


Laboratório de Engenharia de Software

Teste Estrutural 2

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015

Especificação



Laboratório de Engenharia de Software

- Objetivo desse módulo
 - Apresentar critérios de cobertura de caminhos e de análise do fluxo dos dados.
- Justificativa
 - O teste estrutural é utilizado tipicamente para testar módulos e, assim, verificar se os componentes elementares possuem o nível de qualidade necessário.
- Texto
 - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulos 13

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

2

Caminhos



- O que é um caminho em
 - um **método**?
 - um **grafo**?
 - um **diagrama de sequência**?
- Existem níveis de abstração para caminhos?
- Caminhos podem ser fragmentados?
- Do ponto de vista de teste, o que é um caminho?

Caminhos



- Um **caminho** corresponde a uma *determinada forma de percorrer* (sequência de execução de instruções) um conjunto de um ou mais **grafos da estrutura de código**
 - **caminhos completos** vão do início ao término da execução de uma função (ou de um programa)
 - chamadas a funções *encapsuladas* podem ser tratadas
 - ou como pseudo-instruções
 - ou como referências a outro grafo a ser percorrido (expansão do grafo)
 - chamadas a funções *externadas* podem ser tratadas como pseudo-instruções uma vez que podem ser testadas detalhadamente através da sua própria interface
 - em alguns casos pode ser interessante considerar somente um **fragmento de caminho** em um procedimento complexo
 - em outros casos consideram-se **caminhos mais abstratos** que envolvem várias funções, possivelmente de diferentes objetos
 - ex. diagrama de sequência

Critérios baseados em caminhos



- Os critérios baseados em caminhos selecionam um conjunto de caminhos
 - cada caminho nesse conjunto é um **caso de teste abstrato**
 - torna-se necessário examinar os pontos de decisão para que se possa estabelecer o **caso de teste semântico**
 - a valoração dos dados deve assegurar que se **execute exatamente o caminho** escolhido
 - ao aplicar as **condições de contorno** podem ser adicionados mais casos de teste em função das **diversas valorações** dependentes das condições que são exercitadas no caminho
 - o conjunto de caminhos valorados e respectivos oráculos forma a suíte de testes
- Para manter pequeno o custo do teste deseja-se
 - um conjunto pequeno de caminhos curtos
 - entretanto o conjunto de caminhos deve exercitar todo o código

Critério baseado em caminhos completos



- Para gerar de forma mecanizada os caminhos de uma função podem-se utilizar expressões algébricas
 - a expressão algébrica codifica o conjunto de todos os caminhos possíveis
 - em um **programa estruturado** a expressão será sempre uma **expressão regular**
 - expressões regulares admitem as seguintes composições:
 - elementos: $\{ A, B, C \}$;
 - concatenação: $A B C$;
 - seleção: $(A \mid B \mid C)$;
 - repetição: $n_1 - n_2 [A]$;
 - com base na expressão algébrica e no critério de valoração geram-se os caminhos a serem percorridos
 - em alguns casos as expressões podem ser aumentadas com chamadas (recursões, repetições)
 - deixam de ser regulares...

Cobertura de caminhos



- Cálculo da expressão algébrica dos caminhos a partir do código
 - numeram-se as linhas de código e percorre-se o código
 - externa-se o rótulo da instrução
 - se a instrução for início de controle de repetição externa-se '['
 - calcula-se o arrasto e externa-se o valor $arrasto + 1$
 - ao terminar o controle de repetição externa-se ']'
 - para escapar da repetição com um **break** use 'b' e com **return** use 't'
 - se for início de controle de seleção externa-se '('
 - ao atingir um **else** ou **else if** externa-se '|'
 - ao terminar o controle de repetição externa-se ')' ou '|ϕ|'
 - se for uma chamada de função encapsulada externa-se '[' seguido da identificação do método chamado, seguido de ']'
 - mais adiante adicionam-se a lista de caminhos da função encapsulada
 - se o nó a inserir já existe na expressão sendo criada, insere-se meramente o nó sem expandi-lo → recursão

Mai 2015

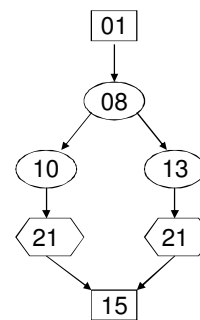
Arndt von Staa © LES/DI/PUC-Rio

7

Integração de expressões algébricas



```
01 void STR_String ::
02     STR_String( int idString )
03 {
04     #ifdef _DEBUG
05         EXC_ASSERT( idString > 0 ) ;
06     #endif
07     int idStr = ( idString & STR_ID ) ;
08     if ( ( idString & STR_MEM ) == STR_MEM )
09     {
10         BuildString( idStr ) ;
11     } else
12     {
13         BuildString( STR_NotImplemented & STR_ID ) ;
14     } // if
15 } // End of: Construct a string given an idString
```



Expressão algébrica:

$[01] :: \{ 08 (10 [21] | 13 [21]) 15 \}$

Caminhos são gerados segundo o critério cobertura de arestas

$[01] :: < 08 10 [21] 15 >$

$[01] :: < 08 13 [21] 15 >$

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

8

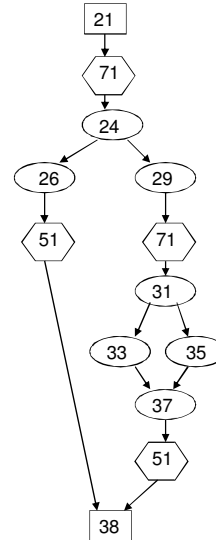
Grafo da estrutura do código: exemplo



```

21 void STR_String :: BuildString( int idString )
22 {
23     int i = FindStringElement( idString ) ;
24     if ( i >= 0 )
25     {
26         BuildString( vtStr[ i ].len , vtStr[ i ].str ) ;
27         return ;
28     } // if
29     char msg[ DIM_MSG ] ;
30     i = FindStringElement( STR_ErrorUndefinedId ) ;
31     if ( i >= 0 )
32     {
33         sprintf( msg , vtStr[ i ].str , idString ) ;
34     } else {
35         sprintf( msg , ILLEGAL_TABLE , idString ) ;
36     } /* if */
37     BuildString( strlen( msg ) , msg ) ;
38 } // End of: Build string for a given Id

```



Expressão algébrica deste método

[21] :: { [71] 24 (26 [51] | 29 [71] 31 (33 | 35) 37 [51]) 38 }

Caminhos

[21] :: < [71] 24 26 [51] 38 > ; [21] :: < [71] 24 29 [71] 31 33 37 [51] 38 > ;

[21] :: < [71] 24 29 [71] 31 35 37 [51] 38 >

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

9

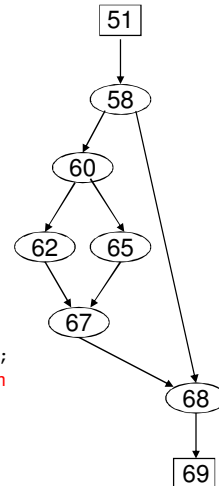
Grafo da estrutura do código: exemplo



```

51 void STR_String :: BuildString( int len ,
53                               char * pStr )
54 {
55     delete pCharString ;
56     length = len ;
57     pCharString = new char[ len + 1 ] ;
58     if ( len > 0 )
59     {
60         if ( pStr != NULL )
61         {
62             memcpy( pCharString , pStr , len ) ;
63         } else
64         {
65             memset( pCharString , STR_NULL_CHAR , len ) ;
66         } /* if */
67     } /* if */
68     pCharString[ len ] = 0 ;
69 } // End of: Build a string of a given size

```



Expressão algébrica deste método

[51] :: { 58 (60 (62 | 65) 67 | φ) 68 69 }

Caminhos

[51] :: < 58 60 62 67 68 69 > ; [51] :: < 58 60 65 67 68 69 > ; [51] :: < 58 68 69 >

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

10

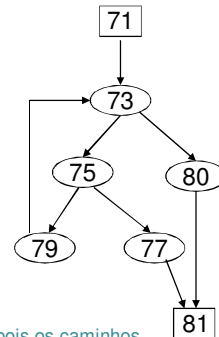
Grafo da estrutura do código: exemplo



```

71 int STR_String :: FindStringElement( int idStr )
72 {
73   for( int i = 0 ; i < NUM_STR_MEM ; i ++ )
74   {
75     if ( vtStr[ i ].idString == idStr )
76     {
77       return i ;
78     } // if
79   } // for
80   return -1 ;
81 } // End of: Find the index of the string element
      in the memory resident string table

```



Expressão algébrica deste método

$[71]::\{ 73 \ [2 \ (75 \ | \ 77 \ 81t) \ 79 \] \ 80 \ 81 \}$

Caminhos:

0 vezes: $[71]::\langle 71 \ 73 \ 80 \ 81 \rangle$; $[71]::\langle 71 \ 73 \ 75 \ 77 \ 81 \rangle$

1 vez: $[71]::\langle 71 \ 73 \ 75 \ 79 \ 80 \ 81 \rangle$; $[71]::\langle 71 \ 73 \ 75 \ 79 \ 75 \ 77 \ 81 \rangle$

2 vezes: $[71]::\langle 71 \ 73 \ 75 \ 79 \ 75 \ 79 \ 80 \ 81 \rangle$; $[71]::\langle 71 \ 73 \ 75 \ 79 \ 75 \ 77 \ 81 \rangle$

O 81 é repetido, pois os caminhos devem corresponder a arcos de uma árvore (no caso a "spanning tree"). O "t" sinaliza que o arco termina após o nó, no caso "81"

- por que isso? a aresta 77 – 81 viola a propriedade de ser um programa **perfeitamente** estruturado

Cobertura de caminhos



- Uma vez de posse da expressão algébrica
 - geram-se os caminhos combinando todas as alternativas
 - se não for tomado cuidado o número de caminhos pode crescer exponencialmente com o número de alternativas de cada decisão
 - **explosão combinatória**
 - para repetições geram-se as alternativas 0 , 1 , e arrasto+1 iterações
 - eliminam-se os caminhos não realizáveis
 - no caso de recursões limita-se o número de chamadas recursivas, em geral a uma.
 - se existe um limite especificado no programa, é necessário realizar tantas recursões quantas necessárias para atingir e passar desse limite.

Cobertura de caminhos



- A quantidade de casos de teste – **número de caminhos** – a serem gerados depende de
 - seleções realizadas (**if**, **switch**, **try / catch**)
 - número de iterações de repetições e de chamadas recursivas
 - formas de terminar (**break**, **return**, **throw**)
- Se a função for muito extensa isso pode levar a um conjunto muito grande de caso de teste
 - recomenda-se quebrar o código em várias funções encapsuladas
 - a seleção agora será cobertura de caminhos ao invés de a combinação de condições
 - entretanto, cada função deve ter um propósito bem definido
 - critério: se não é possível dar um nome que represente exatamente o propósito, a função não possui propósito bem definido
 - pode-se também realizar uma seleção eliminando caminhos similares do conjunto

Teste individual de [71]



Caminho: 71 [[2 73 (75 | 77 81t) 79]] 80 81

casos de teste abstratos e semânticos gerados:

- 71 73 80 81 0 vezes, não achou
 - tabela vazia
- 71 73 75 77 81 0 vezes, achou
 - tabela com ≥ 1 , achou primeiro
- 71 73 75 79 80 81 1 vez, não achou
 - tabela com 1 e não achou
- 71 73 75 79 75 77 81 1 vez, achou
 - tabela com ≥ 2 , achou segundo
- 71 73 75 79 75 79 80 81 2 vezes, não achou
 - tabela com 2, não achou
- 71 73 75 79 75 79 75 77 81 2 vezes, achou
 - tabela com ≥ 3 , achou terceiro

Exemplo: casos semânticos e valorados



- 71 73 80 81 → tabela vazia
 - { }, x
 - 71 73 75 77 81 → tabela com ≥ 1 , achou primeiro
 - { x }, x caso ==
 - { x , y }, x caso >
 - tabela com 1 e não achou
 - { x }, y
 - tabela com ≥ 2 , achou segundo
 - { x , y }, y
 - { x , y , z }, y
 - tabela com 2, não achou
 - { x , y }, z
 - tabela com ≥ 3 , achou terceiro
 - { x , y , z }, z
 - { x , y , z , w }, z
- 0 ciclos completos
- 1 ciclo completo
- 2 ciclos completos, arrasto + 1
- ao todo 9 casos de teste

Teste de funções encapsuladas



- Funções encapsuladas (no módulo: **static**, na classe: **private**, **protected**) não são acessíveis a partir da interface
 - no caso de **protected**: a partir de um objecto não herdeiro
- Como testá-las?

Teste de funções encapsuladas



- solução 1: criar um **módulo temporário** em que não são encapsuladas.
 - usando reflexão em Java pode-se ter acesso a métodos **private**
 - não gosto da ideia...
- Inconvenientes:
 - ser obrigado a redefinir a visibilidade dificulta retestar módulos já aprovados
 - teste de regressão
 - manutenção e evolução
 - a mudança de não encapsulado para encapsulado e vice-versa pode introduzir defeitos


Teste de funções encapsuladas



- solução 2: criar **funções de instrumentação** disponíveis quando o módulo é compilado para fins de teste.
- Inconveniente:
 - a variedade de formas de compilação pode introduzir defeitos
 - pode tornar-se difícil automatizar o reteste

Laboratório de Engenharia de Software

Teste de funções encapsuladas




- solução 3: incorporar as funções encapsuladas **estendendo os pontos em que são chamadas** em funções públicas
 - como se fossem macros
- Inconvenientes:
 - as suítes de teste podem tornar-se muito extensas devido à **explosão combinatória**
 - o total de casos de teste é da ordem de o “produtório” de alternativas
 - a chamada de um método corresponde ao número de alternativas encontradas nesse método
 - nem todas as condições são passíveis de teste a partir da função pública origem do teste
 - propriedades asseguradas pelo cliente para os argumentos em uma determinada chamada tornam impossível determinada condição resultar em `true` (ou `false`)

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
19

Laboratório de Engenharia de Software

Teste de funções encapsuladas



- solução 4: Criar **expressões de caminhos compostos** envolvendo as funções encapsuladas. Durante a criação de um caminho composto:
 - ao encontrar uma chamada a uma função encapsulada, escolher um dos caminhos desta função ainda não escolhidos em algum caminho composto anterior
- vantagens:
 - todos os caminhos de todas as funções serão exercitados pelo menos uma vez
 - como não se faz combinações de condições, o número total de caminhos torna-se “quase aditivo” ao invés de multiplicativo
- desvantagens:
 - como a escolha é qualquer, pode ocorrer que uma combinação defeituosa não seja escolhida

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
20

Caminhos elementares por método



- Possíveis caminhos (linhas) do método 21
1. 01 :: < 08 10 [21: 3; 4; 5] 15 >
 2. 01 :: < 08 13 [21: 3; 4; 5] 15 >
 3. 21 :: < [71: 9; 10; 11; 12 ; 13; 14] 24 26 [51:6; 8] 38 >
 4. 21 :: < [71: 13 ...] 24 29 [71: 12 ...] 31 33 37 [51:6; 8] 38 >
 5. 21 :: < [71: 9 ...] 24 29 [71: 9 ...] 31 35 37 [51:6; 8] 38 >
 6. 51 :: < 58 60 62 67 68 69 >
 7. 51 :: < 58 60 65 67 68 69 > **inviável**
 8. 51 :: < 58 68 69 >
 9. 71 :: < 73 80 81 >
 10. 71 :: < 73 75 77 81 >
 11. 71 :: < 73 75 79 73 80 81 >
 12. 71 :: < 73 75 79 73 75 77 81 >
 13. 71 :: < 73 75 79 73 75 79 73 75 79 73 80 81 >
 14. 71 :: < 73 75 79 73 75 79 73 75 79 73 75 77 81 >
- Nós dos grafos
- 7 é inviável
- Notação: [x: a ; b] – chama o método x nas linhas (primeira coluna) a ou b
- A escolha levou em conta observações relativas ao algoritmo composto. Ex. para percorrer um caminho em que encontrou o string precisa-se compor com um caminho de 71 que o encontre.

Exemplos de caminhos compostos



- Dois caminhos compostos
 - **vermelho**: 01 08 10 21 71 73 75 77 81 24 26 51 58 67 69 38 14 15
 - 08-10 → string deve estar na tabela em memória
 - 71...81 → id deve ser o primeiro da tabela
 - 24-26 → idString deve existir
 - 58-67 → tamanho do string deve ser == 0
 - **roxo**: 01 08 10 21 71 73 75 79 73 75 79 73 75 79 73 80 81 24 29 71 73 75 79 73 75 77 81 31 33 26 51 51 58 60 62 66 67 69 38 14 15
 - 08-10 → string deveria estar na tabela em memória
 - 71...80-81 → deve procurar e não acha em 3 (ou mais) iterações
 - 24-29 → consequência: idString não existe na tabela
 - 71...77-81 → procura e acha na 2ª. iteração o string `STR_ErrorUndef...`
 - 31-33 → consequência: idString `STR_ErrorUndefinedId` existe
 - 58-60 → tamanho do deve ser string > 0
 - 60-62 → tautologia: ponteiro para o string deve estar definido

Tautologia: Função lógica que sempre se converte em uma proposição verdadeira, sejam quais forem os valores assumidos por suas variáveis.

Composições



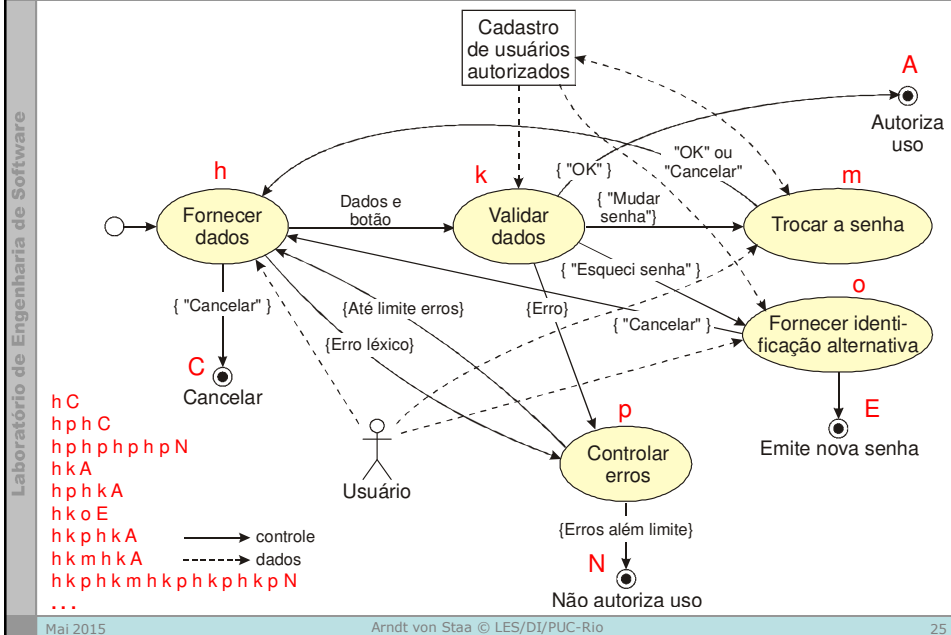
- Ao compor e ao término da composição deve-se verificar se os caminhos sugeridos são viáveis
 - como a composição é manual, caminhos inviáveis poderiam ser eliminada de saída
- Caminhos de métodos são marcados “usados” à medida que forem selecionados
 - a geração termina quando todos os caminhos dos métodos foram selecionados
 - caso um caminho composto seja não viável, os caminhos de métodos utilizados **somente neste caminho** composto devem voltar a ser marcados “não usados”

Composições



- Ao final do processo tem-se um conjunto de casos de teste que devem percorrer os caminhos estabelecidos
 - caso se observe resultados incorretos, ou caminhos incorretamente trafegados, devem ser feitas as correções na tabela e nos dados de teste
- Caso o número de caminhos seja muito grande, pode-se reduzir o número eliminando, se possível, caminhos que possuam subsequências repetidas
 - baseie o critério de decisão em análise de risco

Caminho em máquina de estados



Caminho em máquina de estados



- A lista de caminhos é completa?
 - que tal utilizar uma expressão para gerá-la
- Problema: máquinas de estado
 - não são árvores, são grafos genéricos
 - não correspondem a gramáticas regulares
 - mascaram repetições
 - implícitas na máquina de estados

Caminho em máquina de estados



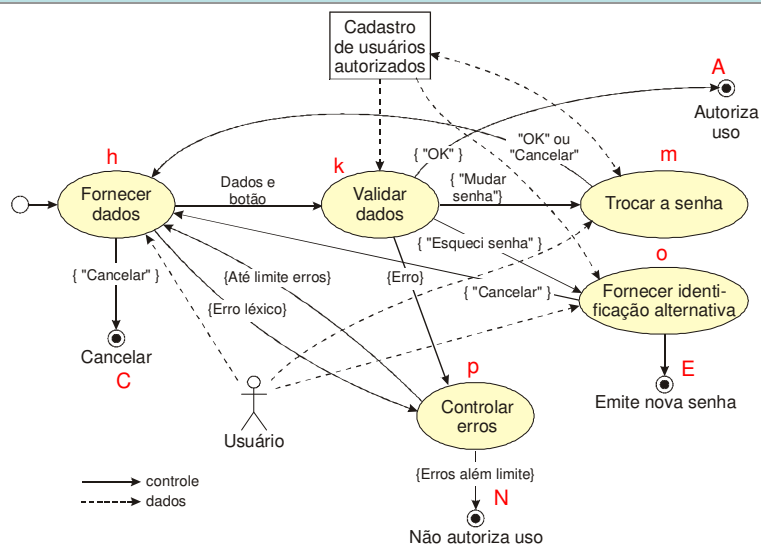
- Solução
 - criar uma expressão a partir do estado inicial
 - usar uma árvore (*spanning tree*) como estrutura auxiliar
 - cada estado tem uma seleção de sucessores
 - sucessores que levam a um estado já existente em um nível acima na árvore não são expandidos
 - gera-se uma "chamada" recursiva
 - os outros sucessores são expandidos
 - podem aparecer repetições de sub-árvores
 - a partir da árvore cria-se a expressão algébrica
 - conterá recursões para as referências a nós acima

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

27

Caminho em máquina de estados, rotular

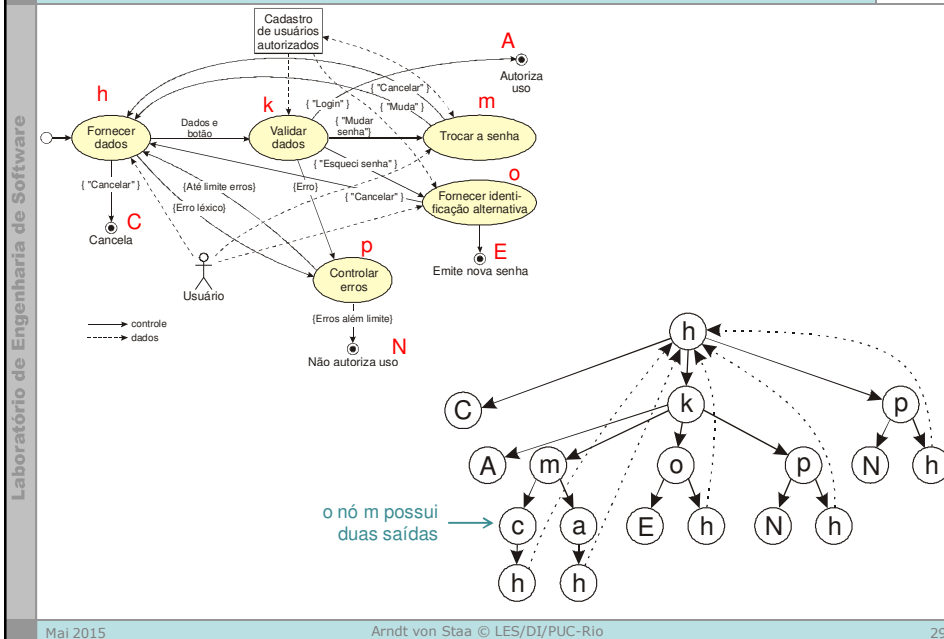


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

28

Caminho em máquina de estados, árvore

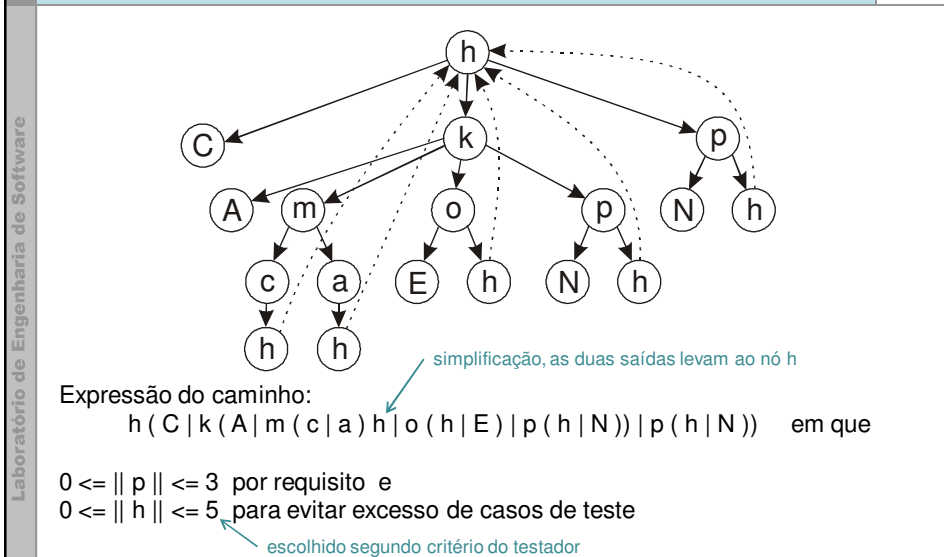


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

29

Caminho em máquina de estados, expressão



Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

30

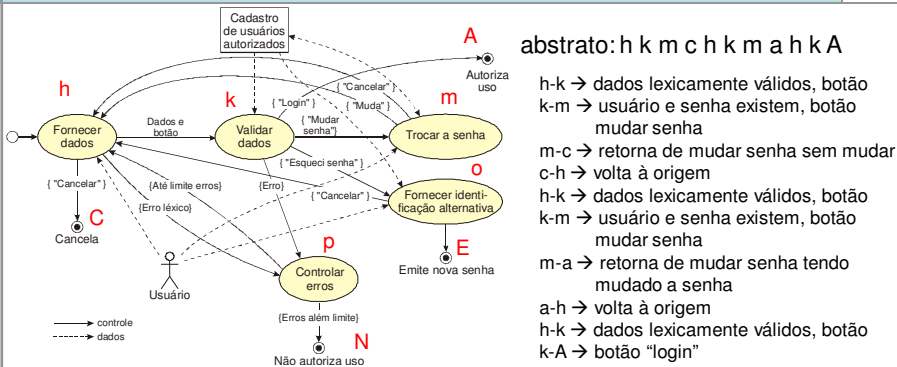
Caminhos abstratos



$h(C|k(A|m(c|a)h|o(h|E)|p(h|N))|p(h|N))$

- hC
- hkA
- hkmchC
- hkmahkA
- hkmahk mchC
- hkmchkmha kA
- hkma hohC
- hkma hohk
- hkma hohmahC
- hkma hohmah kA
- hkma hohmah kohC
- hkma hohmah koh kA
- hkma hohmah koh kphC
- hkma hohmah koh kph kA
- hkma hohmah kohE
- hkma hohmah kphC
- hkma hohmah kph kA
- hkma hohmah kph ohC
- hkmchohmch kph oh kA
- hkmchohmch kph ohE
- hkmchohmah kph phC
- hkmchohmah kph kA
- ...
- hkph kA
- hkph ph kA
- hkph ph ph kA
- hkph kph kph kph kA

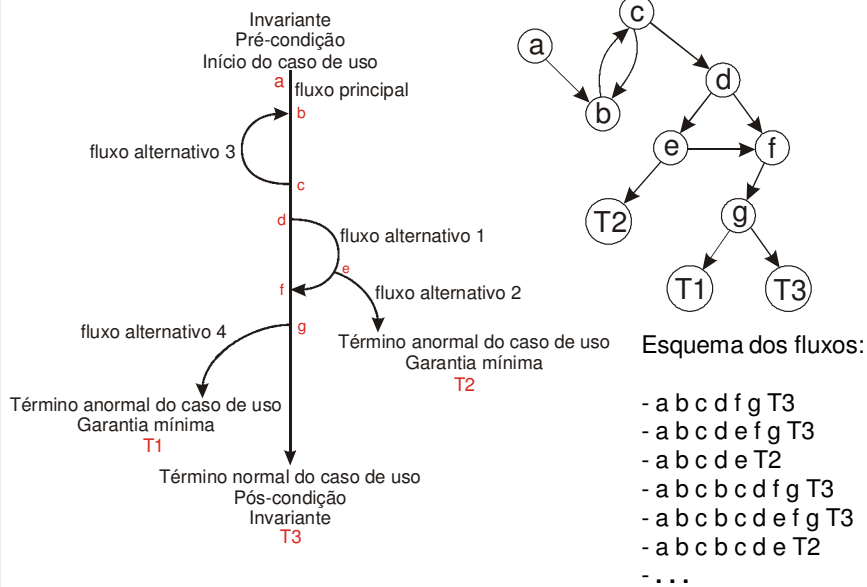
Caminhos semânticos



abstrato: h k p h p h k A

- h-k → dados lexicamente válidos, usuário existe, senha não vale, botão Login ou Mudar senha
- k-p → senha não vale
- p-h → 1º. erro
- h-p → dados lexicamente errados
- p-h → 2º. erro
- h-k → dados lexicamente válidos, senha vale botão Login
- k-A → botão "login"

Caminho em casos de uso



Teste baseado em fluxo de dados



- Princípio geral
 - **análise estática** do fluxo de dados
 - dado um caminho na estrutura do procedimento
 - examinam-se as sequências de operações de criação, uso, atribuição, e destruição que são realizadas sobre cada um dos dados
 - » *CRUD* – *Create, Retrieve, Update, Destroy*
 - alertando defeitos e potenciais defeitos
 - exemplos: uso antes de atribuir, atribuições sucessivas sem uso
 - muitos compiladores já fazem isso
 - » limitações óbvias: o problema é não computável no caso geral, logo nem todas as advertências podem ser geradas
 - » quando receberem uma advertência, corrijam o código
 - **análise dinâmica**
 - identificam-se caminhos que envolvem determinada variável segundo um critério de fluxo de dados
 - criam-se testes para percorrer estes caminhos

Teste baseado em fluxo de dados



- Dados são sujeitos às seguintes operações:
 - **D - definição**: atribuição, inicialização, construção, parâmetro recebido, ...
 - **Di** declaração com inicialização neutra, ex.
 - `int i = 0 ; char * p = NULL ;`
 - mas não: `c * p = new c () ;` nem parâmetro
 - **Dv** atribuição a um **espaço de dados apontado** por ponteiro
 - quando o ponteiro é declarado sem inicialização, ou é inicializado para `NULL`, o estado do espaço é * indefinido
 - **U - uso**:
 - **Uc** - computação: acesso, elemento de fórmula, argumento passando valor, ...
 - **Up** - predicado: uso em uma expressão lógica de seleção ou repetição
 - **L - liberação**: destruição, indefinição, tornar desnecessário, ..., tem como resultado "*", ver a seguir

Teste baseado em fluxo de dados



- Dados são sujeitos às seguintes "operações" (cont.)
 - *** estado indefinido** - não existe, destruído, ...
 - declaração sem inicialização, ex. `int i ;`
 - variável local após `return`
 - espaço apontado por ponteiro não inicializado
 - espaço apontado por ponteiro com valor **sabidamente NULL**
 - espaço apontado por ponteiro após destruição (`delete`)
 - **? estado** do espaço de dados apontado é **desconhecido**
 - o ponteiro pode ser `NULL` ou apontar para um espaço legal
 - `FILE * pArq = fopen(...) ;` pode não abrir o arquivo
 - `tpX * pX = (tpX *) malloc(sizeof(tpX)) ;` pode não alocar
 - `C * pC = dynamic_cast< C * >(pSuper) ;` C não herda de Super
 - quando **não** for **mais necessário** dispor de espaço apontado ou referenciado recomenda-se: em Java atribuir `null` ao ponteiro; em C++ efetuar `delete` e a seguir atribuir `NULL` ao ponteiro

Teste baseado em fluxo de dados



- **a = expressão**
 - todas as variáveis da expressão são do gênero **Uc**
 - a variável **a** se for valor é do gênero **D**
- **read(a, b, c, ...)**
 - todas as variáveis são do gênero **D**
- **write(a, b, c, ...)**
 - todas as variáveis são do gênero **Uc**
- **free(pX)** OU **delete obj**
 - o espaço apontado por **pX**, ou o objeto **obj** são do gênero **L**
- **f(int * parm)** - no cabeçalho de uma função, **parm != NULL**
 - o ponteiro **parm** é **D** e o espaço depende do contrato: *****, **Dv**, ou **?**
- ***parm = alguma-coisa**
 - o ponteiro **parm** é do gênero **Uc** e o respectivo espaço é **Dv**
- **f(parg)** - chamada passando ponteiro
 - o ponteiro **parg** é **Uc** e o espaço se **pArg != NULL** será **Uc** se não ***** ou **?**

Teste baseado em fluxo de dados



- **while (exp)**
 - todas as variáveis em **exp** são do gênero **Up**
- **for (a = b ; a < Lim ; a++)**
 - **a** é do gênero **D** em **a = b** e **a++**; **D** e **Uc** em **a++**; e **Up** em **a < Lim**
 - **b** é do gênero **Uc**
- **if (exp)**
 - todas as variáveis em **exp** são do gênero **Up**
- **switch (exp)**
 - todas as variáveis em **exp** são do gênero **Up**
- **OBS: exp não deve conter efeitos colaterais!!!**
 - **proibido:** **while (--i)** OU **while (scanf(...) != 0) ...**

Fluxo de dados: análise estática



antecessor	sucessor							
	D	Di	Dv	Uc	Up	L	*	?
	D	m	r	~	v	v	p	~
	Di	u	r	~	u	i	i	~
	Dv	~	~	q	v	v	p	v
	Uc	v	r	q	v	v	v	v
	Up	v	r	q	v	v	v	v
	L	v	r	v	e	e	v	v
	*	v	v	q	d	d	x	d
	?	~	~	u	f	f	v	u

- d erro: uso de variável não definida
- e erro: acesso a variável eliminada ou destruída
- f verificar se não **NULL**
- i declaração sem uso, ver
- m definição múltipla, ver
- p definição sem uso, ver
- q possível vazamento de memória, ver
- r erro: declaração repetida
- u usualmente OK, valor inicial pode não valer, ver
- v vale
- x se não estiver definida, é irrelevante indefinir a variável

~ Dv só se aplica a espaços apontados por ponteiros

O tratamento de ponteiros é simplório. Atribuições a ponteiros precisa ser controlado para evitar vazamento de memória.

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

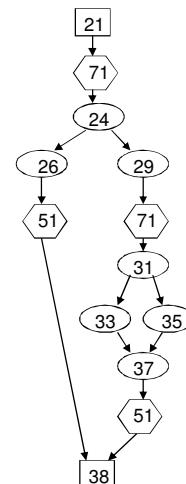
39

Fluxo de dados: caminhos completos



```

21 void STR_String :: BuildString( int idString )
22 {
23     int i = FindStringElement( idString ) ;
24     if ( i >= 0 )
25     {
26         BuildString( vtStr[ i ].len , vtStr[ i ].str ) ;
27         return ;
28     } // if
29     char msg[ DIM_MSG ] ;
30     i = FindStringElement( STR_ErrorUndefinedId ) ;
31     if ( i >= 0 )
32     {
33         sprintf( msg , vtStr[ i ].str , idString ) ;
34     } else {
35         sprintf( msg , ILLEGAL_TABLE , idString ) ;
36     } /* if */
37     BuildString( strlen( msg ) , msg ) ;
38 } // End of: Build string for a given Id
    
```



idString → 21 D 23 Uc 27 L; 21 D 23 Uc 33 Uc 38 L; 21 D 23 Uc 35 Uc 38 L

i → 23 D 24 Up 26 Uc 27 L; 23 D 24 Up 30 D 31 Up 33 Uc 38 L; 23 D 24 Up 30 D 31 Up 37 L

msg → 29 D 33 Uc 38 L; 29 D 35 Uc 38 L

[msg] → 29 * 33 Dv 37 Uc 38 L; 29 * 35 Dv 37 Uc 38 L

msg – o vetor como um todo
[msg] – o conteúdo do vetor

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

40

Fluxo de dados: definição de caminhos



- seja $n_0 - n_1 - n_2 - \dots - n_k - n_f$ um caminho existente no grafo de estrutura do nó n_0 a n_f com $k \geq 0$ nós intermediários relativo a uma variável v
- **c-uso global(v)**: ocorre $c\text{-uso}(v)$ no **nó**, mas não ocorre $def(v)$
- **caminho livre de definição $cld(v)$** : pode ou não ocorrer $def(v)$ em n_0 mas não ocorre $def(v)$ em qualquer outro nó do caminho
- **caminho simples**: não existem nós repetidos no caminho, exceto possivelmente n_0 e n_f *ciclos são percorridos no máximo 1 vez*
- **caminho-du(v)**: o caminho é simples e livre de definição, n_0 contém $def(v)$, e n_f contém $c\text{-uso}(v)$

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

41

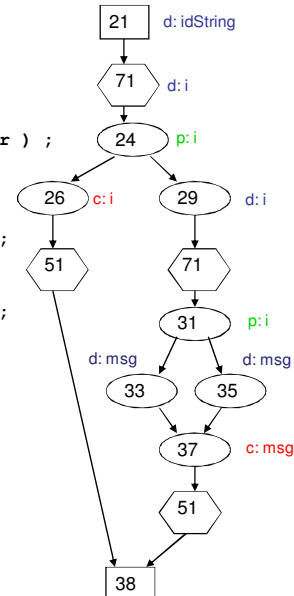
Fluxo de dados: caminhos



```

21 void STR_String :: BuildString( int idString )
22 {
23     int i = FindStringElement( idString ) ;
24     if ( i >= 0 )
25     {
26         BuildString( vtStr[ i ].len , vtStr[ i ].str ) ;
27         return ;
28     } // if
29     char msg[ DIM_MSG ] ;
30     i = FindStringElement( STR_ErrorUndefinedId ) ;
31     if ( i >= 0 )
32     {
33         sprintf( msg , vtStr[ i ].str , idString ) ;
34     } else {
35         sprintf( msg , ILLEGAL_TABLE , idString ) ;
36     } /* if */
37     BuildString( strlen( msg ) , msg ) ;
38 } // End of: Build string for a given Id

```



Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

42

Fluxo de dados: critérios



- Para todas as variáveis x no procedimento deve ser exercitado
 - **todas-defs**: um caminho para um de seus usos
 - **todos-p-usos**: um caminho para todos os seus p-usos
 - **todos-c-usos**: um caminho para todos os seus c-usos
 - **todos-c-usos-alguns-p-usos**
 - **todos-c-usos-alguns-p-usos**
 - **todos-usos**: um caminho para todos os seus c-usos e para todos os seus p-usos

Adaptado de (Delamaro et al, 2007)

Rapps, S.; Weyuker, E.J.; "Selecting Software Test Data Using Data Flow Information"; *IEEE Transactions on Software Engineering* SE-11(4); IEEE Computer Society; 1985; pags 367-378

Fluxo de dados, comentários finais



- Conforme discutido no início, pode tornar-se necessário adotar a solução 3: incorporar as funções encapsuladas estendendo os pontos em que são chamadas em funções públicas
- Tende a gerar um número grande de fragmentos de caminhos
- Pode gerar diversos caminhos não executáveis
 - ex. o caso `pStringParm == NULL` na linha 60

Laboratório de Engenharia de Software

Referências bibliográficas



- Delamaro, M.E.; Maldonado, J.C.; Jino, M.; *Introdução ao Teste de Software*; Rio de Janeiro, RJ: Elsevier / Campus; 2007
- Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008

Mai 2015Arndt von Staa © LES/DI/PUC-Rio45

Laboratório de Engenharia de Software

FIM



Mai 2015Arndt von Staa © LES/DI/PUC-Rio46