

Aspectos Formais de Apoio aos Testes 2

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015

Especificação

- Objetivo desse módulo
 - mostrar como assegurar **por construção** a corretude das transições em máquinas de estado
 - apresentar um método de teste baseado em assertivas
- Justificativa
 - o teste baseado em assertivas requer um bom domínio do uso de assertivas e de argumentação da corretude
- Texto
 - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulos 5, 7 e 8
 - Staa, A.v.; *Programação Modular*; Rio de Janeiro: Campus/Elsevier; 2000; capítulo 13

Parêntesis – quadrado mágico



```
InserirLinha( int inxLinha )
{
    if ( inxLinha >= tamQuadrado )
    {
        EncontrouQuadrado( ) ;
        return ;
    } /* fim if */
    for ( int i = 0 ; i < numPalavras ; i++ )
    {
        if ( ValePrefixoPalavraLinha( vtPalavras[ i ] , inxLinha ) )
        {
            CopiaRestoPalavraLinha( vtPalavras[ i ] , inxLinha ) ;
            InserirColuna( inxLinha ) ;
            ApagaRestoPalavraLinha( inxLinha ) ;
        } /* fim if */
    } /* fim for */
} /* fim inserir linha */
```

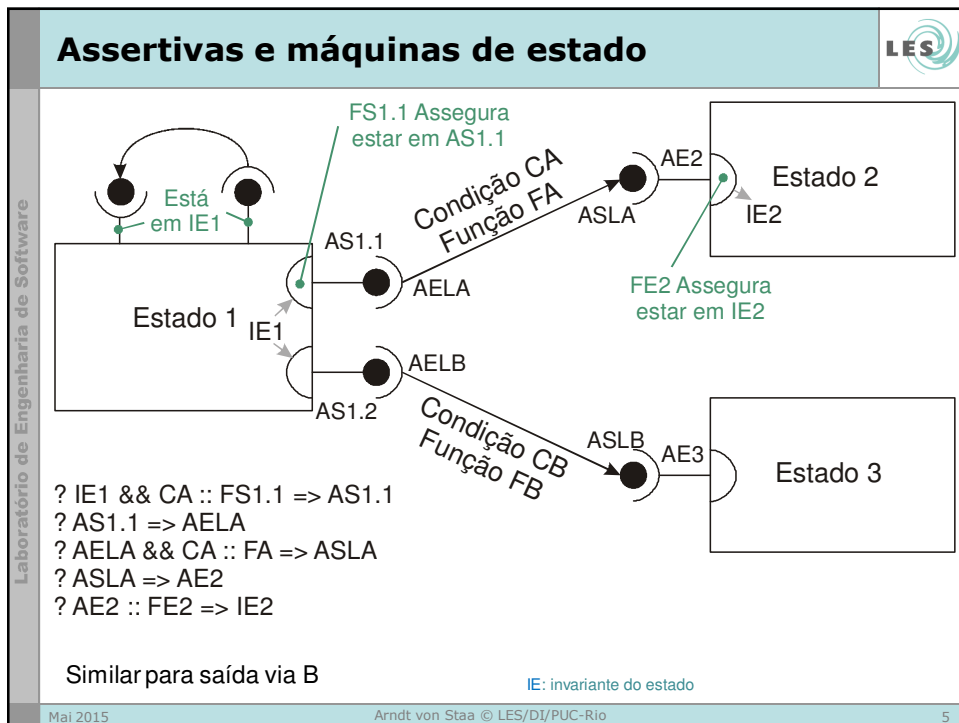
```
InserirColuna( int inxColuna )
{
    EXC_ASSERT( inxColuna < tamQuadrado ) ;
    for ( int i = 0 ; i < numPalavras ; i++ )
    {
        if ( ValePrefixoPalavraColuna(
            vtPalavras[ i ] , inxColuna ) )
        {
            CopiaRestoPalavraColuna(
                vtPalavras[ i ] , inxColuna ) ;
            InserirLinha( inxColuna + 1 ) ;
            ApagaRestoPalavraColuna( inxColuna ) ;
        } /* fim if */
    } /* fim for */
} /* fim inserir coluna */
```

vtPalavra é global para todas as instâncias de recursão e guarda o estado acumulado até o corrente nível de recursão

Máquinas de estados generalizadas



- Em máquinas de **estado generalizadas**
 - os estados (elipses ou caixas)
 - estabelecem assertivas de entrada a serem satisfeitas quando ativados de outro estado
 - estabelecem uma invariante do estado
 - possivelmente realizam processamento
 - estabelecem assertivas de saída
 - possivelmente especializadas para as transições de saída
 - as transições (arestas)
 - estabelecem uma assertiva de entrada
 - definem uma condição a ser satisfeita para poder transitar pela aresta
 - no máximo uma pode ser “else”, transita se nenhuma outra condição for válida
 - possivelmente realizam algum processamento
 - estabelecem uma assertiva de saída (término da transição)



Assertivas e máquinas de estado

Laboratório de Engenharia de Software

LES


- Cada estado possui
 - assertiva invariante de **estar no estado**
 - ex. editor gráfico: as pegas do objeto selecionado existem, estão visíveis e podem ser selecionadas pelo mouse
 - assertiva invariante de **não estar no estado**
 - ex. editor gráfico: as pegas do objeto não existem
- Observações
 - a **ação de entrada** deve assegurar a validade da assertiva de estar no estado
 - podem ser realizadas outras ações, tais como inicializações de variáveis locais ao estado
 - a **ação de saída** deve assegurar a validade da assertiva de não estar no estado
 - podem ser realizadas outras ações, por exemplo log de término

Pegas – marcadores através dos quais se pode aumentar, diminuir ou girar o(s) objeto(s) gráfico(s) selecionado(s)

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
6

Laboratório de Engenharia de Software

Assertivas e máquinas de estado




- Cada estado possui, continuação
 - assertiva de entrada externa
 - condições que devem valer ao ativar o estado ao **vir de outro** estado
 - ex. editor gráfico: o *mouse* deve estar sobre parte ativa da área de seleção do objeto
 - ação de entrada
 - ação a ser realizada caso o estado seja ativado a partir de outro estado
 - assegura a validade da assertiva de estar no estado
 - » ex. editor gráfico: tornar visíveis as pegadas
 - » ex. preparar *undo* pontual – salvar o estado do objeto gráfico selecionado

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
7

Laboratório de Engenharia de Software

Assertivas e máquinas de estado



- Cada estado possui, continuação
 - assertiva de saída externa
 - condições que devem valer ao sair do estado e **ir para outro estado** com base em determinada transição
 - a assertiva de saída pode depender da transição selecionada
 - » entretanto, é melhor ter uma única assertiva de saída
 - asseguram a validade de não estar no estado
 - » ex. editor gráfico: o mouse deve estar sobre nenhuma parte ativa da área de seleção do objeto ao ser clicado
 - ação de saída
 - ação a ser realizada caso se transite para outro estado
 - ex. editor gráfico: apagar as pegadas restaurando as imagens que haviam sido obliteradas
 - ex. desempilhar recursos – eliminar os objetos que não foram ancorados em outra estrutura

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
8

Assertivas e máquinas de estado



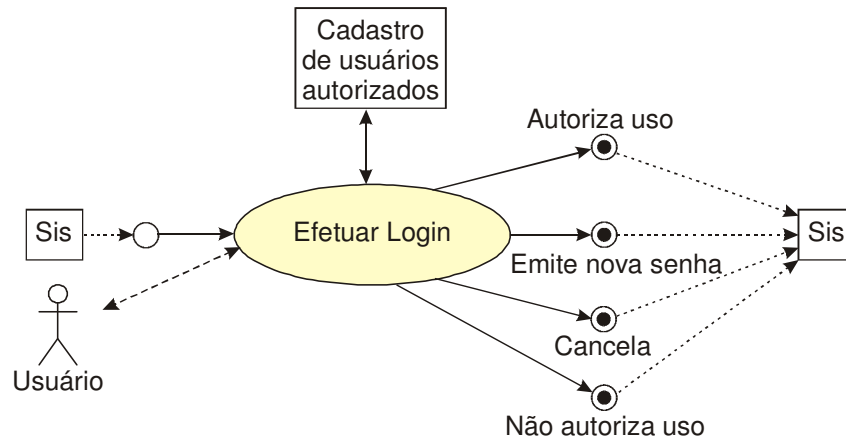
- Cada estado possui, continuação
 - Assertiva de entrada interna e assertiva de saída interna
 - condições que valem quando a transição retorna ao mesmo estado
 - ex. realizar operações sobre elementos do estado

Assertivas e máquinas de estado



- Cada transição estabelece as condições que permitem transitar por ela
 - para cada par $\langle t_i, t_j \mid i \neq j \rangle$ de transições de saída de um dado estado deve valer $t_i \ \&\& \ t_j == \text{false}$
- Uma transição **else** (sem condição explícita) corresponde a:
 $\text{!trans}_1 \ \&\& \ \text{!trans}_2 \ \&\& \ \dots \ \&\& \ \text{!trans}_n$
onde n é o número de transições de saída rotuladas
 - pode existir no máximo uma transição **else**
- Transições podem também definir ações a serem realizadas ao transitar por elas

Login, contexto máquina de estados



Login, contexto máquina de estados



- Assertiva de entrada
 - no caso é vazia, pois corresponde a não estar no estado
- Assertiva de saída
 - Autoriza uso $\Rightarrow \{ \text{AUTORIZA}, \text{DireitosUso} \}$
 - Não autoriza uso $\Rightarrow \{ \text{NAO_AUTORIZA}, \text{vazio} \}$
 - Emite nova senha $\Rightarrow \{ \text{NOVA_SENHA}, \text{vazio} \}$
 - Cancela $\Rightarrow \{ \text{CANCELA}, \text{vazio} \}$
- Invariante de estar no estado
 - `ContaErros` == total de erros cometidos até o momento
 - $0 \leq \text{ContaErros} \leq \text{LimiteErrosLogin}$
 - janela `login` aberta
- Invariante de não estar no estado
 - janela `login` não existe

variável local do estado

Login: bloco Início da máquina de estados



- Esboço do código

```
abrir janela de login  
condicaoUso = ObterCondicaoUso( INICIO ) ;  
fechar janela de login  
retornar condicaoUso ;
```

- estados inicial e final de "ObterCondicaoUso"

```
case INICIO :  
    condicaoUso = < MODO_ILEGAL , VAZIO > ;  
    estado = FORNECER_DADOS ;  
    break ;  
  
case FIM:  
    return condicaoUso ;
```

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

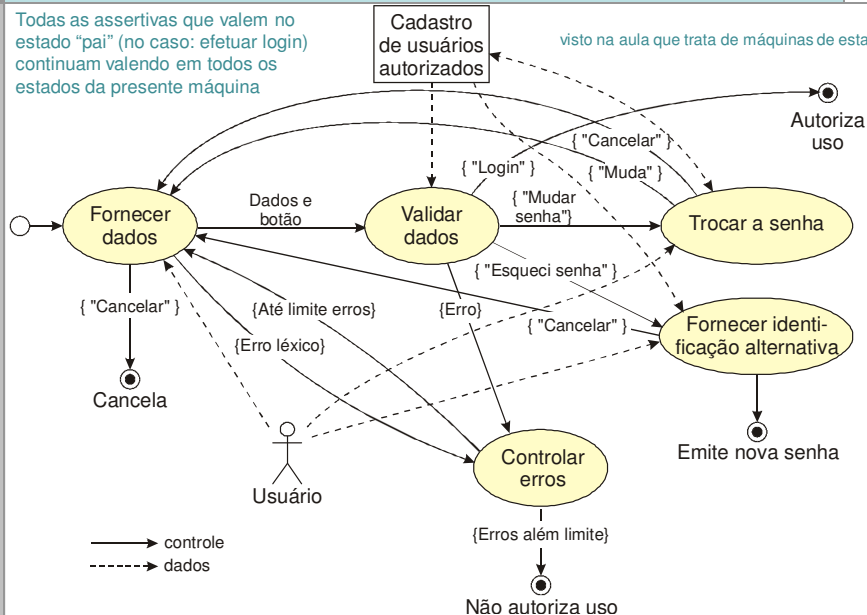
13

Login, máquina de estados, decomposição



Todas as assertivas que valem no estado "pai" (no caso: efetuar login) continuam valendo em todos os estados da presente máquina

visto na aula que trata de máquinas de estados



Mai 2015

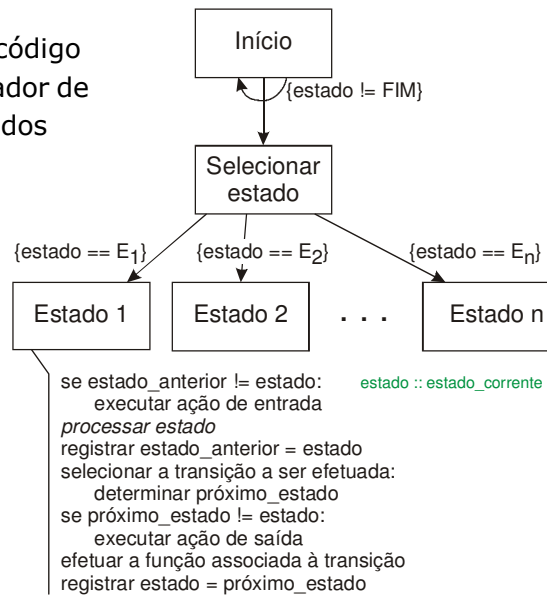
Arndt von Staa © LES/DI/PUC-Rio

14

Máquina de estados, implementação



- Organização do código de um interpretador de máquina de estados



Assume-se que exista
 somente uma assertiva de
 saída em cada estado

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

15

Login, estado "Fornecer dados"



- Ação de entrada e assegurar invariante de estar no estado
 - esvaziar campos de entrada de dados
 - gerar caracteres de controle
 - invariante: campos de entrada de dados selecionáveis
- Ação de saída e assegurar invariante de não estar no estado
 - invariante: campos de entrada de dados não selecionáveis
- Assertiva de entrada
 - qualquer ← Precisa disso? Idêntico à invariante de não estar no estado.
- Assertiva de saída
 - Dados e botão => (idUsuário , senha , botão)
 - Erro léxico => () ou (idMensagem) ?
 - Cancelar => (Cancela , vazio)

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

16

Login, estado "Fornecer dados"



- Esboço do código

AE: qualquer

case FORNECER_DADOS :

Limpar campos de entrada de dados

Gerar caracteres de controle

Ativar entrada de dados

Usuário digita nos campos

Usuário clica botão de ação

Programa espera usuário fazer alguma coisa.
Neste exemplo, clicar um botão devolve o controle para o programa.

if (botao == CANCELAR) estado = CANCELAR ;

else if ((! controleValido(caracteresControle)

|| ! idUsuarioVale(idUsuario)) estado = ERRO ;

else estado = VERIFICAR ;

break ;

deveria verificar se os botões
selecionados são válidos?
Se não forem temos um erro de
implementação

AS: Se estado == VERIFICAR

idUsuario lexicamente correto

senha qualquer

botão ∈ { OK , TROCAR_SENHA , EMITIR_NOVA_SENHA }

Se estado == CANCELAR ou ERRO
qualquer

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

17

Login, estado "Validar dados"



- Assertiva de entrada

- idUsuario lexicamente correto, senha ,
botão ∈ { LOGIN , TROCAR_SENHA , EMITIR_NOVA_SENHA }
- OBS: deveria existir um método validarLexicoUsuario(idUsuario)

- Assertiva de saída

- OK => condicaoUso ==
{ AUTORIZA , DireitosUso }
- Mudar senha => valem (idUsuario , senha)
- Esqueci senha => vale (idUsuario)
- Erro => () ou (msg == idMensagem) ?

- Invariante de estar no estado

-

- Invariante de não estar no estado

-

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

18

Login, estado "Validar dados"



- Esboço do código

AE: idUsuario lexicamente correto
senha qualquer
botão ∈ { OK, TROCAR_SENHA, EMITIR_NOVA_SENHA }

```
case VERIFICAR :
    dadosUsuario = obterDados( idUsuario ) ;
    if ( ! existe( dadosUsuario ) ) {
        estado = ERRO ; break } AS: qualquer
    if ( botao == EMITIR_NOVA_SENHA ) {
        estado = NOVA_SENHA ; break } AS: <idUsuario> existe no cadastro
    if ( ! dadosUsuario.verificarSenha( senha ) ) {
        estado = ERRO ; break ; } AS: qualquer
    if ( botao == TROCAR_SENHA ) {
        estado = TROCA_SENHA ; AS: <idUsuario, senha> existe no cadastro
    } else if ( botao == OK ) {
        condicaoUso = { AUTORIZA , dadosUsuario.getDireitosUso( ) } ;
        estado = FIM ; AS: <idUsuario, senha> existe no cadastro,
    } else estado = ESTADO_ILEGAL ; direitosUso corresponde a esse par
    break ;
```


Por que este código? Qual seria a assertiva de saída? E se fosse um `throw`?

O resto fica para exercício ☺



Laboratório de Engenharia de Software

TESTE USANDO ASSERTIVAS




Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

21

Laboratório de Engenharia de Software

Assertivas como oráculos



- Teste **metamórfico**

Metamorfose: Transformação de um ser (problema a resolver) em outro (Aurélio) : "prova" da correteude através de exemplos

 - utiliza **assertivas executáveis** como oráculo
- seja $AE :: f(d) \Rightarrow AS$ a especificação da função f a ser implementada
 - sem perda de generalidade AE e AS englobam a $AINV$
- seja D o domínio dos dados aceitos por f
- seja p uma implementação de f
- seja $T \subseteq D$ uma suíte de teste para p
- assumindo que seja gerada uma **exceção** caso uma **assertiva executável não valha** para um $t \in T$

Evidentemente, se AE não valer, toda a argumentação não valerá

$$AE :: p(t) ::= \text{exceção}$$

produz, ou resulta em

$$? !(\exists t \in T \mid (AE :: p(t) ::= \text{exceção})) \Rightarrow$$

$$\forall t \in T : (AE :: p(t) \Rightarrow AS)$$

equivalente segundo a suíte de teste

mas então $\forall t \in T : (p(t) == f(t))$

logo $? T \rightarrow D \Rightarrow p \rightarrow f$

$a \rightarrow b :: a$ tende para ser igual a b

Gotlieb, A.; Botella, B.; "Automated metamorphic testing"; Proceedings of the COMPSAC 2003 - 27th Annual International Computer Software and Applications Conference, 2003; pages 34-40

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

22

Problemas com o teste metamórfico 1/4



- $\forall d \in D \mid ? AE :: f(d) \Rightarrow AS$
 - ou seja a especificação está perfeita
- de maneira geral é impossível determinar se uma suíte de testes $T \subseteq D$ é **confiável**
 - uma suíte de teste T será **confiável** para p se, caso p contenha um defeito, então existirá pelo menos um caso de teste $t \in T$ para o qual $p(t) \Rightarrow AS$ é falso
- se não ocorrer exceção não saberemos se T é confiável, logo não saberemos se p é uma implementação exata de f
- porém, à medida que $T \rightarrow D$ sem que ocorra exceção, nossa **crença** de que T é confiável ganha sustentação
 - caso as assertivas permaneçam ativas, T conterá também todos os dados usados durante o uso produtivo de p ,
 - isso implica que à medida que o **tempo de uso aumenta** sem que ocorram exceções a nossa crença ganha substância, **desde que as assertivas permaneçam ativas**

Problemas com o teste metamórfico 2/4



- $\exists d \in D \mid ? AE :: f(d) \not\Rightarrow AS$
 - temos ou um **erro de especificação** ou uma **AS errada**
 - evidentemente este erro deveria refletir-se também em p
 - se p **for** uma implementação exata de f será gerada uma exceção para todos $t \in S$, onde S é o **conjunto similar a d**
 - **conjunto similar a d** :: $S \subseteq D \mid \forall s \in S$ gera uma exceção usando o **mesmo throw** que o usado por d
 - entretanto, se p **não for** uma implementação exata de f pode ocorrer que não seja gerada a exceção para algum $t \in S$
 1. pode ser que p tenha **corrigido o defeito** da especificação de f e assim passou a ser uma implementação correta
 - a correção realizada **deveria estar documentada**
 2. pode ser que p contenha **outro defeito** além do erro herdado da especificação, gerando ou não uma exceção não similar a d
 3. pode ser que $\neg(\exists t \mid t \in T \ \&\& \ t \in S)$
- ou seja, se a especificação puder estar errada e não ocorrer uma exceção ao usar p , então não saberemos nem se a **especificação** está errada, nem se p contém defeito ou não

Problemas com o teste metamórfico 3/4



- o teste metamórfico **depende da qualidade das assertivas**
 - assertivas incompletas podem deixar de observar falhas
 - podem ocorrer **falsos negativos**
 - assertivas incorretas levam a **falsos positivos**
- Para reduzir as chances de assertivas incompletas ou incorretas
 - devem figurar nas assertivas de entrada **todos os dados, estados e recursos** usados no fragmento de interesse antes de serem redefinidos (alterados, destruídos) neste fragmento
 - no caso de funções, considere a interface conceitual
 - no caso de fragmentos de código considere os elementos usados no fragmento
 - as assertivas de entrada devem poder ser **assumidas verdadeiras** ao iniciar
 - precisa-se verificar se isso é verdade → instrumentação
 - devem figurar nas assertivas de saída **todos os dados, estados e recursos** criados, alterados ou destruídos no fragmento de interesse
 - é mais importante controlar a saída do que a entrada

Problemas com o teste metamórfico 4/4



- sempre podemos definir assertivas de entrada e saída efetivamente calculáveis?
 - e se o cálculo de AS envolver a funcionalidade de p ?
 - considere o cálculo do juro não uniforme composto
 - juros diferentes mês a mês, acumulando juros com o principal
 - e se a assertiva depende da **acurácia** dos dados externos ?
 - considere a verificação de defeitos em um tubo
 - a partir dos dados recebidos de sensores queremos determinar se existe ou não uma rachadura
 - como saber se os dados recebidos correspondem **sempre** à realidade?
 - como saber se as medições deveriam ou não acusar uma rachadura?
 - e se a assertiva depende de uma redundância inexistente?
 - considere o problema do elemento B antecessor de um elemento A em uma lista mono encadeada
 - considere o problema de verificar se o elemento corrente de um grafo denso com um número muito grande de vértices é um elemento deste grafo

Teste usando assertivas, conclusão



- Se não ocorrerem exceções, não saberemos
 - qual o grau de eficácia da suíte de teste
 - se a especificação está correta e adequada
 - uma especificação correta não implica que seja **adequada** !
 - se as assertivas estão completas e corretas
 - nunca temos como saber se ocorreram **falsos negativos**
 - caso sejam incorretas, pode ocorrer que não sejam exercitadas de modo a apresentar os correspondentes **falsos positivos** usualmente pode
 - consequentemente, se puder existir $d \in D \mid d \notin T$ e para o qual $p(d)$ falhe, deveríamos deixar as assertivas ativas em p e prever um adequado tratamento caso ocorra uma exceção
 - problema: custo operacional induzido pelas assertivas executáveis
 - infelizmente, as assertivas executáveis podem estar incompletas, portanto, mesmo que permaneçam no código, podem ocorrer casos em que erros passem despercebidos

Teste usando assertivas, conclusão



- Como esperado

Não existe solução miraculosa para assegurar que o teste de programas seja confiável

Teste usando assertivas, tratamento



- Se ocorrerem exceções temos que verificar as causas:
 - defeito na especificação (ou no projeto)
 - corrigir a especificação
 - defeito no programa
 - corrigir o programa
 - defeito nas assertivas
 - caso seja um **falso positivo**: corrigir as assertivas
 - caso seja um **falso negativo**, i.e. um erro observado por outros meios: corrigir a assertiva e a suíte de teste de modo que passe a observar o erro e **somente então** corrigir o programa
 - sempre conduzir uma **inspeção rigorosa das assertivas** para verificar se estão completas
 - o custo disso tende a ser bem menor do que o custo de inspecionar o programa
 - além disso reduz o retrabalho inútil e o custo de diagnose

Teste usando assertivas, pragmática



- Consequência do teste metamórfico
 - deixa-se para as assertivas executáveis a tarefa de identificar as falhas, i.e. a assertiva é o oráculo
 - mesmo correndo o risco de um teste **não confiável** pode-se
 - gerar casos de teste usando **geradores de dados aleatórios**
 - conviver com falhas provocadas por causas exógenas ou por defeitos ainda não conhecidos
 - as assertivas geram exceções
 - os “objetos exceção” devem conter informações de apoio à diagnose
 - arquivo / linha de código da assertiva, ou do **throw**
 - estado da pilha de execução
 - estado de variáveis relevantes
 - ...
 - consegue-se uma **sensível redução de custos dos testes**
 - consegue-se programas **robustos**

Laboratório de Engenharia de Software

LES

POR QUE ARGUMENTAR A CORREITUDE ?

Mai 2015Arndt von Staa © LES/DI/PUC-Rio31

Laboratório de Engenharia de Software

LES

Objetivo da argumentação da corretude

- Importante não é a prova em si, mas é relevante:
 - o fato de **saber como provar** a corretude de um programa
 - a **existência** de “boas” assertivas
 - ou seja, importante é procurar tornar formais as especificações

Mai 2015Arndt von Staa © LES/DI/PUC-Rio32

Objetivo da argumentação da corretude



- A **argumentação** da corretude
 - é uma técnica de revisão (ou inspeção)
 - ou seja, é anterior aos testes
 - **pode ser aplicada** rotineira e economicamente ao desenvolver software, ainda **antes de se dispor do código**
 - verifica a **corretude** de programas
 - **leve** → requer pouco esforço e formação pouco sofisticada
 - **eficaz** → reduz sensivelmente o número de defeitos remanescentes
 - no entanto **não assegura a corretude**
 - ou seja, não elimina a necessidade de testes cuidadosos

Por que argumentar a corretude?



- Erros que permanecem despercebidos podem gerar grandes lesões
- Erros somente podem ser controlados após terem sido observados, i.e. quando forem falhas
 - é importante capacitar os programas a observarem erros
 - quanto mais cedo melhor
 - mais fácil diagnosticar a causa
 - é importante eliminar o **máximo economicamente justificável** dos defeitos existentes
 - é importante poder estimar a probabilidade do código conter defeitos
 - isso é possível?
 - como não é e como não sabemos quais são os defeitos remanescentes, precisamos monitorar a execução

Por que argumentar a corretude?



- Programas devem ser capazes de **detectar** a ocorrência de falhas relevantes (dano e/ou relevância altos)
 - auto-verificação (será vista quando estudarmos instrumentação)
- Quanto maiores os possíveis danos (i.e. impacto)
 - maior deve ser a preocupação para **evitar a injeção** de defeitos
 - maior deverá ser o **rigor** das assertivas inseridas
 - idealmente verificadas em tempo de execução, i.e. assertivas executáveis
 - maior deve ser o **rigor** da argumentação

ARGUMENTAÇÃO DA CORRETUDE



Argumentação de chamadas



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

1. Determinar a correta associação entre a chamada e o corpo da função a ser efetivamente executado)
 - o corpo pode variar em virtude de herança, ou do uso de ponteiros para função
2. Determinar a correta associação entre os argumentos contidos na chamada e os parâmetros definidos na implementação da função que será ativada
 - a ordenação de argumentos e parâmetros deve estar coerente
 - a correspondência semântica entre argumentos e parâmetros deve estar correta
3. Demonstrar que a chamada está correta
4. Demonstrar que o retorno está correto

Argumentação de chamadas



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.
 - redefinição, no caso de herança
 - ponteiros para função no caso de C/C++
 - devem estar definidas as assertivas de entrada e saída da família, bem como os requisitos e as hipóteses da família
 - a assertiva de entrada da família deve implicar a assertiva de entrada de cada um dos membros dessa família
 - a assertiva de entrada de cada função da família não pode ser mais restritiva que a assertiva de entrada da família
 - a assertiva de saída de cada membro da família deve implicar a assertiva de saída da família
 - a assertiva de saída da família não pode ser mais restritiva que a assertiva de saída de cada função da família

- usualmente a associação é posicional
 - precisa-se assegurar que a ordenação dos parâmetros é exatamente igual à dos argumentos
- usualmente o tipo usado nas declarações é computacional
 - exemplos: int, double, string
 - a associação de tipos computacionais não assegura a correteude semântica da associação, ex.
 - é possível atribuir o string nome de pessoa ao string nome de produto
 - ideal: usar análise estática para verificar a correteude semântica
 - precisa-se poder assegurar a correteude semântica
 - o tipo semântico de cada um dos parâmetros precisa estar documentado

- imediatamente antes da chamada de uma função, argumenta-se a satisfação da assertiva de entrada desta função, ou, se for o caso, da família de funções
- devem assegurar a validade da assertiva de entrada:
 - o **fragmento de código** que imediatamente **antecede** a chamada
 - as eventuais expressões utilizadas na **lista de argumentos**
 - os **estados** requeridos pela função para satisfazer as necessidades da **chamada cliente**, exemplos:
 - a pilha não deve estar vazia
 - arquivo X está aberto

ambos exemplos dependem do estado interno do objeto (módulo) que contém a função,

- devem existir funções que informam a validade da propriedade (funções **predicado**) ex. `ehPilhaVazia ()`
- não se pode utilizar essas propriedades caso não existam as **funções predicado**

Argumentação de chamadas



4. Demonstrar que o **retorno está correto**
 - no código cliente (chamada de uma função) deve-se verificar se as assertivas de saída da função implicam as assertivas de entrada do fragmento imediatamente após a chamada
 - deve-se verificar se os estados de saída correspondem ao esperado pelo cliente
 - devem-se verificar as diferentes condições de retorno e 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

Argumentação de corotinas



- Deve ser considerado o uso de **corotinas**.
 - Corotinas são subprogramas que podem ser **suspensos**, podendo depois **retomar** a execução no ponto em que foram suspensas. Iteradores são exemplos de corotinas
 - As seguintes operações podem ser realizadas com corotinas:
 - **chamar**, neste caso é criada uma instância de ativação
 - **suspender**, neste caso cessa a execução, sem destruir a instância de ativação, retornando ao último ponto de chamada ou retomada
 - **retomar**, neste caso retoma-se a execução de uma instância de ativação que estava suspensa, no mesmo estado em que esta se encontrava ao suspender pela última vez
 - **terminar**, neste caso é destruída a instância de ativação

Exemplo de corotina – orientação a objetos



- criar objeto → construtor
 - chamar um construtor corresponde a *chamar* a corotina
 - estabelece o “registro de ativação”, i.e a instância de ativação → um objeto de uma dada classe
 - retornar do construtor corresponde a *suspender*
 - o estado do objeto é preservado até a próxima retomada
- chamar método de um objeto
 - chamar método corresponde a *retomar*
 - usa e possivelmente modifica o “registro de ativação”
 - retornar do método corresponde a *suspender*
 - o estado do objeto é preservado até a próxima retomada
- destruir objeto → destrutor
 - o “registro de ativação” é destruído
 - corresponde a *terminar* a corotina

Exemplo de corotina – multi-programação



- Processos em um ambiente de multi-programação (*multi-threading*) são exemplos de corotinas sem pontos de suspensão explícitos
 - criar processo
 - cria um descritor de estado do processo
 - interromper um processo
 - preserva o estado do processo no momento da interrupção
 - ativar o processo
 - restaura o estado do processo partir do descritor preservado e retoma a execução
 - terminar o processo
 - destrói o descritor de estado do processo

Argumentação da sequência



- As assertivas auxiliares AC_i conterão a cada passo o efeito acumulado do processamento até o ponto i

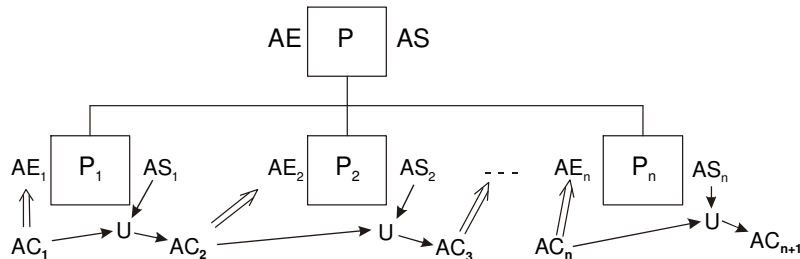
$$AE \Rightarrow AC_1$$

$$\forall i | (1 \leq i \leq n): AC_i \Rightarrow AE_i :: P_i \Rightarrow AC_{i+1} = AS_i \cup AC_i$$

$$AC_{n+1} \Rightarrow AS$$

- O operador U é uma união "adaptada"

- **substitui** os itens existentes em AC_i pelos resultantes 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

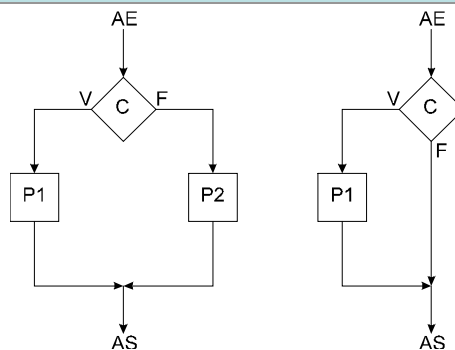


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

45

Argumentação de seleções



- if then ... else ...:

- $(AE \ \&\& \ (C == \text{TRUE})) :: P1 \Rightarrow AS$
- $(AE \ \&\& \ (C == \text{FALSE})) :: P2 \Rightarrow AS$

- if then ... sem a cláusula else:

- $(AE \ \&\& \ (C == \text{TRUE})) :: P1 \Rightarrow AS$
- $(AE \ \&\& \ (C == \text{FALSE})) \Rightarrow AS$

Proposições a serem provadas

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

46

Argumentação de seleções múltiplas



```
if C1
{
    P1
} else if C2
{
    P2
} else
{
    ...
} else if Cn
{
    Pn
} else
{
    Pdefault
}
```

Agora temos que provar $n + 1$ proposições

- $\forall i \mid (1 \leq i \leq n) :$
 $(AE \ \&\& \ ((\forall j \mid (1 \leq j < i) : C_j == \text{FALSE})$
 $\ \&\& \ C_i == \text{TRUE}) :: P_i \Rightarrow AS)$
- $AE \ \&\& \ (\forall i \mid (1 \leq i \leq n) : C_i == \text{FALSE})$
 $:: P_{default} \Rightarrow AS$

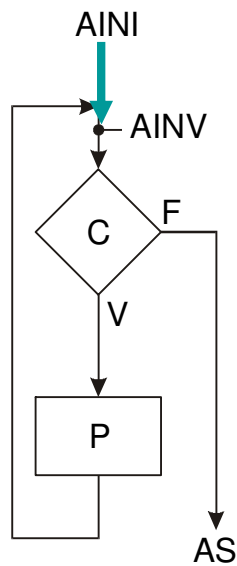
Para **switch** precisa de uma adaptação simples

Assertiva invariante



- A argumentação de repetições utiliza indução matemática
- A hipótese de indução é estabelecida pela **assertiva invariante** (AINV)
 - a assertiva invariante deve inter-relacionar todas as variáveis e estados que são **criados ou alterados** no decorrer da repetição
 - as relações devem ser estabelecidas de modo que a (mesma) assertiva invariante reflita o estado da repetição para qualquer número de 0 ou mais iterações
 - a assertiva invariante deve assumir que o número de iterações é indefinido
 - repetindo, deve-se dissociar o comportamento da repetição, refletido pela assertiva invariante, da condição de término, refletido pelas expressões de controle da repetição.

Proposições para a repetição

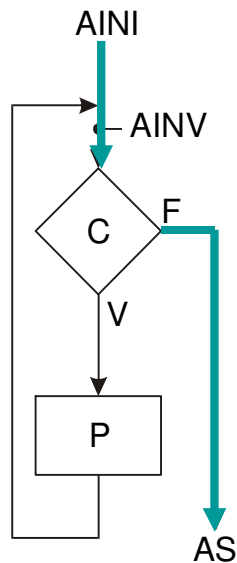


Devem ser argumentadas as proposições:

1. $AINI \Rightarrow AINV$

- a invariante vale ao iniciar

Proposições para a repetição



Devem ser argumentadas as proposições:

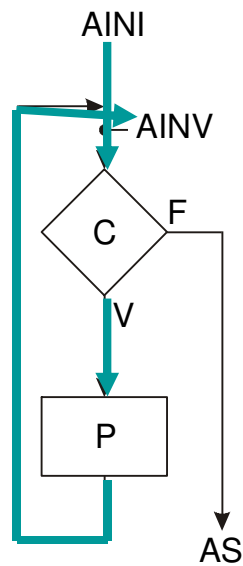
1. $AINI \Rightarrow AINV$

- a invariante vale ao iniciar

2. $AINI \ \&\& \ (C == F) \Rightarrow AS$

- executa corretamente 0 iterações

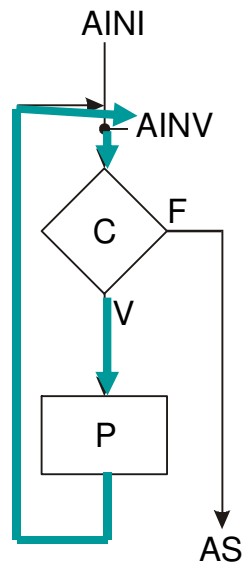
Proposições para a repetição



Devem ser argumentadas as proposições:

1. $AINI \Rightarrow AINV$
 - a invariante vale ao iniciar
2. $AINI \ \&\& \ (C == F) \Rightarrow AS$
 - executa corretamente 0 iterações
3. $AINI \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - executa corretamente a primeira iteração (base da indução)

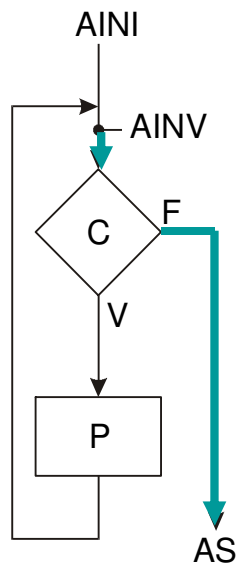
Proposições para a repetição



Devem ser argumentadas as proposições:

1. $AINI \Rightarrow AINV$
 - a invariante vale ao iniciar
2. $AINI \ \&\& \ (C == F) \Rightarrow AS$
 - executa corretamente 0 iterações
3. $AINI \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - executa corretamente a primeira iteração (base da indução)
4. $AINV \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - acrescenta corretamente mais uma iteração

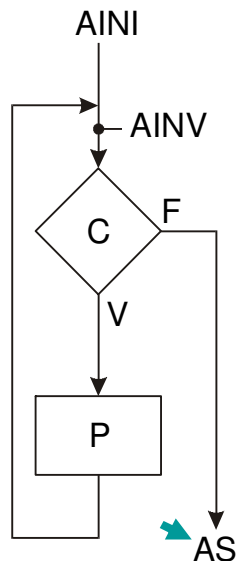
Proposições para a repetição



Devem ser argumentadas as proposições:

1. $AINI \Rightarrow AINV$
 - a invariante vale ao iniciar
2. $AINI \ \&\& \ (C == F) \Rightarrow AS$
 - executa corretamente 0 iterações
3. $AINI \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - executa corretamente a primeira iteração (base da indução)
4. $AINV \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - acrescenta corretamente mais uma iteração
5. $AINV \ \&\& \ (C == F) \Rightarrow AS$
 - sai correto após $n > 0$ iterações

Proposições para a repetição



Devem ser argumentadas as proposições:

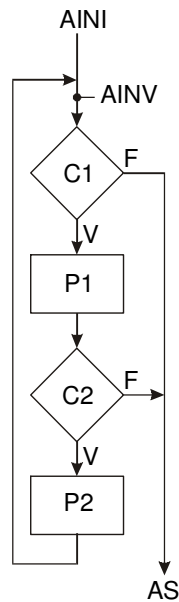
1. $AINI \Rightarrow AINV$
 - a invariante vale ao iniciar
2. $AINI \ \&\& \ (C == F) \Rightarrow AS$
 - executa corretamente 0 iterações
3. $AINI \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - executa corretamente a primeira iteração (base da indução)
4. $AINV \ \&\& \ (C == V) :: P \Rightarrow AINV$
 - acrescenta corretamente mais uma iteração
5. $AINV \ \&\& \ (C == F) \Rightarrow AS$
 - sai correto após $n > 0$ iterações
6. *término*
 - a repetição pára depois de um número finito de iterações

Argumentação de repetições



- Deve-se argumentar que o fragmento de **código pára**
- Potencialmente qualquer repetição pode executar um número **indefinido** (i.e. tendendo ao infinito) de iterações
 - mesmo que tenha sido explicitamente programada para terminar após um **número definido** de iterações
 - ex. `for (i = 0 ; i < 10 ; i++)`
- A argumentação de que a repetição pára deve ser separada da argumentação de que **progride corretamente**
 - isso permite mudar o limite da repetição sem precisar refazer toda a argumentação

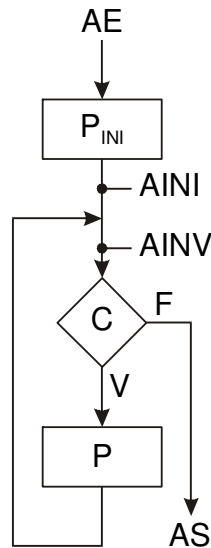
Proposições para a repetição



No caso de várias saídas de uma repetição (**break** ou **return**) devem ser argumentados, entre outros ajustes:

2. $AINI \ \&\& \ (C1 == F) \Rightarrow AS$,
 $AINI \ \&\& \ ((C1 == V) :: P1) \ \&\& \ (C2 == F) \Rightarrow AS$
 - executa corretamente 0 iterações
5. $AINV \ \&\& \ (C1 == F) \Rightarrow AS$,
 $AINV \ \&\& \ ((C1 == V) :: P1) \ \&\& \ (C2 == F) \Rightarrow AS$
 - sai correto após $n > 0$ iterações

Proposições para a repetição



- Todas as repetições requerem um **estado inicial**
 - de maneira geral o estabelecimento do estado inicial antecede imediatamente o código da repetição
- Temos então que argumentar mais uma proposição:

0. $AE :: P_{INI} \Rightarrow AINI$

- Também deve ser **verificado** se P contém o **avanço** para o próximo estado

Bibliografia



- Gries, D.; *The Science of Programming* ; New York: Springer; 1981
- Karaorman, M.; Hölzle, U.; Bruno, J.; *jContractor: A Reflective Java Library to Support Design by Contract*, www.cs.ucsb.edu/TRs/TRCS98-31.html
- Kneuper, R.; "Limits of Formal Methods"; *Formal Aspects of Computing* 9(4); Berlin: Springer; 1997; pages 379-394
- Kramer, R.; *iContract*, www.reliable-systems.com
- Meyer, B.; *Applying Design by Contract*; *IEEE Computer* 25(10); 1992; pages 40-51
- Meyer, B.; *Object-Oriented Software Construction*, 2nd edition, New Jersey: Prentice Hall; 1997
- Mitchell, R.; *Design by contract: Bringing together formal methods and software design*; University of Brighton; 2004
- Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008
- Staa, A.v.; *Programação Modular* ; Rio de Janeiro: Campus/Elsevier; 2000

Laboratório de Engenharia de Software

LES

FIM

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

59