- O altă propunere de "optimizare" s-ar putea face relativ la instrucțiunea

if $i \leq j$

then ...

unde se observă că în cazul $i = j$, interschimbarea valorilor lui $x[i]$ și $x[j]$ este inutilă.

- Se constată însă destul de ușor că schimbarea condiției $i \leq j$ prin $i < j$ face ca procedura modificată să fie incorectă. Într-adevăr, dacă $x = (10, 20)$, atunci $pivot = 10$ și după prima iterație se ajunge la situația:
 $10_j, 20_i$
- Din acest moment, instrucțiunea **while** $i \leq j \dots$ ciclează deoarece dacă $i = j$, atunci i și j nu mai sunt modificate.

- Pentru a remedia acest neajuns, se poate propune înlocuirea instrucțiunii
if $i \leq j$...
prin următoarele trei instrucțiuni:

if $i < j$ **then begin**

$t := x[i];$

$x[i] := x[j];$

$x[j] := t$

end;

$i := i + 1; j := j - 1$

- Față de propunerile anterioare, aceasta are șanse mari de a fi acceptată ca fiind corectă.
- Nici unul din eșantioanele testului anterior nu semnalează vreo eroare.
- Totuși, se poate constata că în cazul cînd

$x = (2, 3, 1, 1, 2, 4, 2, 5, 6)$

și deci pivot = 2, după prima iterație se ajunge la

2, 3_i, 1, 1, 2, 4_j, 2, 5, 6

De aici se obține situația finală

2, 2, 1_j, 1, 3, 4_i, 2, 5, 6

Se observă însă că elementele din subșirul x_{j+1}, \dots, x_{i-1} nu sunt egale între ele, deci procedura modificată nu mai realizează funcția pentru care a fost creată.

- Trebuie recunoscut că dacă textul initial al procedurii Partitionare ar fi fost cel propus prin ultima modificare, eroarea ar fi avut puține șanse de a fi descoperită prin teste.
- Mai mult, sensibilitatea textului acestei proceduri la modificări ce păreau plauzibile (cel puțin la prima vedere) ne face să ne îndoim de corectitudinea sa și să resimțim nevoia unei justificări riguroase a bunei sale funcționări.
- Aceasta poate fi obținută prin metode statice de verificare (metode formale de demonstrare a corectitudinii programelor).

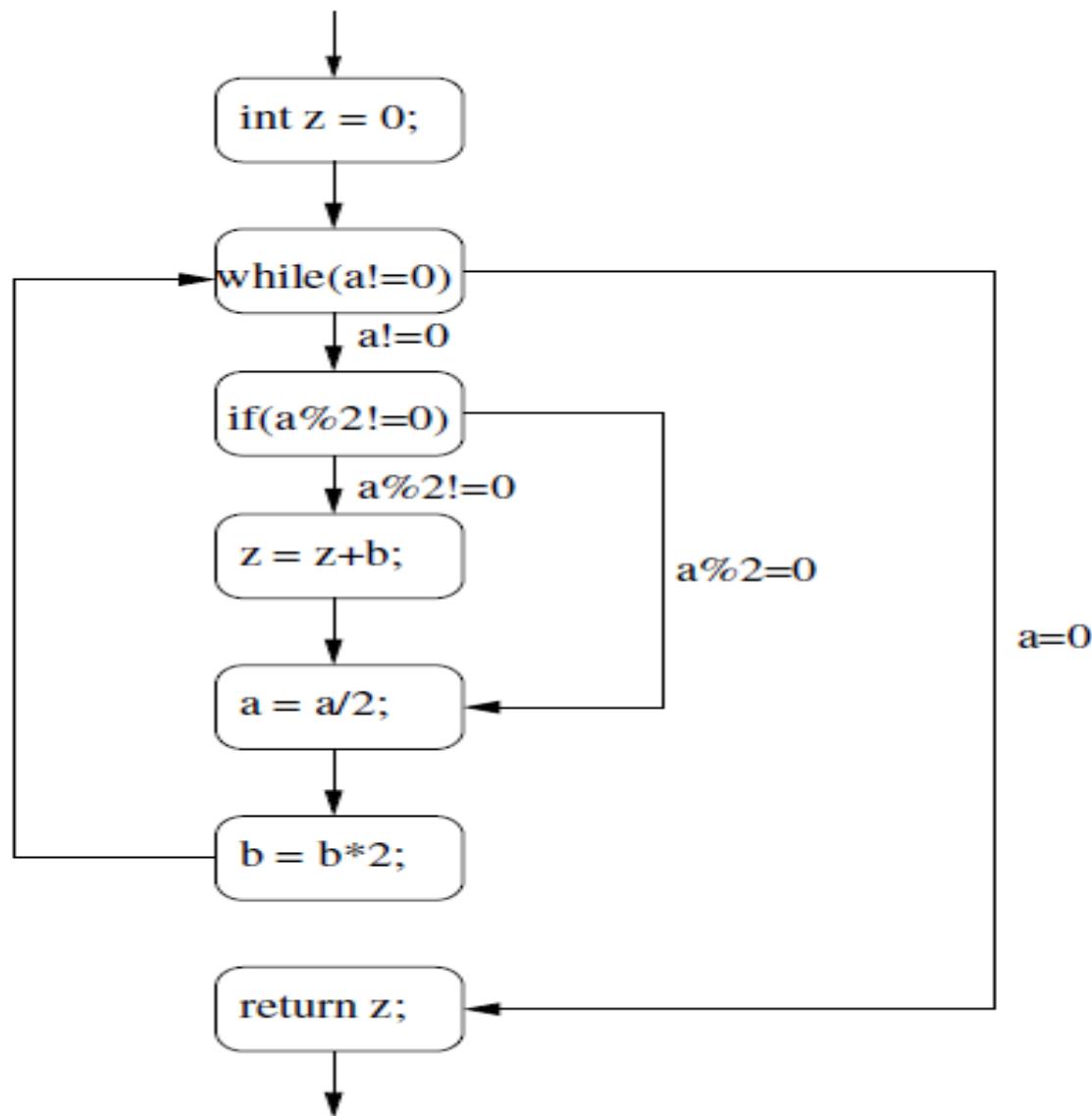
Exercise: Russian Multiplication

(Moa Johansson, Wolfgang Ahrendt, Vladimir Klebanov: Testing, Debugging, and Verification)

<Testing docs 2015\Testing3.pdf>

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

Russian Multiplication, control flow graph



Russian Multiplication, coverage criteria

Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- ▶ [a=3, b=3] SC
- ▶ [a=0, b=2] neither
- ▶ [a=4, b=1] SC and BC

Number of execution paths is 2^{31}
Size of a test suite satisfying PC is 2^{31}

PC cannot be achieved in practice

Evaluarea testelor prin mutantă

Mutation testing (mutation analysis)

(curs Florentin Ipate)

- Tehnica de evaluare a unui set de teste pentru un program (avand un set de teste generat, putem evalua cat de eficient este, pe baza rezultatelor obtinute de acest test asupra mutantilor programului)
- Mutation = modificare f. mica (din punct de vedere sintactic) a unui program
- Pentru un program P, un mutant M al lui P este un program obtinut modificand f. usor P; M trebuie sa fie corect din punct de vedere sintactic.

Exemplu

Program P

```
begin
```

```
    int x, y;
```

```
    read(x, y);
```

```
    if (x > 0)
```

```
        write(x+y)
```

```
    else
```

```
        write(x*y)
```

```
end
```

Mutant M1

```
begin
```

```
    int x, y;
```

```
    read(x, y);
```

```
    if (x > =0)
```

```
        write(x+y)
```

```
    else
```

```
        write(x*y)
```

```
end
```

Mutant M2

begin

int x, y;

read(x, y);

if (x > 0)

write(x-y)

else

write(x*y)

end

Mutant M3

begin

int x, y;

read(x, y);

if (x > 0)

write(x+y+1)

else

write(x*y)

end

Tehnica Mutation testing

- -Generarea mutantilor pentru programul P (folosind o multime de operatori de mutatie)
- -Rularea setului de teste asupra programului P si setului de mutanti; daca un test distinge intre P si un mutant M spunem ca *P omoară mutantul M.*

Mutanti de primul ordin / mutanti de ordin mai mare (first-order/higher-order mutants)

- First-order mutants = mutanti obtinuti facand o singura modificare in program
- n-order mutants = mutanti obtinuti facand n modificari in program
- n-order mutant = first-order mutant of a (n-1)-order mutant, $n > 1$
- n-order mutant, $n > 1$, sunt numiti higher-order mutants

Mutant de ordin 2

begin

```
int x, y;  
read(x, y);  
if (x >= 0) write(x-y)  
else write(x*y)
```

end

In general, in practica sunt folositi doar mutantii de ordin 1. Motive:

- Numarul mare de mutanti de ordin 2 sau mai mare
- Coupling-effect

Principiile de baza ale mutation testing

- Competent programmer hypothesis (CPH):
 - Pentru o problema data, programatorul va scrie un program care se afla in vecinatatea unui program care rezolva in corect problema (si deci, erorile vor fi detectate folosind munati de ordinul 1)
- Coupling effect
 - Datele de test care disting orice program care difera cu putin de programul corect sunt suficient de puternice pentru a distinge erori mai complexe.
 - Rezultate experimentale arata ca un set de teste care distinge un program de mutantii sai de ordin 1 este aproape de a distinge programul de mutantii de ordin 2
 - Explicatie intuitiva: in general erorile simple sunt mai greu de detectat. Erorile complexe pot fi detectate de aproape orice test

Strong mutation/ weak mutation

- Un test t omoara mutantul M (distinge M fata de P) daca cele doua se comporta diferit pentru testul t.
- Intrebare: cand observam comportamentul celor doua programe ?
 - Testul t aduce pe P si M in stari diferite - se observa starea programului (valorile variabilelor afectate) dupa executia instructiunii mutate.
 - Schimbarea starii se propaga la sfarsitul programului - se observa valoarile variabilelor returnate si alte efecte (schimbarea variabilelor globale, fisiere, baza de date), imediat dupa terminarea programului
- *Weak mutation: prima conditie este satisfacuta*
- *Strong mutation: ambele conditii sunt satisfacute*

Exemplu

Program P

```
begin
    int x, y;
    read(x, y);
    y := y+1;
    if (x > 0) write(x)
    else write(y)
end
```

Mutant M

```
begin
    int x, y;
    read(x, y);
    y := y-1;
    if (x > 0) write(x)
    else write(y)
end
```

- Testul (1, 1) distinge intre P si M d.p.d.v. weak mutation, dar nu distinge intre P si M d.p.d.v. strong mutation
- Testul (0, 1) distingre intre P si M d.p.d.v. strong mutation
- Strong mutation: mai puternica. Se asigura ca testul t detecteaza cu adevarat problema
- Weak mutation: necesita mai putina putere de calcul; strans legata de ideea de acoperire

Mutanti echivalenti

- Un mutant M a lui P se numeste echivalent daca el se comporta identic cu programul P pentru *orice date de intrare*. *Altfel, se spune ca M poate fi distins de P.*
- Din punct de vedere teoretic: in general, problema determinarii daca un mutant este echivalent cu programul parinte este nedecidabila (este echivalenta cu halting problem)
- In practica: determinarea echivalentei se face prin analiza codului (formal methods)
- Determinarea mutantilor echivalenti poate fi un proces foarte complex – principala problema practica a tehnicii mutation testing (avem nevoie sa decidem daca mutantii sunt sau nu echivalenti pentru a putea evalua eficiența testelor)

Utilitatea mutation testing

- Evaluarea unui set de date existent (si construirea de noi teste, daca testele existente nu omoara toti mutantii)
- Detectarea unor erori in cod
- **Evaluarea unui set de date existent (exemplu)**

begin

```
int x, y;  
read(x, y);  
if (x > 0) write(x+y)  
else write(x*y)
```

end

- Consideram urmatorii operatori de mutatie:
- + inlocuit de –
- * inlocuit de /
- o variabila sau o constanta x este inlocuita de x+1

begin

int x, y;

read(x, y);

if (x > 0)

M1 if ($x+1 > 0$)

M2 if ($x > 0+1$)

 write(x+y)

M3 write($x+1+y$)

M4 write($x+y+1$)

M5 write($x-y$)

else

 write($x*y$)

M6 write($(x+1)*y$)

M7 write($x*(y+1)$)

M8 write(x/y)

end

- Set de teste $T = \{t_1, t_2, t_3, t_4\}$
 $t_1 = (0, 0), t_2 = (0, 1), t_3 = (1, 0), t_4 = (-1, -1)$

P	t1	t2	t3	t4	Mutant distins
P(t)	0	0	1	1	
M1(t)	0	1	NE	NE	Y
M2(t)	0	0	0	NE	Y
M3(t)	0	0	2	NE	Y
M4(t)	0	0	2	NE	Y
M5(t)	0	0	1	1	N
M6(t)	0	1	NE	NE	Y
M7(t)	0	0	1	0	Y
M8(t)	ND	NE	NE	NE	Y

- Mutanti nedistinsi (alive) = {M5}
- Intrebare: Este M5 mutant echivalent ?
Raspuns: Nu. (1, 1) distinge intre P si M5

Mutation score

- **Mutation score: $MS(T) = D/(L+D)$, unde**
- D – numarul de mutanti distinsi
- L – numarul de mutanti nedistinsi (live mutants) neechivalenti
- Pentru exemplu: $MS(T) = 7/8$

Detectarea erorilor folosind mutatia (exemplu)

Program P

```
begin
    int x, y
    read(x)
    y = 1
    y = 2  <--- Eroare: instructiune lipsa
    if (x < 0) y = 3
    if (x>2) y = 4
    write(y)
end
```

Mutant M

```
...
if (x < 1) y = 3
...
```

- Aratam ca mutantul M genereaza teste care detecteaza eroarea
- Pentru ca un test t sa distinga intre P si M trebuie ca:
 - -Reachability: Instructiunea mutata sa fie executata la aplicarea lui t
 - -State infection: Instructiunea mutata sa afecteze statea programului
 - -State propagation: Schimbarea de state sa se propage in exterior

- Pentru exemplul dat, pentru ca un test t sa distinga intre P si M:
 - - Reachability: TRUE
 - - State infection: $(x < 1 \wedge \neg(x < 0))$
 - - State propagation: $\neg(x > 2)$
- Conditia rezultata: $(x = 0) \wedge (x \leq 2) \Leftrightarrow x = 0$
- Pentru $x = 0$ programul corect intoarce 2 in timp ce programul gresit returneaza 3

Operatori de mutație

- **Operator de mutatie**
 - Operator de mutatie = Regula care se aplica unui program pentru a crea mutanti (e.g. inlocuirea/adaugarea/stergerea unor operanzi, stergerea unor instructiuni, etc.)
 - Programul nou obtinut trebuie sa fie valid din punct de vedere sintactic
- **Operator de mutatie din Java (MuJava)**
 - Traditional mutation operators (method-level operators) – operatori aplicabili oricarui limbaj procedural
 - Class mutation operator – operatori specifici paradigmii orientate pe obiect si sintaxei Java
 - Incapsulare
 - Mostenire
 - Polimorfism si dynamic binding
 - Suprascrierea metodelor
 - Java specific
- **Operatori traditionali** <http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>
- **Operatori de clasa** <http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf>

Description of Class Mutation Mutation Operators for Java

- The class mutation operators are classified into four groups, based on the language features that are affected.
 1. Encapsulation
 2. Inheritance
 3. Polymorphism
 4. Java-Specific Features
- Generally, all the behaviors of mutation operators fall under one of the three categories:
 - (1) delete,
 - (2) insert, and
 - (3) change a target syntactic element.

Yu-Seung Ma

Electronics and Telecommunications Research Institute, Korea
ysma@etri.re.kr

Jeff Offutt

Information and Software Engineering, George Mason University
offutt@ise.gmu.edu

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IGR	Overriding method rename
	ISI	super keyword insertion
	ISD	super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
	PNC	new method call with child class type
Inheritance	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
	JTI	this keyword insertion
Polymorphism	JTD	this keyword deletion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change
Java-Specific Features		

Table 1: Mutation Operators for Inter-Class Testing

Encapsulation

- AMC { Access modifier change: The AMC operator changes the access level for instance variables and methods to other access levels. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct.

Original Code

```
public Stack s;
```

AMC Mutants

- ▲ private Stack s;
- ▲ protected Stack s;
- ▲ Stack s;

IHD

IHD { Hiding variable deletion: The IHD operator deletes a hiding variable, a variable in a subclass that has the same name and type as a variable in the parent class.

Original Code

```
class List {  
    int size;  
    ... ...  
}  
class Stack extends List {  
    int size;  
    ... ...  
}
```

IHD Mutant

```
class List {  
    int size;  
    ... ...  
}  
class Stack extends List {  
    Δ      // int size;  
    ... ...  
}
```

IHI

- IHI { Hiding variable insertion: The IHI operator inserts a hiding variable into a subclass. It is a reverse case of IHD.
- The ability of a subclass to override a method declared by an ancestor allows a class to modify the behavior of the parent class.
- When there are overriding methods, it is important for testers to ensure that a method invocation actually invokes the intended method.

Original Code

```
class List {  
    int size;  
    ... ...  
}  
class Stack extends List {  
    ... ...  
}
```

IHI Mutant

```
class List {  
    int size;  
    ... ...  
}  
class Stack extends List {  
     $\Delta$     int size;  
    ... ...  
}
```

IOD

- IOD { Overriding method deletion: The IOD operator deletes an entire declaration of an over-riding method in a subclass so that references to the method uses the parent's version. The mutant act as if there is no overriding method for the method.

Original Code

```
class Stack extends List {  
    ... ...  
    void push (int a) { ... }  
}
```

IOD Mutant

```
class Stack extends List {  
    ... ...  
    △      // void push (int a) { ... }  
}
```

IOP

- IOP { Overridden method calling position change: Sometimes, an overriding method in a child class needs to call the method it overrides in the parent class. This may happen if the parent's method uses a private variable *v*, *which means the method in the child class may not modify v directly.*
- However, an easy mistake to make is to call the parent's version at the wrong time, which can cause incorrect state behavior.
- The IOP operator moves calls to overridden methods to the first and last statements of the method and up and down one statement.

Original Code

```
class List {  
    ... ...  
    void SetEnv()  
        {size = 5; ... }  
}
```

```
class Stack extends List {
```

```
    ... ...  
    void SetEnv() {  
        super.SetEnv();  
        size = 10;  
    }  
}
```

IOP Mutant

```
class List {  
    ... ...  
    void SetEnv()  
        {size = 5; ... }  
}
```

```
class Stack extends List {
```

```
    ... ...  
    void SetEnv() {  
        size = 10;  
        super.SetEnv();  
    }  
}
```

IOR

- IOR { Overridden method rename: The IOR operator is designed to check if an overriding method adversely affects other methods. Consider a method *m()* that calls another method *f()*, both in a class *List*. Further, assume that *m()* is inherited without change in a child class *Stack*, but *f()* is overridden in *Stack*. When *m()* is called on an object of type *Stack*, it calls *Stack*'s version of *f()* instead of *List*'s version. In this case, *Stack*'s version of *f()* may have an interaction with the parent's version that has unintended consequences.
- The IOR operator renames the parent's versions of these methods so that the overriding cannot affect the parent's method. It models the situation that the overriding method is declared as a new method with different name in the child class.
- The super keyword is used to access parent's members (variables or methods) within the child class. When there is variable shadowing or method overriding, use of the super keyword should be careful because it changes the reference to variables or methods from the child class to super class.

IOR

Original Code

```
class List {  
    ... ...  
    void f0() { ... }  
    void m0() {... f0(); ... }  
}
```

```
class Stack extends List {  
    ... ...  
    void f0() { ... }  
    void g0() {... f0(); ... }  
}
```

IOR Mutant

```
class List {  
    ... ...  
    void f'0() { ... }  
     $\Delta$  void m0() {... f'0(); ... }  
}
```

```
class Stack extends List {  
    ... ...  
    void f0() { ... }  
     $\Delta$  void g0() {... f0(); ... }  
}
```

ISI

- ISI { super keyword insertion: The ISI operator inserts the super keyword so that a reference to the variable or the method goes to the overridden instance variable or method.
- The ISI operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

Original Code

```
class Stack extends List {  
    ...  
    int MyPop( ) {  
        ...  
        return val*num;  
    }  
}
```

ISK Mutant

```
class Stack extends List {  
    ...  
    int MyPop( ) {  
        ...  
        return val*super.num;  
    }  
}
```

ISD

- ISD { super keyword deletion: The ISD operator deletes occurrences of the super keyword so that a reference to the variable or the method goes to the overriding instance variable or method. It is a reverse case of the ISI.
- Although constructors are not inherited the way other methods are, a constructor of the superclass is invoked when subclasses are instantiated.
- When we create new objects of a derived class, the default constructor (no arguments) for the parent class is automatically called first, then the constructor of the derived class is called. However, the subclass can use the super keyword to call a specific parent class constructor. This is usually done to pass arguments to one of the parent class's non-default constructors.

ISD

Original Code

```
class Stack extends List {  
    ... ...  
    int MyPop( ) {  
        ... ...  
        return val*super.num;  
    }  
}
```

ISK Mutant

```
class Stack extends List {  
    ... ...  
    int MyPop( ) {  
        ... ...  
        return val*num;  
    }  
}
```

Δ

IPC

- IPC : Explicit call of a parent's constructor deletion
- The IPC operator deletes super constructor calls, causing the default constructor of the parent class to be called. To kill mutants of this type, it is necessary to find a test case for which the parent's default constructor creates an initial state that is incorrect.

Original Code

```
class Stack extends List {  
    ...  
    Stack (int a) {  
        super (a);  
        ...  
    }  
}
```

IPC Mutant

```
class Stack extends List {  
    ...  
    Stack (int a) {  
        // super (a);  
        ...  
    }  
}
```

Polymorphism

- Object references can have different types with different executions. That is, object references may refer to objects whose actual types differ from their declared types.
- The actual type can be from any type that is a subclass of the declared type.
- Polymorphism allows the behavior of an object reference to be different depending the actual type.
- Therefore, it is important to identify and exercise the program with all possible type bindings.
- The polymorphism mutation operators are designed to ensure this type of testing.

PNC

- PNC { new method call with child class type: The POI operator changes the instantiated type of an object reference. This causes the object reference to refer to an object of a type that is different from the declared type. In the example below, class Parent is the parent of class Child.

Original Code

```
Parent a;  
a = new Parent();
```

PNC Mutant

```
Parent a;  
△ a = new Child();
```

PMD

- PMD { Member variable declaration with parent class type:
The PMD operator changes the declared type of an object reference to the parent of the original declared type. The instantiation will still be valid (it will still be a descendant of the new declared type).
- To kill this mutant, a test case must cause the behavior of the object to be incorrect with the new declared type.

Original Code

```
Child b;  
b = new Child();
```

PMD Mutant

△ Parent b;
b = new Child();

PPD

- PPD { Parameter variable declaration with child class type: The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables.
- It changes the declared type of a parameter object reference to be that of the parent of its original declared type.

Original Code

```
boolean equals (Child o) { ... }
```

PPD Mutant

△

```
boolean equals (Parent o) { ... }
```

PCI

- PCI { Type cast operator insertion: The PCI operator changes the actual type of an object reference to the parent or child of the original declared type. The mutant shows different behavior when the object to be casted has hiding variables or overriding methods.

Original Code

```
Child cRef;  
Parent pRef = cRef;  
pRef.toString();
```

PCI Mutant

```
Child cRef;  
Parent pRef = cRef;  
Δ   ((Child)pRef).toString();
```

PCD

- PCD { Type cast operator deletion: The PCD operator deletes type casting operator. It models a reverse case of PCI.

Original Code

```
Child cRef;  
Parent pRef = cRef;  
((Child)pRef).toString();
```

PPD Mutant

```
Child cRef;  
Parent pRef = cRef;  
△ pRef.toString();
```

PCC

- PCC { Cast type change: The PCC operator change the type that a variable is to be cast into. The change is occurred with subclasses or ancestors of the type.

Original Code

((Parent)ref).toString();

PPD Mutant

△ ((Child)ref).toString();

PRV

- PRV { Reference assignment with other compatible type:
Object references can refer to objects of types that are descendants of its declared type. The PRV operator changes operands of a reference assignment to be assigned to objects of subclasses.
- In the example below, *obj* is of type Object, and in the original code it is given an object of type String. In the mutated code, it is given an object of type Integer.

Original Code

```
Object obj;  
String s = "Hello";  
Integer i = new Integer(4);  
obj = s;
```

PRV Mutant

```
Object obj;  
String s = "Hello";  
Integer i = new Integer(4);  
△ obj = i;
```