

DIM0436

26. Testes de caixa branca  
Cobertura estrutural

20141104

# Sumário

- 1 Introdução
- 2 Cobertura do fluxo de controle
- 3 Cobertura do fluxo de dados

- 1 Introdução
- 2 Cobertura do fluxo de controle
- 3 Cobertura do fluxo de dados

# Definições

## Caixa preta

Testes de caixa preta são feitos a partir das especificações do código

- dados de concepção
- interfaces das funções / módulos
- modelo formal / semi-formal

## Caixa branca

Baseado no código fonte.

- Interessante se o programa não for bem especificado

# Teste de caixa preta (resumo)

- Não precisa do conhecimento da estrutura interna do sistema
- Precisa da especificação funcional do sistema : relativamente pequeno
- Assegura a conformidade entre especificações e a implementação
  - ▶ Problema com detalhes finos de programação
- Oráculos de testes mais fáceis, concretização problemática
- Bom para o teste de unidade e o teste de sistema

# Teste de caixa branca (essa aula)

- Precisa da estrutura interna do sistema
- Baseado no código fonte: preciso mas maior do que a especificação
- Dados de testes mais finas e numerosas
- Concretização simples, oráculos difíceis
- Sensíveis aos defeitos finos de programação, mas não percebe funcionalidades ausentes

# Teste exaustivo de caixa branca ?

## Problema do critério

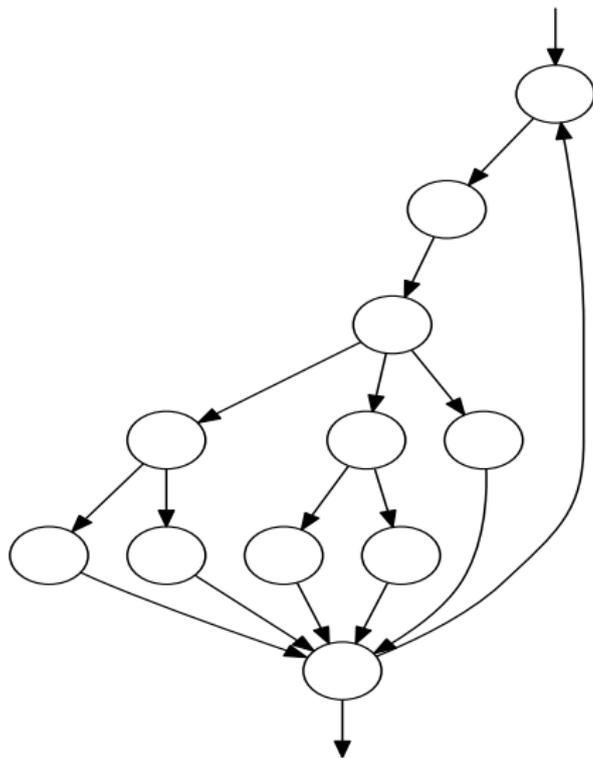
- Executar todas as instruções ?
- Executar todos os caminhos

*exhaustive path testing = exhaustive input testing*

## Exhaustive path testing

Os casos de testes executam todos os caminhos possíveis do fluxo de controle do programa. Assim o programa fica completamente testado

# Sobre o número de caminhos lógicos



# Sobre a completude

## Garantia sobre a especificação

Exhaustive path testing não pode descobrir a falta de caminhos necessários.

## Caminhos escolhidos

Exhaustive path testing não pode descobrir a falta de caminhos necessários.

## Data-sensitivity

Exhaustive path testing não vai necessariamente descobrir erros que dependem dos dados de entrada

# Métodos de seleção de testes

## Cobertura estrutural

- cobertura do grafo de fluxo de controle (CFG)
- cobertura do grafo de fluxo de dados (DFG)

## Teste por mutação

Seleção de casos de testes em função dos efeitos sobre o sistema.

- 1 Introdução
- 2 Cobertura do fluxo de controle
- 3 Cobertura do fluxo de dados

# Grafo de fluxo de controle (CFG)

## Definição

- 1 nó por instrução, 1 nó de entrada, 1 nó final de saída
- Para cada instrução do programa, o CFG tem um arco do nó da instrução ao nó da instrução seguinte (o do nó final)

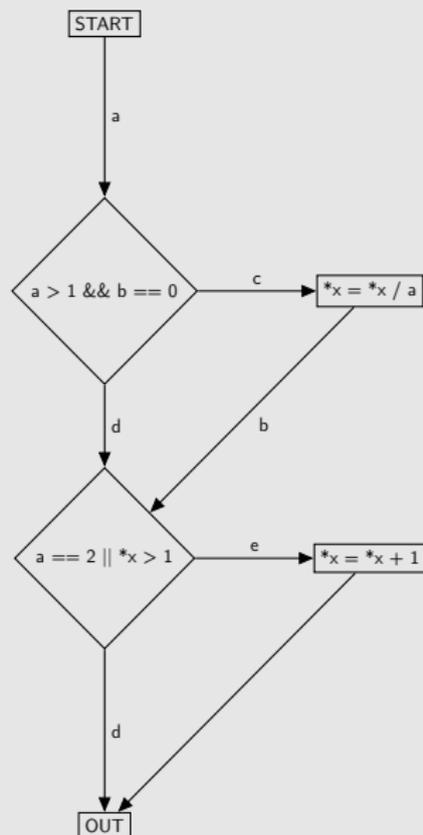
## Instruções condicionais

Seja a instrução `if (a < 3 && b > 4) { ... } else { ... }`

- `if` é uma **instrução condicional** / com ramificações
- `(a < 3 && b > 4)` é a **condição**
- tem duas **decisões** possíveis
  - ① `[ (a < 3 && b > 4), true ]`
  - ② `[ (a < 3 && b > 4), false ]`
- as **condições simples** são `a < 3` e `b > 4`

# Representação

```
void  
f(int a, int b, int * x)  
{  
    if (a > 1 && b == 0)  
        *x = *x / a;  
    if (a == 2 || *x > 1)  
        *x = *x + 1;  
}
```



# Critérios de cobertura

## Alguns critérios

Critério	Abreviação	Observação
Instruções	I	o mais fraco
Todas decisões (todo arcos)	D	teste de cada decisão
Todas condições simples	C	$\leq D$
Todas condições/decisões	DC	
Todas combinações de condições	MC	explosão combinatória
Todos caminhos	P	o mais forte <b>impossível em presença de laços</b>

# Critério I

## Objetivo

Executar as instruções do programa pelo menos uma vez.

## Exemplo

No exemplo, um único teste executando *ace* é suficiente.

## Observação

- se o `&&` devesse ser um `||` ?
- ou a segunda decisão `x > 0` ?
- é um error que `x` seja não mudado no caminho *abd* ?

# Critério D

## Objetivo

- Todas as decisões devem ter um resultado `false` e um `true` pelo menos uma vez (i.e. todas as ramificações serão executadas)
- Todo ponto de entrada deve ser executado

## Exemplo

- *ace* e *abd* ou *acd* e *abe*
- $\{\{a = 3, b = 0, x = 3\}, \{a = 2, b = 1, x = 1\}\}$

## Observação

- $D \succeq I$
- só 50% chance de executar *abd*

# Critério C

## Objetivo

- Toda condição numa decisão deve ter um resultado `false` e um `true` pelo menos uma vez
- Todo ponto de entrada deve ser executado

## Exemplo

- 4 condições :  $a > 1$ ,  $b == 0$ ,  $a == 2$ ,  $*x > 1$
- $ace + abd$  é suficiente
- $\{\{a = 1, b = 1, x = 1\}, \{a = 2, b = 0, x = 4\}\}$

## Observação

- **Geralmente**  $C \succeq D$

# Diferenças entre D e C

```
k = 0;  
do { k++; }  
while (k <= 50 && j + k < quest);
```

- **D** não precisa explorar o caso  $j + k \geq \text{quest}$
- **C** precisa testar  $k \leq 50$ ,  $k > 50$ ,  $j + k < \text{quest}$  e  $j + k \geq \text{quest}$

```
if (a && b) { /*...*/ } else { /*...*/ }
```

- **C** é preenchido com  $\{\{a = 1, b = 0\}, \{a = 0, b = 1\}\}$
- Tal jogo de teste não testa a ramificação `then`
- Nesse caso  $C \not\subseteq D$

# Critério DC

## Objetivo

- Todas as decisões devem ter um resultado `false` e um `true` pelo menos uma vez (i.e. todas as ramificações serão executadas)
- Todo ponto de entrada deve ser executado
- Todas condições devem ser executadas

## Exemplo

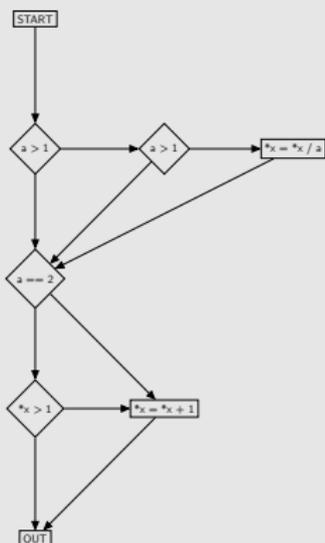
- 4 condições :  $a > 1$ ,  $b == 0$ ,  $a == 2$ ,  $*x > 1$
- 2 decisões
- $ace + abd$
- $\{\{a = 1, b = 1, x = 1\}, \{a = 2, b = 0, x = 4\}\}$

## Observação

- $DC \succeq D$  e  $DC \succeq C$

# Limitação de DC

## Código compilado



- Os critérios não levam em conta o fato que avaliar as expressões lógicas pode esconder / bloquear a avaliação de outras condições.
- Se uma expressão `&&` for falsa, não precisamos avaliar nenhuma condição subsequente.

# Critério MC

## Objetivo

- Todas as combinações de resultados de condições em cada decisão devem ser testadas
- Todo ponto de entrada deve ser executado

## Exemplo (compilado)

- Deve-se testar 8 combinações :  $(a > 1, b == 0)$ ,  $(a > 1, b! = 0)$ ,  $(a \leq 1, b == 0)$ ,  $(a \leq 1, b! = 0)$ ,  $(a == 2, x > 1)$ ,  $(a == 2, x \leq 1)$ ,  $(a! = 2, x > 1)$ ,  $(a! = 2, x \leq 1)$
- Testes :  $\{\{a = 2, b = 0, x = 4\}, \{a = 2, b = 1, x = 1\}, \{a = 1, b = 0, x = 2\}, \{a = 1, b = 1, x = 1\}\}$

## Observação

- $MC \succeq DC$
- Os testes não cobrem todos os caminhos

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)

# Exercício

```
#include <stdbool.h>
```

```
typedef enum {OK, KO} response;
```

```
response g(bool a, bool b, bool c) {  
    if (a && b)  
        return OK;  
    else if (c)  
        return OK;  
    else return KO;  
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)
  - ▶ L6 : (a, true), (a, false), (b, true), (b, false)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)
  - ▶ L6 : (a, true), (a, false), (b, true), (b, false)
  - ▶ L8 : (c, true), (c, false)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)
  - ▶ L6 : (a, true), (a, false), (b, true), (b, false)
  - ▶ L8 : (c, true), (c, false)
- Combinações de condições (MC)

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)
  - ▶ L6 : (a, true), (a, false), (b, true), (b, false)
  - ▶ L8 : (c, true), (c, false)
- Combinações de condições (MC)
  - ▶ L6 : ((a, true), (b, true)), ((a, true), (b, false)), ((a, false), (b, true)), ((a, false), (b, false))

# Exercício

```
#include <stdbool.h>

typedef enum {OK, KO} response;

response g(bool a, bool b, bool c) {
    if (a && b)
        return OK;
    else if (c)
        return OK;
    else return KO;
}
```

## Critério de cobertura

- Decisões (D)
  - ▶ L6 : ((a && b), true), ((a && b), false)
  - ▶ L8 : (c, true), (c, false)
- Condições simples (C)
  - ▶ L6 : (a, true), (a, false), (b, true), (b, false)
  - ▶ L8 : (c, true), (c, false)
- Combinações de condições (MC)
  - ▶ L6 : ((a, true), (b, true)), ((a, true), (b, false)), ((a, false), (b, true)), ((a, false), (b, false))
  - ▶ L8 : (c, true), (c, false)

# Critério MCDC

## Uso

- Aviônica (DO-178B)
- Potência entre DC e MC ... mas com um número **razoável** de testes

## Definição (Critério MCDC)

- Critério DC
- e os testes devem mostrar que cada condição atômica pode influenciar a decisão
- Para um condição  $C = a \wedge b$  os casos  $\{a = 1, b = 1\}$  e  $\{a = 1, b = 0\}$  demonstram que  $b$  sozinho pode influenciar a decisão global  $C$

## Observações

- Problema se condições atômicas ligadas
- O que fazer se a uma expressão lógica complexa é escondida numa expressão/chamada de função ?  
 $a = (d \ \&\& \ c) \ || \ (d \ != \ c); \ \text{if} \ (a) \ \{ \dots \} \ \text{else} \ \{ \dots \}$

# Hierarquia dos critérios

## Definição (Comparação de critérios)

$C_1 \succeq C_2$  ( $C_1$  é mais forte/subsume  $C_2$ ) se para todo programa  $P$  e todo conjunto de testes  $TS$ , se  $TS$  cobre  $C_1$  (em  $P$ ), então  $TS$  cobre  $C_2$  (em  $P$ ).

## Questão

Suponha que  $TS_2$  cobre  $C_2$  e acha uma falha de  $P$ , e  $TS_1$  cobre um critério  $C_1$  tal que  $C_1 \succeq C_2$ .

$TS_1$  acha necessariamente a mesma falha que  $TS_2$  ?

# Elementos sem cobertura

Alguns caminhos do CFG não podem ser seguidos.

Algumas instruções/ramificações podem ser "não cobríveis"

- `if (debug) { ... }`, com uma variável `debug` inicializada a falso.
- Teste de falha do hardware: `x = 0; if (x != 0) { problem (); }`;

- perca de tempo para cobrir um tal elemento
- diminui artificialmente o nível de cobertura atingido
- **Saber se um elemento de CFG é alcançável é indecidível**

- 1 Introdução
- 2 Cobertura do fluxo de controle
- 3 Cobertura do fluxo de dados

# Grafo de fluxo de dados

Os critérios de testes baseados no fluxo de dados selecionam os dados de testes em função das definições e dos usos das variáveis

## Definição

- Um variável é **definida** numa instrução si o valor dela é mudado (atribuição, declaração)
- Uma variável é **referenciada** se o valor da variável é usado
  - ▶ se esse valor é usado num predicado numa instrução de decisão, é um **p-uso**, senão é um **c-uso** (cálculo)

# Grafo de dados

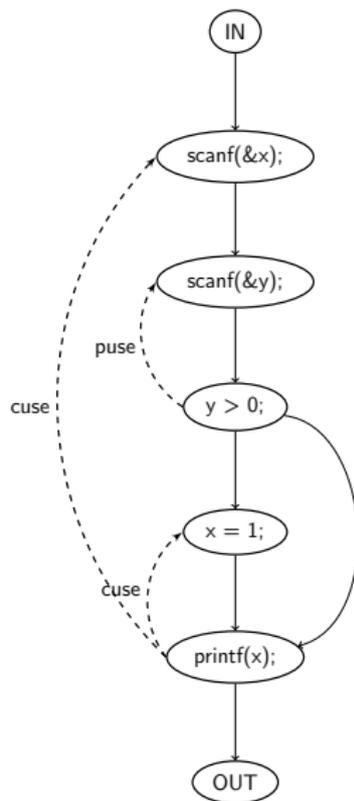
## Grafo de dependência de dados (DDG)

O DDG é um CFG com adição das informações que ligam definições e usos de variáveis.

## Definição

- $def(x)$ : instrução onde  $x$  é definida
- $use(x)$ : instrução onde  $x$  é usada
- **par def-use** para  $x$  ( $DU(x)$ ): par de instruções ( $D, U$ ) para  $x$  tais que exista pelo menos um caminho de execução passando por  $D$ , e depois por  $U$ , tal que  $x$  não seja redefinida entre  $D$  e  $U$
- **caminho def-use** (DU-path) para  $x$ : todo caminho passando por um par ( $D, U$ ) de  $x$  como explicado acima

# Exemplo de DDG



# Critérios de cobertura

## All defs, one use (all-def)

Para toda definição  $D$ , deve ter um teste ligando  $D$  a uma dos seus usos.

## All du, one path (all-uses)

- Cobrir pelo menos um caminho para todo par DU
- Variações: *all-c-uses*, *all-p-uses*

## All du, all paths (all-du-paths)

Cobrir todos os caminhos para todo par DU

# Aplicação de critérios estruturais

## Uso geral

- Critérios estruturais são usados ao nível unitário
- Pode ser usado em outros níveis

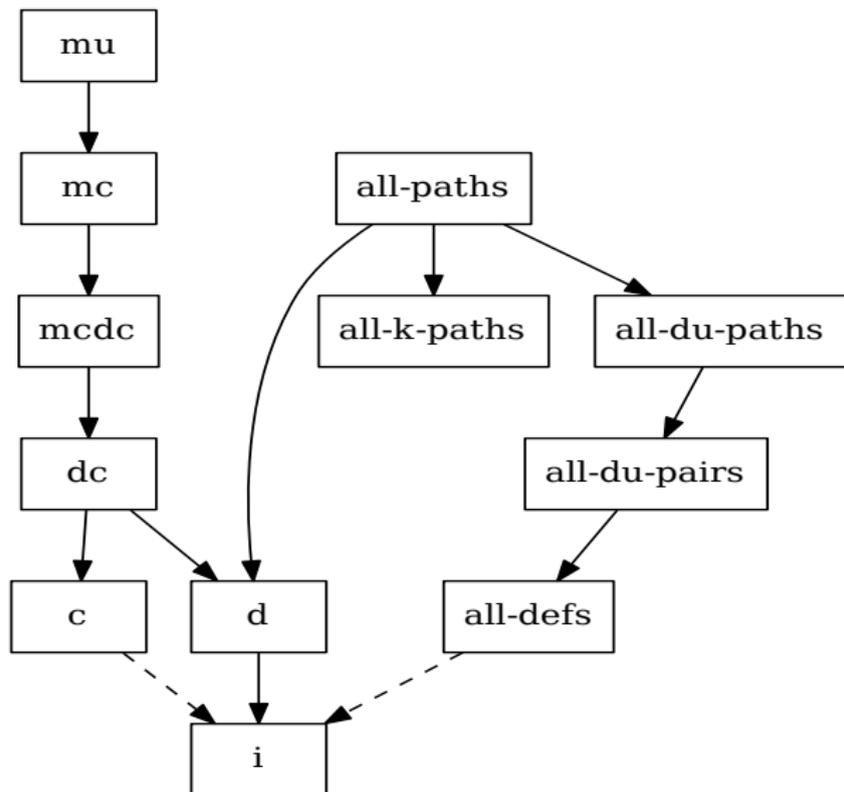
## Testes de integração / sistema

- Dificuldade = dimensionamento
- Cobrir algumas funções só (`main`)
- Usar critérios menos estritos para o código inteiro
  - ▶ Cobrir o grafo de chamadas de funções (nós, chamadas de função)
- Ter objetivos específicos: cobrir os usos de `lock`, `unlock`

## Avaliação

- Código crítico, teste de unidade : 100% du MCDC
- Código normal: 50% I, até 85% D

# Hierarquia dos critérios



# Cobertura estrutural e bugs verdadeiros

Só o teste estatístico dá formalmente uma medição de qualidade do software.

## Dados experimentais

- **Mutações:** resultado elevado com bom poder de detecção de bugs
- **Cobertura:** correlação inicial, e estagnação

# Ferramentas

## Cálculo de cobertura

- gcover
- coverlipse

## Mutações

- Proteum
- Lava

## Geração automática de dados de testes

- Pex C#
- PathCrawler, JavaPathFinder, Cute, DART, EXE

# Resumo

- 1 Introdução
- 2 Cobertura do fluxo de controle
- 3 Cobertura do fluxo de dados

# Referências

-  Paul Ammann and Jeff Offutt, *Introduction to software testing*, 1 ed., Cambridge University Press, New York, NY, USA, 2008.
-  Glenford J. Myers and Corey Sandler, *The art of software testing*, 3 ed., John Wiley & Sons, 2004.

Perguntas ?



<http://dimap.ufrn.br/~richard/dim0436>