Laboratório de Engenharia de Software

Aula 24
Argumentação da Corretude

Alessandro Garcia
LES/DI/PUC-Rio
Maio 2009

Laboratório de Engenharia de Software

Argumentação e outras técnicas

- Inspeção e Teste vs. Argumentação
- Medição vs. Argumentação

- Argumentação:
  - visa maximizar a redação de algoritmos/funções que são corretos antes de teste/inspeção/medição

Especificação

- Objetivos dessa aula
  - Motivar a técnica de desenvolvimento dirigida por argumentação
  - Revisitar conceitos relacionados com assertivas de entrada, de saída e estruturais no contexto de argumentação da corretude
  - Estabelecer uma notação rigorosa para assertivas
  - Mostrar como se argumenta corretude na presença de chamadas, seqüências de código e seleções
- Referência básica:
  - Capítulo 13
- Referência complementar
  - Hall, A.; "Seven Myths of Formal Methods"; *IEEE Software* 7(5); Los Alamitos, CA: IEEE Computer Society; 1990; pags 11-19

Sumário

- Assertivas e argumentação
- Argumentação de um fragmento de código simples
- Processo: argumentação e decomposição
- Argumentação de chamadas
- Argumentação de seqüências de fragmentos de código
- Exemplo
- Argumentação de seleções

O que é argumentar a corretude?

- Entretanto, programadores tentam de alguma forma explicar a corretude de seus programas
  - certamente procuram explicar que os seus programas se comportam em conformidade com as suas crenças
- O **PROBLEMA** está:
  - na forma de raciocinar que se baseia em crenças não fundamentadas
- **SOLUÇÃO:** Argumentar a corretude é convencer de uma forma sistemática a si próprio e a outros que um artefato corresponde à sua especificação
  - com base em técnicas de prova da corretude
  - para poder argumentar, precisamos dispor de uma especificação suficientemente formal

O que é argumentar a corretude?

- Definição: demonstrar que um fragmento de código que:
  1. caso as AEs sejam válidas antes da execução do fragmento, e caso chegue-se às *ASs*, estas últimas serão sempre válidas
  2. sua execução sempre chegará às ASs
- O objetivo é: ser uma técnica de verificação da corretude
  - leve (requer pouco esforço e é menos caro do que prova formal)
  - eficaz (reduz sensivelmente o número de defeitos remanescentes)
  - que possa ser aplicada rotineira e economicamente ao desenvolver módulos e diagnostificar faltas em módulos
- A argumentação não é uma **prova formal completa**
  - pois não consideramos todas as possíveis assertivas
    - provas formais são caras, são mais usadas onde teste do código é difícil ou em sistemas altamente críticos
- Quanto maior for o dano nominal
  - menor deve ser a probabilidade do programa conter defeitos
  - maior deverá ser o rigor da argumentação
    - no caso limite teremos que efetuar provas formais completas

Limitação

- A argumentação não é uma prova formal
  - conseqüentemente é possível que se argumente que um algoritmo está supostamente "correto", quando, na realidade, ainda contém defeitos
- Como provas formais são caras, são mais usadas onde teste do código é difícil ou em sistemas altamente críticos
- Cabe observar que mesmo provas formais podem falhar, em particular por serem realizadas por humanos
  - em [Gerhart, S.L.; Yelowitz, L. "Observations of fallibility in applications of modern programming methodologies"; *IEEE Transactions on Software Engineering 2(9)*; setembro 1976, pags 195-207] são discutidos 10 algoritmos provados corretos e que não resistiram a um teste ou mesmo a uma inspeção

Por que argumentar a corretude?

- Precisamos saber argumentar
  - ao desenvolver
  - ao diagnosticar (e depurar para encontrar a fonte de) uma falha
  - ao gerar casos de teste de forma sistemática
- Quanto maior for o dano nominal
  - menor deve ser a probabilidade do programa conter defeitos capazes de provocar o dano em questão
  - maior deverá ser o rigor da argumentação
    - no caso limite teremos que efetuar provas formais completas

Avaliação de assertivas de entrada

Critérios de avaliação de assertivas de entrada

- devem figurar nas assertivas de entrada todos os dados, estados e recursos usados antes de redefinir (alterar)
  - *no caso de contratos de funções*, considere a interface conceitual
  - *no caso de fragmentos de código* considere os elementos usados no fragmento
- Sempre que as fontes de dados não forem confiáveis, é necessário verificar a validade dos dados por meio de código
  - fornecidos por um usuário humano
  - recebidos através da rede
  - lidos de arquivos com procedência não confiável

Avaliação de assertivas de saída

Critérios de avaliação de assertivas de saída

- devem figurar todos os dados, estados e recursos alterados ou criados durante o processamento do fragmento que antecede a assertiva
  - variáveis locais ao fragmento não precisam figurar nas AEs e ASs
- devem relacionar todos os recursos que devem ser desalocados ou liberados ao sair da função
  - ponteiros para memória dinâmica
  - recursos requisitados ao sistema operacional, ex. arquivos
- a assertiva de saída da função deve relacionar os resultados considerando todas as formas de término da execução
  - return no meio do corpo
  - para C++ , Java , C# : **`throw`**

Avaliação de assertivas estruturais

Critérios de avaliação de assertivas estruturais

- as assertivas estruturais devem relacionar todos os atributos e referências (ponteiros) contidos na estrutura
  - é permitido definir variáveis utilizadas somente pelas assertivas
    - ex. número de elementos de uma lista

O que precisamos?

- Para podermos argumentar a corretude precisamos de uma especificação suficientemente formal
  - modelos e/ou programas
  - assertivas de entrada
  - assertivas de saída
  - assertivas estruturais
- O simples fato de redigir as assertivas já leva a código de qualidade melhor

Relembrando... O que são assertivas?

- Assertivas são relações (expressões lógicas) envolvendo dados e estados manipulados
- São definidas em vários níveis de abstração
  - **programas**
    - devem estar satisfeitas para os dados persistentes (arquivos)
  - **módulos (ou classes)**
    - devem estar satisfeitas ao entrar e ao retornar de todas funções
    - assertivas invariantes:
      - assertivas estruturais: corretude das instâncias de estruturas de dados
      - também: certos limites inferiores e superiores associados com certos atributos de classes
  - **funções**
    - devem estar satisfeitas em determinados pontos do corpo da função
    - usualmente assertivas de entrada e assertivas de saída
      - pré e pós condições
      - na definição das pós-condições, assume-se que as pré-condições foram satisfeitas

Terminologia

- Contratos e assertivas são similares
  - ambos são expressões lógicas estabelecendo o que deve ser verdade em determinado ponto caso o programa esteja operando corretamente
- Contratos relacionam somente os dados e propriedades semânticas visíveis na interface e que devem ser conhecidas pelos clientes
  - estabelecem o compromisso entre os clientes e os servidores
- Assertivas relacionam os dados e propriedades semânticas da implementação
  - assertivas são encapsuladas
- Conseqüentemente um contrato é uma assertiva, mas nem toda assertiva é um contrato

É necessário...

- Estabelecer uma notação rigorosa para assertivas, com o objetivo de apoiar:
  - a argumentação da corretude de programas
  - a implementação, possivelmente através de geração, de assertivas executáveis
- Notação Rigorosa
  - Operadores lógicos
  - Conjuntos
  - Quantificadores
  - Exemplos de assertiva estrutural: lista e árvore n-ária
  - Funções

Implicação

1) **se** *premissa* **então** *conseqüente*
2) *premissa* ⇒ *conseqüente*

   se *premissa* for **`verdadeira`**, e a *conseqüente* também for **`verdadeira`**, a expressão será **`verdadeira`**

   se *premissa* for **`verdadeira`**, e *conseqüente* for **`falsa`**, a expressão será **`falsa`**

   se *premissa* for **`falsa`**, a expressão passa a ser irrelevante

- não existe **`else`**, deve-se redigir a implicação com a premissa negada, exemplo

  ! *premissa* ⇒ *conseqüente*

Exemplo:

```
pLista->numElem == 0 ⇒ ( pLista->pOrg == NULL ) &&
                       ( pLista->pFim == NULL ) &&
                       ( pLista->pCorr == NULL )
```

Conjunção, expressão "e"

Laboratório de Engenharia de Software

1) *condição*$_1$ && *condição*$_2$ && ... && *condição*$_n$
2) *condição*$_1$ $\wedge$ *condição*$_2$ $\wedge$ ... $\wedge$ *condição*$_n$
3) *condição*$_1$, *condição*$_2$, ... , *condição*$_n$
4) *condição*$_1$
   condição$_2$
   ...
   condição$_n$

- as quatro expressões são equivalentes
- para que a expressão seja **`verdadeira`**, todas as condições *1*, *2*, ... *n* têm que ser **`verdadeira`**s

Disjunção "ou", "xor" e Negação

Laboratório de Engenharia de Software

- **Disjunção (convencional):**

1) *condição*$_1$ || *condição*$_2$ || ... || *condição*$_n$
2) *condição*$_1$ $\vee$ *condição*$_2$ $\vee$ ... $\vee$ *condição*$_n$

- as duas formas são equivalentes
- para que a expressão seja **`verdadeira`** uma ou mais das condições *1*, *2*, ... *n* têm que ser **`verdadeira`**s

- **Disjunção exclusiva**

1) *condição*$_1$ **`xor`** *condição*$_2$ **`xor`** ... **`xor`** *condição*$_n$

- para que a expressão seja **`verdadeira`** exatamente uma das condições *1*, *2*, ... *n* têm que ser **`verdadeira`**s
- exatamente: uma e somente uma

- **Negação**

1) ! *Condição*

- se *Condição* for **`verdadeira`**, a expressão será **`falsa`**
- se *Condição* for **`falsa`**, a expressão será **`verdadeira`**

Conjuntos

- { A1 , A2 , ... , An }
- TipoEstrutura{ refEstrutura } /* notação similar a C */
  - é o conjunto de todos os elementos que constituem a instância de estrutura
  - sugestão:
    - referencie sempre a cabeça da estrutura

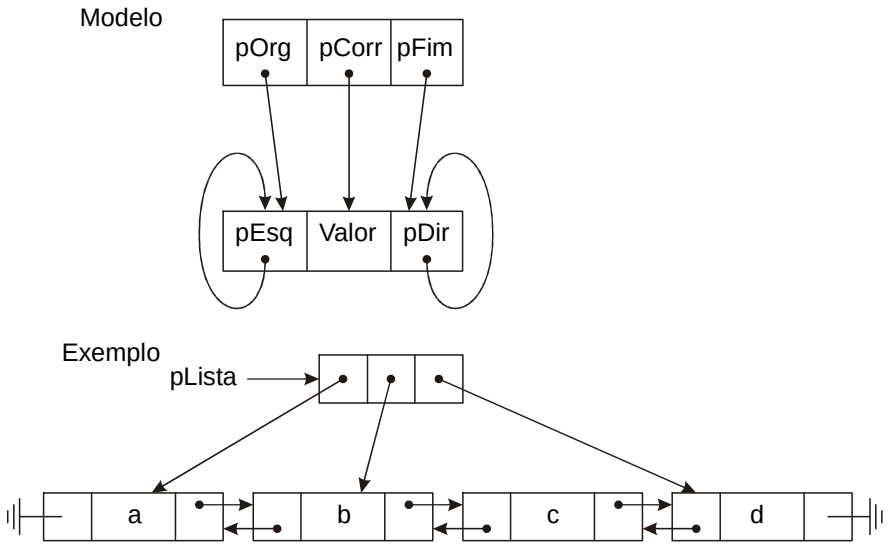
determina o tipo da estrutura

∀ pElem ∈ Lista{ pLista } : condição1 , condição2 , ...

Quantificadores

- ParaTodos , ou ∀
- Pertence , ou ∈
- Vale , ou :
- Existe , ou ∃
- TalQue , ou |
- Exemplos:
  - ∀ pElem ∈ Lista{ pLista } : condição1 , condição2 , ...
    - para todos os elementos *pElem* pertencentes à lista *pLista* valem as condições: *condição1* , *condição2* , ...
    - use *pElem* caso se trate de um ponteiro para o elemento ou *refElem* caso se trate de uma referência explícita
  - ∀ i | (0 <= i < n) : condição1 , condição2 , ...
    - para todos os i tal que i esteja no intervalo [0 .. n) valem as condições: *condição1* , *condição2* , ...
  - ∃ pElem ∈ Lista{ pLista } | condição1 && condição2
    - existe um elemento *pElem*, pertencente à lista *pLista*, tal que as condições *condição1* e *condição2* sejam verdadeiras

Exemplo assertiva estrutural: lista



Exemplo assertiva estrutural: lista

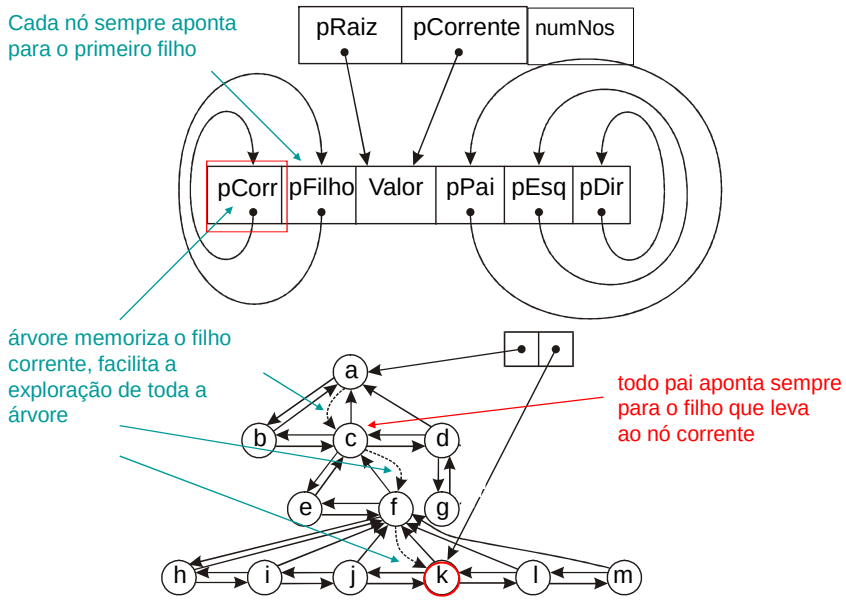
pLista->numElem == 0 ⇒
(pLista->pOrg == NULL) &&
(pLista->pFim == NULL) &&
(pLista->pCorr == NULL)

(pLista->pOrg == NULL) || (pLista->pFim == NULL) ||
(pLista->pCorr == NULL) ⇒ pLista->numElem == 0

pLista->numElem != 0 ⇒
((pLista->pOrg != NULL) && (pLista->pOrg->pEsq == NULL)) &&
((pLista->pFim != NULL) && (pLista->pFim->pDir == NULL)) &&
(∃ Elem ∈ Lista{ pLista } : pLista->pCorr == pElem)

∀ Elem ∈ Lista{ pLista } : /* calculado percorrendo a lista */
pElem->pEsq == NULL ⇒ pLista->pOrg == pElem
pElem->pEsq != NULL ⇒ pElem->pEsq->pDir == pElem
pElem->pDir == NULL ⇒ pLista->pFim == pElem
pElem->pDir != NULL ⇒ pElem->pDir->pEsq == pElem

Exemplo assertiva estrutural: árvore n-ária



Exemplo assertiva estrutural: árvore

Cabeça da árvore *pArv

- Se a árvore está vazia: pArv->pRaiz == NULL e
  pArv->pCorrente == NULL
- Se a árvore não está vazia:
  - pArv->pRaiz aponta para a raiz da árvore
  - pArv->pCorrente aponta para um dos nós da árvore

Outra redação

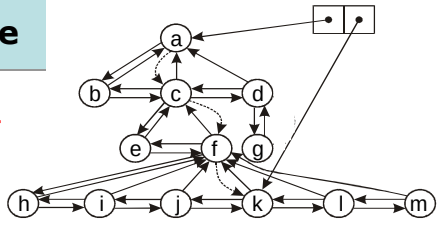
pArv->numNos == 0 ⇒ pArv->pRaiz == NULL &&
pArv->pCorrente == NULL

pArv->numNos != 0 ⇒ pArv->pRaiz->pPai == NULL &&
∃ pNo ∈ Arvore{ pArv } | pNo == pArv->pCorrente

Exemplo assertiva: árvore

Laboratório de Engenharia de Software

O que está errado?



Corpo da árvore:

∀ pNo ∈ Arvore{ pArv } : /* calculado percorrendo a árvore */

pNo->pPai == NULL ⇒ pArv->pRaiz == pNo &&
pNo->pEsq == NULL &&
pNo->pDir == NULL
/* pNo é raiz */

pNo->pPai != NULL ⇒ ∃ Elem ∈ Lista{ pNo->pPai->ListaFilho } |
pElem == pNo
/* pNo deve estar na lista de filhos do pai */

pNo->pFilho == NULL ⇒ pNo->pCorr == NULL
/* se pNo não tem filhos, não indica caminho/nó corrente */

pNo->pFilho != NULL ⇒
(∀ pFil ∈ Lista{ pNo->pFilho } : pFil->pPai == pNo) &&
(∃ pFil ∈ Lista{ pNo->pFilho } | pNo->pCorr == pFil)

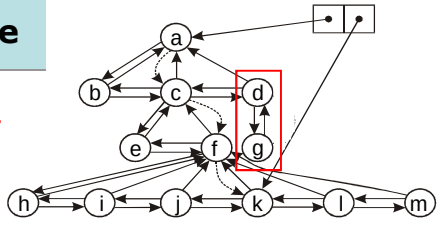
/* todos os filhos pFil de pNo apontam para o pai pNo */

/* existe pelo menos um filho que aponta para o caminho/no corrente */

Exemplo assertiva: árvore

Laboratório de Engenharia de Software

O que está errado?



Corpo da árvore:

∀ pNo ∈ Arvore{ pArv } : /* calculado percorrendo a árvore */

pNo->pPai == NULL ⇒ pArv->pRaiz == pNo &&
pNo->pEsq == NULL &&
pNo->pDir == NULL
/* pNo é raiz */

pNo->pPai != NULL ⇒ ∃ Elem ∈ Lista{ pNo->pPai->ListaFilho } |
pElem == pNo
/* pNo deve estar na lista de filhos do pai */

pNo->pFilho == NULL ⇒ pNo->pCorr == NULL
/* se pNo não tem filhos, não indica caminho/nó corrente */

pNo->pFilho != NULL ⇒
(∀ pFil ∈ Lista{ pNo->pFilho } : pFil->pPai == pNo) &&
~~(∃ pFil ∈ Lista{ pNo->pFilho } | pNo->pCorr == pFil)~~

/* todos os filhos pFil de pNo apontam para o pai pNo */

/* existe pelo menos um filho que aponta para o caminho/no corrente */

Maio 2009 / 38

Assertivas e argumentação

- Argumentar a corretude de um fragmento de código é mostrar que
  - dada a validade das assertivas de entrada e estrutural antes de iniciar a execução do fragmento
    - note que nada pode ser concluído a partir de uma premissa falsa
  - a execução do fragmento implica a validade das assertivas de saída (e estrutural) ao terminar a execução do fragmento
  - devem ser levados em conta todas as formas de se terminar a execução do fragmento
    - atingir o final do fragmento
    - **`break`**
    - **`return`**
    - **`throw`**
  - e, ainda, o fragmento de código sempre terminará de executar

Formato de ASs de uma função:
condição1, condição 2, etc...
CondRet == condRetNormal1 ⇒ condição1.1, ...
CondRet == condRetNormal2 ⇒ condição2.1, ...
CondRet == condRetExcep1 ⇒ condiçãoE1.1, ...
CondRet == condRetExcep2 ⇒ condiçãoE2.1, ...
CondRet ∈ {condRetNormal1, ...}

possíveis valores das cond. retorno deveriam ser válidos para qq resultado

Argumentação de um passo de execução

- A notação: $AE \oplus P \Rightarrow AS$ significa:
  - A expressão resulta em **`TRUE`** se for válida a assertiva de entrada *AE* e a execução do fragmento de código *P* implicar a validade da assertiva de saída *AS*
- Como ler a expressão: $A \oplus P \Rightarrow B$
  - ⊕ significa: a execução de (um fragmento de código)
  - dada a validade de *A* a execução do fragmento *P* implica a validade de *B*

Exemplo: Fragmento P pode ser...

1. corpo da função: *AbrirArquivo(char * NomeArq , tpModo Modo) => FILE* pArq , tpCondRet CondRet*

2. qualquer bloco de código em *AbrirArquivo*

Certas assertivas estruturais podem ser inválidas em partes do código

Argumentação de um passo de execução

- *AE* contém tudo que é somente utilizado e tudo que será atualizado ou destruído (i.e. o que é usado e alterado)
  - nada que será criado, nem o que não será utilizado
- *AS* contém tudo que foi atualizado, criado ou destruído
  - nada que foi somente utilizado por *P*, ou sequer aparece em *P*
  - não contém nada que seja local a *P*, ou que somente interesse no âmbito de *P:* deixa de ter interesse além do contexto de *P*
  - não esquecer de checar condições de retorno
  - os itens de *AE* atualizados terão sido substituídos em *AS* pelas alterações realizadas por *P*
  - para referenciar na saída os dados de entrada utilizaremos: **`entrada:nome`** (outras notações: old:nome; nome, @nome)
    - exemplo: checar que uma determinada lista foi modificada por P: Lista{entrada:pMinhaLista} ≠ Lista{pMinhaLista}

Laboratório de Engenharia de Software

Exemplo de um passo de execução

abstração raiz

AE:

NomeArq – o nome do arquivo tal como fornecido pelo usuário, string ASCII com no máximo dimNomeArq caracteres

Modo – como deve ser aberto o arquivo, ver tpModo

*AbrirArquivo(char * NomeArq , tpModo Modo) =>*
FILE pArq , tpCondRet CondRet*

AS:

Não muda o valor, mas muda o que sabemos a respeito da sua qualidade

CondRet == OK

NomeArq – é sintaticamente válido, o arquivo existe e pode ser acessado por este programa

pArq é o ponteiro para o descritor do arquivo aberto

CondRet != OK

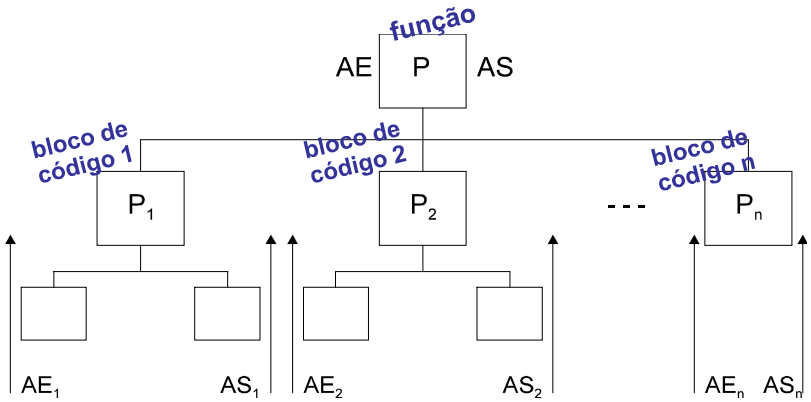
pArq == NULL

CondRet informa o detalhe do erro, ver tpCondRet

Laboratório de Engenharia de Software

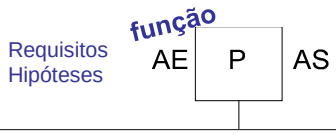
Argumentação e decomposição sucessiva

Do ponto de vista de programação, cada função pode ser vista como uma *abstração raiz*...



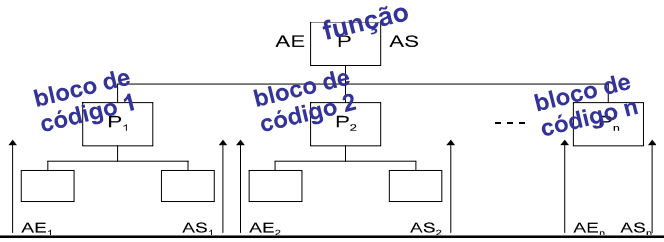
Argumentação e decomposição sucessiva

- Inicialmente especifique
  - os requisitos e as hipóteses da abstração raiz
  - as assertivas de entrada e saída da abstração raiz
  - se for o caso, referencie as assertivas estruturais que governam a abstração raiz
    - P.e. "AE: Valem as assertivas estruturais da árvore binária"
  - reveja cuidadosamente essa especificação
    - critérios de qualidade (p.e. aula 10): completude, não-ambigüidade, concisão, rastreabilidade, etc...



Argumentação e decomposição sucessiva

- A especificação da abstração raiz tendo sido aceita
  1. proponha um conjunto de solução
  2. verifique a qualidade estrutural deste conjunto
     - Não tem replicação de código e, se possível, faz uso de esquemas de algoritmos: iterators, template methods, strategy, etc...
  3. defina outras assertivas estruturais (se necessário) que governam as estruturas de dados utilizadas
  4. verifique a completeza e consistência de ASn e AEn+1 (para todo n) e corrija até estar convencido que a seqüência está boa



Argumentação e decomposição sucessiva

- A especificação da abstração raiz tendo sido aceita (cont.)
  5. dependendo da complexidade e do rigor necessário
     - refine assertivas de entrada e saída de cada elemento do conjunto de solução
     - verifique a completeza e a consistência de todas essas especificações
  6. argumente a corretude do conjunto de solução
     - leve em consideração chamadas, seleções, etc...
  7. caso o conjunto de solução não seja satisfatório
     - proponha outro conjunto
  8. repita a partir de 1 até que o conjunto de solução seja satisfatório

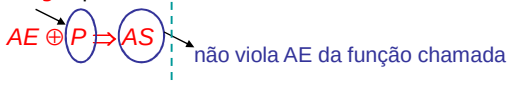
Argumentação de chamadas

A argumentação de chamadas de funções ou métodos é particionada em três etapas:

1. Determinar a associação entre a chamada e a implementação da função que será ativada
   - funções que podem ser associadas a uma mesma chamada formam uma família
     - ponteiro para função
     - redefinição, no caso de herança
   - definir AEs e ASs comum à família
     - a assertiva de entrada da família deve estar consistente com a assertiva de entrada de cada um dos *membros* dessa família
       - Obs.: detalhes de assertivas de membros específicos não podem conflitar com as assertivas da família
     - a assertiva de saída de cada *membro* da família deve estar consistente com a assertiva de saída da família
       - Idem Obs acima

Argumentação de chamadas

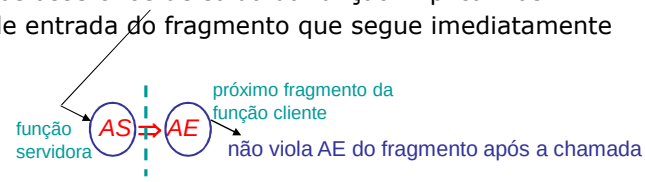
2. Demonstrar que a chamada está correta
   - imediatamente antes da chamada de uma função, argumenta-se a satisfação do contrato de entrada:
     - desta função, ou
     - da família de funções se for o caso
   - devem assegurar a validade do contrato de entrada da função:
     - o fragmento de código que imediatamente antecede a chamada



     - argumentação deve considerar tais casos especiais:
       - as eventuais expressões utilizadas na lista de argumentos da chamada
       - contrato de entrada da função a ser chamada pode depender do estado interno do módulo alvo
         - » solução: usa-se ou cria-se funções predicado do módulo servidor que informa a validade da condição associada do estado desejável

Argumentação de chamadas

3. Demonstrar que o retorno está correto
   - imediatamente após à chamada de uma função deve-se verificar se as assertivas de saída da função implicam as assertivas de entrada do fragmento que segue imediatamente a chamada



   - NÃO ESQUECER: deve-se verificar as assertivas de saída associadas com as diferentes condições de retorno
     - Isto é: se estas correspondem ao que é esperado pela chamada cliente
   - C++, Java e C#: cuidado especial deve ser tomado para o caso de **`throw`**, uma vez que exceções podem provocar o término de funções sem realizar todas as liberações de recursos necessárias

Laboratório de Engenharia de Software

Argumentação da seqüência, tentativa 1

$$AE : \uparrow_{AS_0}\ p_1\ \uparrow_{AS_1}\ p_2\ \uparrow_{AS_2}\ \cdots\ p_{n-1}\ \uparrow_{AS_{n-1}}\ p_n\ \uparrow_{AS_n}\ \Rightarrow AS$$

seja AS_0 = AE

$AS_0 \oplus p_1 \Rightarrow AS_1 \wedge AS_1 \oplus p_2 \Rightarrow AS_2 \wedge \ \ldots$, ou seja:

$$\forall i \mid (0 \le i < n): \quad AS_i \oplus P_{i+1} \Rightarrow AS_{i+1}$$

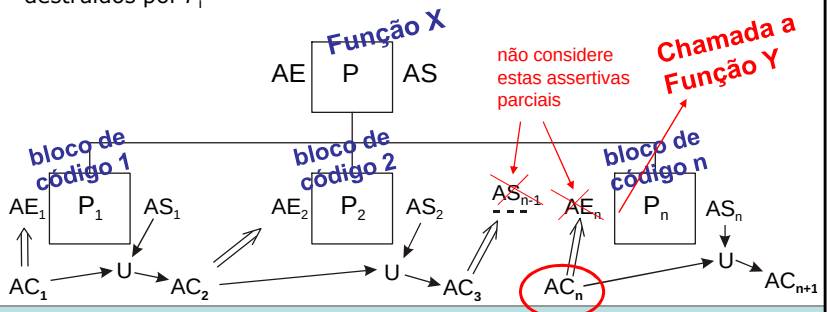
e $AS_n \Rightarrow AS$

- Problemas?
  - existem itens de assertivas que "pulam" fragmentos de código
    - são produzidos pelo fragmento P_i , mas não aparecem na entrada de P_{i+1}
  - assertivas de saída poderão precisar mencionar itens que são entrada do fragmento a seguir mas não são gerados por *P*
    - estabelecem, assim, um conflito com os critérios de qualidade das assertivas

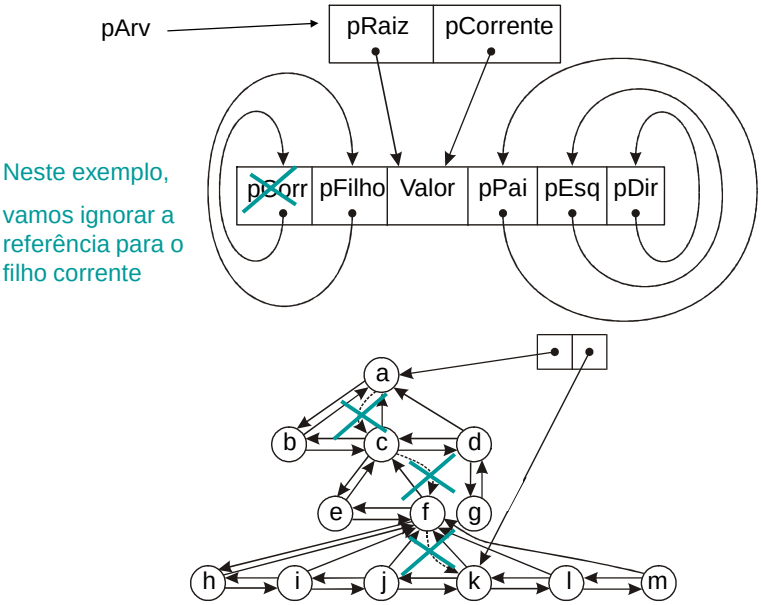
Laboratório de Engenharia de Software

Argumentação da seqüência

- Vamos usar assertivas AC_i que a cada passo conterão o efeito acumulado do processamento até o ponto i
- O operador U é uma união "adaptada"
  - substitui os itens existentes em AC_i pelos alterados por P_i (em AS_i)
  - adiciona ao conjunto os itens de AS_i que ainda não figuram em AC_i
  - exclui de AC_i os itens destruídos por P_i



Exemplo modelo: árvore n-ária



Neste exemplo,

vamos ignorar a referência para o filho corrente

Exemplo assertiva estrutural: árvore n-ária

Laboratório de Engenharia de Software

/* se a árvore está vazia, não existe raiz e nó corrente */

pArv->numNos == 0 ⇒ pArv->pRaiz == NULL && pArv->pCorrente == NULL

pArv->numNos != 0 ⇒ pArv->pRaiz->pPai == NULL &&
∃ pNo ∈ Arvore{ pArv } | pNo == pArv->pCorrente

/* caso contrário: pRaiz aponta p. raiz e pCorrente aponta para um dos nós */

∀ pNo ∈ Arvore{ pArv } :

/* se pNo é raiz, não deve ter pai e irmãos */

pNo->pPai == NULL ⇒ pArv->pRaiz == pNo &&
pNo->pEsq == NULL && pNo->pDir == NULL

/* pNo deve estar na lista de filhos do pai */

pNo->pPai != NULL ⇒ ∃ pElem ∈ Lista{pNo->pPai->ListaFilho} | pElem == pNo

/* todos os filhos de pNo apontam para o pai pNo */

pNo->pFilho != NULL ⇒ (∀ pFil ∈ Lista{ pNo->pFilho } : pFil->pPai == pNo)

pNo->pEsq == NULL && pNo->pPai == NULL ⇒ pArv->pRaiz == pNo

/* se for nó mais a esquerda (1º. Filho), deve ser o Raiz... ou deve ser apontado pelo pai */

pNo->pEsq == NULL && pNo->pPai != NULL ⇒ pNo->pPai->pFilho == pNo

pNo->pEsq != NULL ⇒ pNo->pEsq->pDir == pNo

/* todos os filhos não-1º. Devem ser apontados pelo irmão à esquerda */

pNo->pDir == NULL ⇒ true

/* últimos filhos não possuem irmãos à direita */

pNo->pDir != NULL ⇒ pNo->pDir->pEsq == pNo

/* todos os filhos, exceto último, devem ser apontados pelo irmão à direita */

Exemplo assertiva estrutural: árvore n-ária

Laboratório de Engenharia de Software

/* se a árvore está vazia, não existe raiz e nó corrente */

/* caso contrário: pRaiz aponta p. raiz e pCorrente aponta para um dos nós */

/* se pNo é raiz, não deve ter pai e irmãos */

/* pNo deve estar na lista de filhos do pai */

/* todos os filhos de pNo apontam para o pai pNo */

/* se for nó mais a esquerda (1º. Filho), deve ser o Raiz... ou deve ser apontado pelo pai */

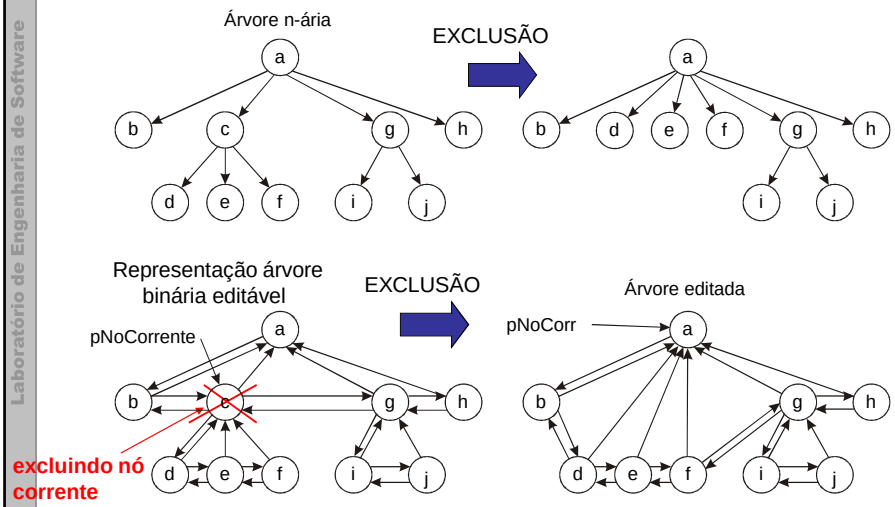
/* todos os filhos não-1º. Devem ser apontados pelo irmão à esquerda */

/* últimos filhos não possuem irmãos à direita */

/* todos os filhos, exceto último, devem ser apontados pelo irmão à direita */

Exemplo excluir nó: árvore n-ária

- Efeito de excluir nó corrente, ilustração



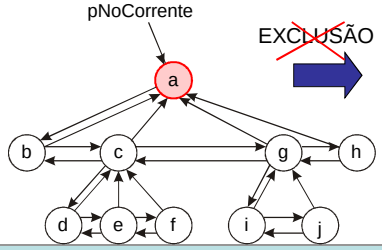
Exemplo excluir nó: árvore n-ária

Seja pNoCorrIni = entrada:pArv->pNoCorrente ;

Excluir nó corrente esboço da implementação

Verificar se **`entrada:pArv->pNoCorrente`** é raiz, termina se for

- Razão: não se permite excluir a raiz da árvore, pois poderia resultar em uma floresta.
- Poder-se-ia tratar o caso particular de raiz com nenhum ou um único filho, isso não será feito neste exemplo



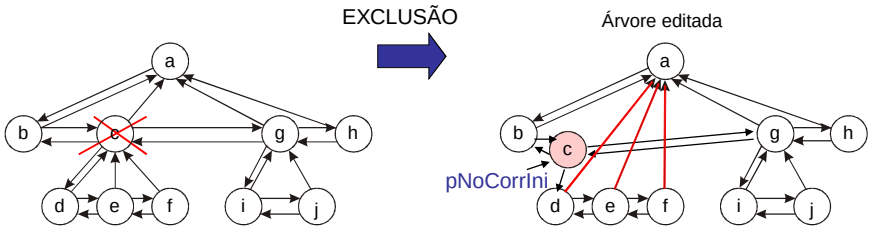
Exemplo excluir nó: árvore n-ária

Seja pNoCorrIni = entrada:pArv->pNoCorrente ;

Excluir nó corrente esboço da implementação

Verificar se **`entrada:pArv->pNoCorrente`** é raiz, termina se for

Acertar **`pPai`** de todos os filhos de **`pNoCorrIni`**



Laboratório de Engenharia de Software

Exemplo excluir nó: árvore n-ária

Seja pNoCorrIni = entrada:pArv->pNoCorrente ;

Excluir nó corrente esboço da implementação

Verificar se **`entrada:pArv->pNoCorrente`** é raiz, termina se for

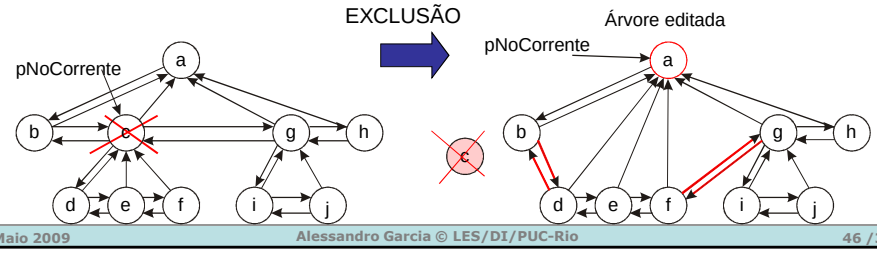
Acertar **`pPai`** de todos os filhos de **`pNoCorrIni`**

Concatenar lista filhos com lista irmãos à direita de **`pNoCorrIni`**

Concatenar lista irmãos à esquerda de **`pNoCorrIni`** com lista anterior

Acertar **`pNoCorrente`**

Excluir **`pNoCorrIni`**



Laboratório de Engenharia de Software

Exemplo excluir nó: árvore n-ária

Laboratório de Engenharia de Software

AE: Assertiva estrutural

Excluir nó corrente

AS: Seja pNoCorrIni = entrada:pArv->pNoCorrente ;

Assertiva estrutural &&

efeito da exclusão

pNoCorrIni == pArv->pRaiz ⇒ TRUE

pNoCorrIni != pArv->pRaiz ⇒

∀ pFil ∈ Lista{ pNoCorrIni->pFilhos } : pFil->pPai == pNoCorrIni->pPai

pais dos filhos são os avôs

/* faz nada se corrente é a raiz */

pNoCorrIni->pPai->pFilhos == ConcatLista(ListaAté{ pNoCorrIni->pEsq},

concatena os filhos com a lista dos irmãos à esquerda

ConcatLista(ListaFim{ pNoCorrIni->pFilhos } , Lista{ pNoCorrIni->pDir }))

concatena os filhos com a lista dos irmãos à direita

pArv->pNoCorrente == pNoCorrIni->pPai

pNoCorrIni destruído

ListaFim{ pX } representa a lista de irmãos a partir de pX inclusive até o final

ListaAté{ pX } representa a lista de irmãos da origem até o elemento pX inclusive

Exemplo excluir nó: árvore n-ária

Laboratório de Engenharia de Software

AE1: seja pNoCorrIni = entrada:pArv->pNoCorrente

pNoCorrIni->pPai != NULL *// testa se não é o raiz*

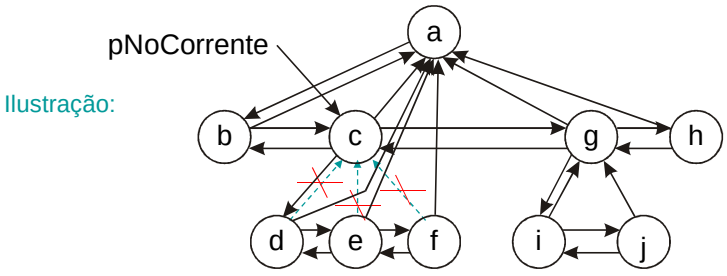
Acertar pPai de todos os filhos de pNoCorrIni

AS1: Paratodos pFil pertence Lista{ pNoCorrIni->pFilhos } :

pFil->pPai == pNoCorrIni->pPai

filhos agora apontam para o avô

Ilustração:



Exemplo excluir nó: árvore n-ária

Laboratório de Engenharia de Software

AE2: pNoCorrIni != NULL AS1 é irrelevante para o resto do algoritmo

Concatenar lista filhos com lista irmãos à direita de pNoCorrIni

AS2: pIrmao == PrimeiroElem(ConcatLista(

1 Lista{ pNoCorrIni->pListaFilhos } ,

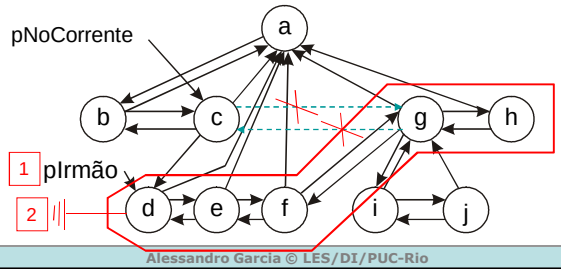
ListaFim{ pNoCorrIni->pDir }))

2 pIrmao != NULL ⇒ pIrmao->pEsq == NULL

// pIrmao será NULL caso a lista seja vazia.

// Ocorre se e somente se pNoCorrIni não tiver filhos e não tiver irmãos à direita.

Ilustração:



Exemplo excluir nó: árvore n-ária

Laboratório de Engenharia de Software

AE3: AS2 U AE2

pNoCorrIni != NULL

pIrmao == PrimeiroElem(ConcatLista(Lista{ pNoCorrIni->pListaFilhos } ,

ListaFim{ pNoCorrIni->pDir }))

pIrmao != NULL ⇒ pIrmao->pEsq == NULL

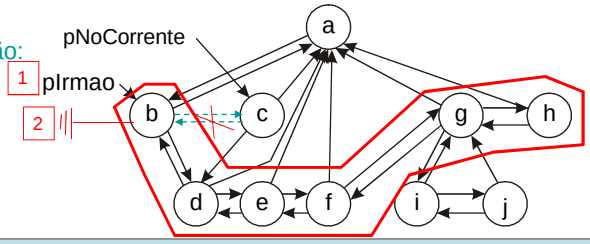
Concatenar irmãos à esquerda de pNoCorrIni com lista pIrmão

AS3: 1 pIrmao == PrimeiroElem(ConcatLista(

ListaAté{ pNoCorrIni->pEsq } , entrada:pIrmao))

2 pIrmao != NULL ⇒ pIrmao->pEsq == NULL

Ilustração:



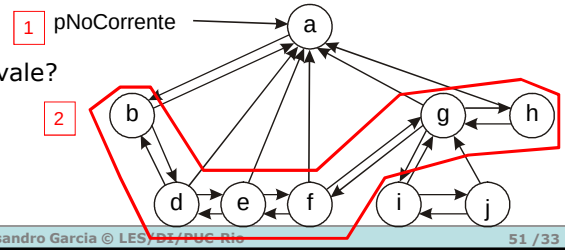
Exemplo excluir nó: árvore n-ária

AE4: AS3 U AE3
pNoCorrIni != NULL
pIrmao == PrimeiroElem(ConcatLista(ListaAté{ pNoCorrIni->pEsq } ,
ConcatLista(Lista{ pNoCorrIni->pListaFilhos } ,
Lista{ pNoCorrIni->pDir })))
pIrmao->pEsq == NULL

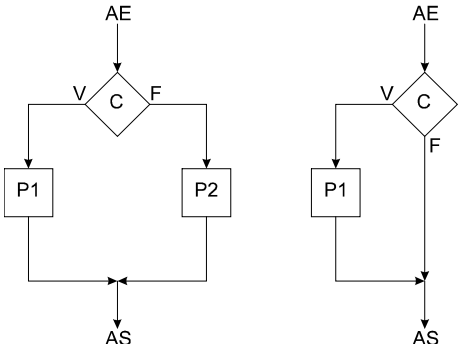
Acertar pNoCorrente

AS4: 1 pArv->pNoCorrente == pNoCorrIni->pPai
2 pArv->pNoCorrente ->pFilhos == PrimeiroElem(ConcatLista(
ListaAté{ pCorrIni->pEsq } , ConcatLista(
Lista{ pCorrIni->pFilhos } ,
ListaFim{ pCorrIni->pDir })))
pNoCorrIni destruído

- A assertiva estrutural vale?



Argumentação de seleções



- if then ... else ...:
  - (*AE* && (*C* == **TRUE**)) ⊕ *P1* ⇒ *AS*
  - (AE && (*C* == **FALSE**)) ⊕ *P2* ⇒ *AS*
- if then ... sem a cláusula else:
  - (*AE* && (*C* == **TRUE**))⊕ *P1* ⇒ *AS*
  - (*AE* && (*C* == **FALSE**)) ⇒ *AS*

Proposições a serem provadas

Argumentação de seleções múltiplas

```
if C1
{
   p1
} else if C2
{
   p2
} else if C3
{
   ...
} else if Cn
{
   pn
} else
{
   pdefault
}
```

Agora temos que provar n + 1 proposições

- $\forall\, i \mid (1 \le i \le n) :$
  AE && (($\forall\, j \mid (1 \le j < i) : C_j$ == **FALSE**)
  && C_i == **TRUE**) $\oplus$ $P_i \Rightarrow AS$
- AE && ($\forall\, i \mid (1 \le i \le n) : C_i$ == **FALSE**)
  $\oplus$ $P_{default} \Rightarrow AS$
- $\forall\, i \mid (1 \le i \le n)$ && (($\forall\, j \mid (1 \le j < i) :$
  C_i && C_j == **FALSE**) condições não são ambíguas
- $\bigcup_{1 \le i \le n} C_i$ == **TRUE** o conjunto de condições é completo

Para **`switch`** é similar

FIM