



Instituto Federal de Educação, Ciência e Tecnologia da Bahia - IFBA
Departamento de Ciência da Computação
Tecnólogo em Análise e Desenvolvimento de Sistemas

Garantia de confiança e segurança

André L. R. Madureira <andre.madureira@ifba.edu.br>
Doutorando em Ciência da Computação (UFBA)
Mestre em Ciência da Computação (UFBA)
Engenheiro da Computação (UFBA)

Análise estática

- São técnicas de verificação da especificação e projeto de sistema que não envolvem a execução de um programa
 - **Objetivo:** encontrar erros antes que uma versão executável do sistema esteja disponível
- Ao contrário dos testes de software, técnicas de análise estática não são afetadas por defeitos no sistema
- **Ex:** revisão e inspeção, ferramentas de modelagem de projeto

Revisão e Inspeção

- Programadores verificam as especificações, o projeto ou o programa em busca de possíveis erros ou omissões.

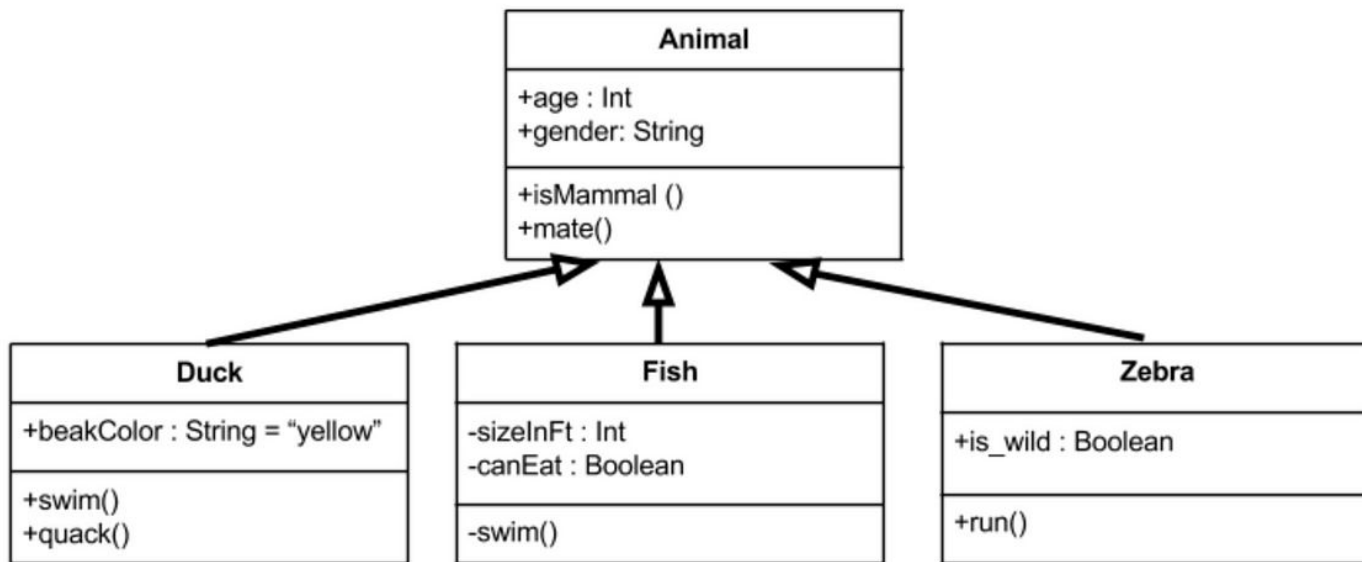


Geralmente se adota a **programação em pares (*pair programming*)** para essa etapa

Pair programming: equipes divididas em pares de programadores, que trabalham juntos para examinar, analisar e reestruturar o projeto ou código

Ferramentas de modelagem de projeto

- Verificam se existem anomalias na UML,
 - **Ex:** mesmo nome sendo usado para objetos diferentes



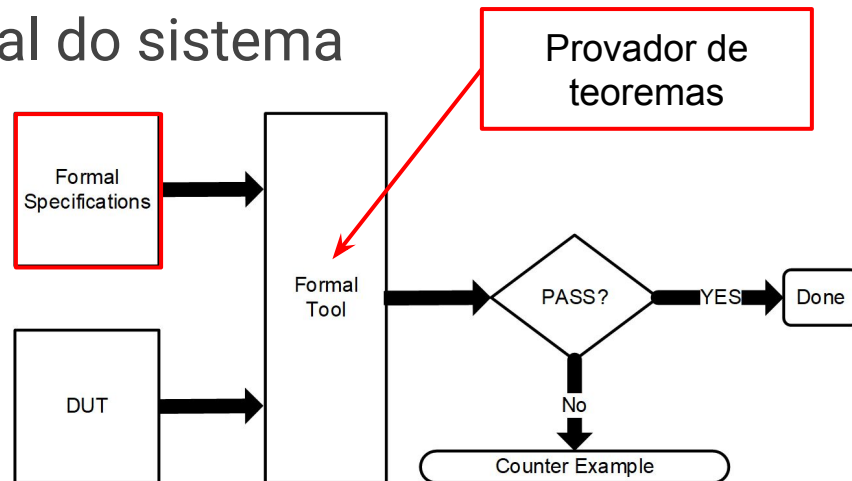
Análise estática

- Para sistemas críticos, técnicas adicionais de análise estática podem ser usadas
 - **Verificação formal**
 - **Verificação de modelos**
 - **Análise automatizada de programa**



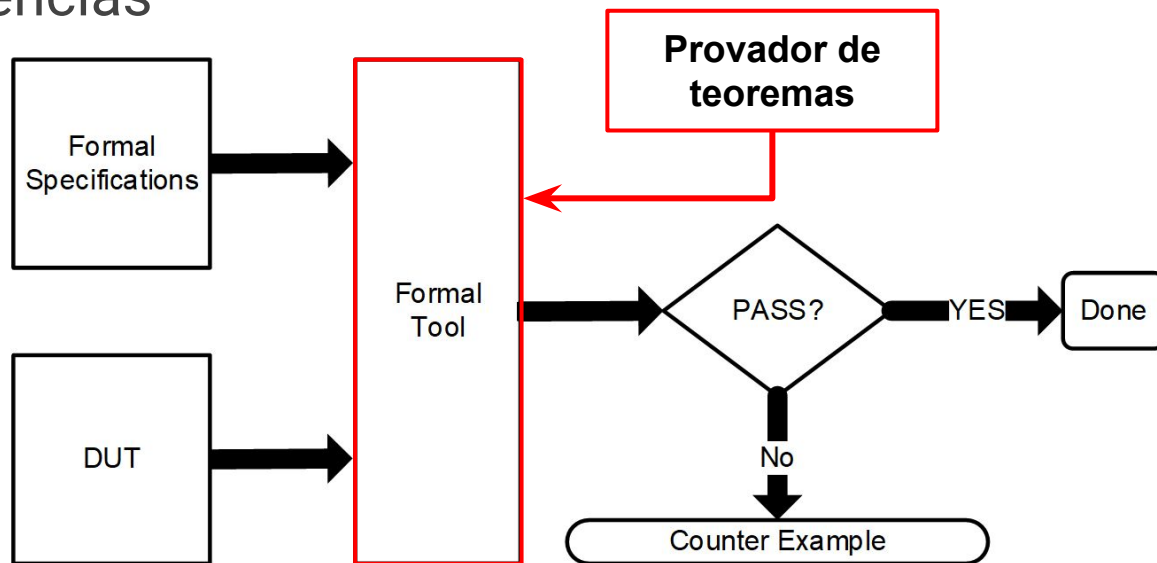
Verificação formal

- Define o conjunto de argumentos rigorosos (modelos matemáticos) que asseguram que o programa atende às suas especificações
 - **Resultado:** descrição formal do sistema
- A descrição é verificada pelo **prorador de teoremas**, que identifica inconsistências na especificação formal



Verificação de modelos

- Etapa onde um **prorador de teoremas** é usado para verificar uma descrição formal do sistema, para ver se existem inconsistências



Limitações da Verificação formal e de modelos

- Construir uma descrição formal do sistema é uma tarefa complexa
- A especificação e prova formais não garantem que o software será confiável na prática, pois:
 - A especificação pode não refletir os requisitos reais
 - Modelo formal \neq Especificação do sistema
 - A prova pode conter erros ou fazer suposições incorretas sobre a maneira como o sistema será usado

Por essas razões, raramente técnicas de verificação formal são utilizadas em softwares comerciais

Análise automatizada de programa

- O código-fonte de um programa é verificado por um outro software a procura de padrões conhecidos de erros potenciais
 - **Ex:** *code linting*

```
import fs from 'fs'
import writeGood from '../write-good'
●import annotate from 'annotate'

●if (files.length === 0) {
●  console.log('You did not provide any files to check');
●  proces
}
Unexpected console statement. (no-console)
```

Análise estática automatizada

- **Vantagem:** testes são construídos e executados rapidamente
- **Desvantagem:** não é possível identificar algumas classes de erros que poderiam ser identificados em revisões e inspeções do programa
 - Apesar disso, a técnica ainda é muito útil na detecção de **anomalias** e na construção de **testes de proteção**
 - **Anomalias:** omissões ou erros de programação

A análise estática automatizada também pode gerar falsos positivos. Isto é, detectar um erro que não necessária existe ou interfere no funcionamento normal do sistema

Exemplos de defeitos detectáveis pela análise estática automatizada

Classe de defeito	Verificação de análise estática
Defeitos de dados	Variáveis usadas antes da iniciação Variáveis declaradas, mas nunca usadas Variáveis atribuídas duas vezes, mas nunca usadas entre atribuições Possíveis violações de limites de vetor Variáveis não declaradas
Defeitos de controle	Código inacessível Ramos incondicionais dentro de <i>loops</i>
Defeitos de entrada/saída	Saída de variáveis duas vezes sem atribuição intermediária
Defeitos de interface	Incompatibilidades de tipo de parâmetro Incompatibilidades de número de parâmetros Não uso de resultados de funções Funções e procedimentos não chamados
Defeitos de gerenciamento de armazenamento	Ponteiros não atribuídos Ponteiro aritmético Perdas de memória

Exercício

Marque a alternativa que associa corretamente a descrição da técnica de análise estática (descrita nos numerais romanos abaixo) com o nome da técnica (descrita nas letras abaixo).

I - Conjunto de argumentos rigorosos (modelos matemáticos) que asseguram que o programa atende às suas especificações. **D**

II - O código-fonte de um programa é verificado por um outro software a procura de padrões conhecidos de erros potenciais. **C**

III - Programadores verificam as especificações, o projeto ou o programa em busca de possíveis erros ou omissões. **A**

IV - Etapa onde um provador de teoremas é usado para verificar uma descrição formal do sistema, para ver se existem inconsistências. **B**

A - Revisão e Inspeção

B - Verificação de modelos


C - Análise automatizada de programa

D - Verificação formal

☐ I-B, II-D, III-C, IV-A.

☐ I-B, II-C, III-A, IV-D.

☐ I-B, II-A, III-C, IV-D.

 ☒ I-D, II-C, III-A, IV-B.

☐ I-D, II-A, III-C, IV-B.

Análise estática automatizada

- Há três níveis de verificação que podem ser implementados
 - **Verificação de erros característicos**
 - **Verificação de erros definidos pelo usuário**
 - **Verificação de asserções**

Verificação de erros característicos

- Analisador estático detecta erros comuns
 - A ferramenta analisa o código, buscando por padrões que são característicos desses problemas
 - **Ex:** Variáveis sendo usadas antes de serem inicializadas
 - Erros são mostrados para o programador
- A técnica é muito efetiva na detecção de erros, sendo capaz de identificar 90% dos erros de programas C/C++ (Zheng et al., 2006)

Verificação de erros definidos pelo usuário

- Padrões de erro customizados podem ser fornecidos para o analisador estático, estendendo os tipos de erro que podem ser detectados
- Dessa forma, é possível assegurar que determinados requisitos de um sistema são sempre atendidos
 - **Ex:** método A deve ser chamado sempre antes do método B
 - **Ex:** variável C deve ser obrigatoriamente do tipo inteiro

Exemplo de Verificação de erros (característicos e definidos pelo usuário)

```
45
46 export const normalizeDate = (value) => {
47   ....if (isDate(value)) return value;
48
49   ....if (typeof value === 'string') {
50     ⚡ ....let valueToDate = parseISO(value);
```

You, a few seconds ago

⊗ dates.js 1 of 1 problem

'valueToDate' is never reassigned. Use 'const' instead. eslint(prefer-const)

```
51
52   ....if (isDate(valueToDate)) return valueToDate;
53   ....}
54
55   ....return null;
56 };
57
```


Verificação de asserções

- **Abordagem mais geral e mais poderosa da análise estática**
- Os desenvolvedores incluem **asserções formais** (condições) em seus programas, que devem ser válidas naquele ponto de programa
 - **Ex:** `assert C == 0`
(se a variável C for diferente de 0, temos uma falha na verificação da asserção)
- Permite construir testes de proteção automatizados
 - **Ex:** testes unitários automatizados (ex: JUnit, PHPUnit, unittest)

Verificação de asserções

Asserção
(escrita em Python)

```
def divide(x, y):  
    assert y != 0  
    return round(x/y, 2)  
  
z = divide(21,3)  
print(z)  
  
a = divide(21,0)  
print(a)
```

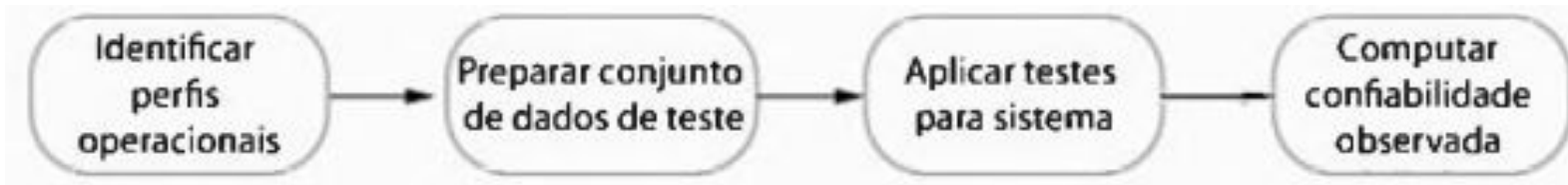
**O que essa asserção faz
no código?**

Garante que $y \neq 0$ após a
asserção ser executada

Se $y == 0$, ela levanta uma
Exceção no Python do tipo
AssertionError

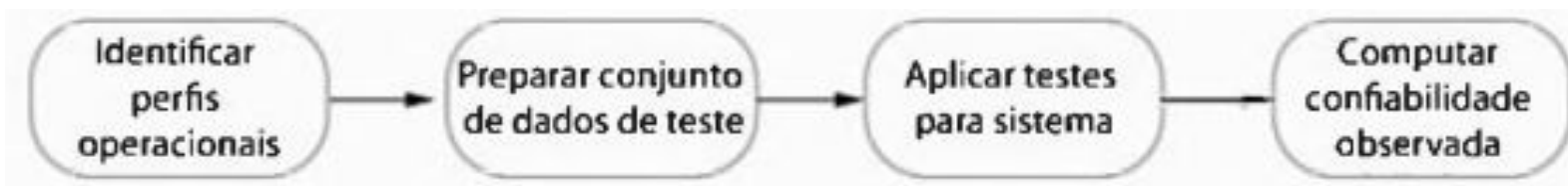
**Outras linguagens de
programação também
possuem esse recurso
(Java, C++, PHP, etc)**

Processo de medição de confiabilidade (ou Processo de teste estatístico)



Perfil operacional: identifica classes de entradas de sistema e a probabilidade de que essas entradas ocorram com uso normal.

Ex: um sistema A pode recebe um número inteiro X , que está entre $-\infty$ e $+\infty$, e realiza o calculo X^2 . Existe uma probabilidade de 10% de $X < 0$ e 90% de $X \geq 0$.
Logo, devemos projetar os testes do sistema pensando nessas probabilidades.



Identificação de perfis operacionais:

Estudar sistemas do mesmo tipo para compreender como eles são usados na prática.

Resultado: perfil operacional

Preparar conjunto de dados de teste:

Construir testes usando o perfil operacional do sistema para verificar o funcionamento deste em um cenário mais próximo do ambiente real de utilização do sistema.

Aplicar os testes:

Verificar quais testes falham, e quando essas falhas ocorrem para gerarmos estatísticas.

Computar confiabilidade:

Utilizando as estatísticas coletadas nos testes, calcular as métricas de confiabilidade (ex: *POFOD*, *ROCOF*, *AVAIL*)

Testes de proteção e Análise estática

- Para verificar a proteção de um sistema você pode usar uma combinação de técnicas de análise estática:
 - **Verificação formal**
 - **Testes baseados em experiência**
 - **Equipes tigre**
 - **Testes baseados em ferramentas**

Verificação formal

- Assim como a verificação formal de confiabilidade, podemos verificar o sistema contra uma especificação formal de proteção
 - Essa técnica é raramente utilizada, devido à:
 - Alta complexidade da especificação formal
 - Especificação formal diferente dos requisitos do sistema
 - Alto custo associado à construção da especificação e demais modelos formais

Testes baseados em experiência

- O sistema é analisado contra os tipos de ataques conhecidos pela equipe de validação
 - **Ex:** Ataque de *SQL Injection* (ou envenenamento SQL)
 - **Ex:** Negação de serviço (DoS)
- Checklists de verificação de problemas de proteção são construídos para ajudar a elaborar esses testes

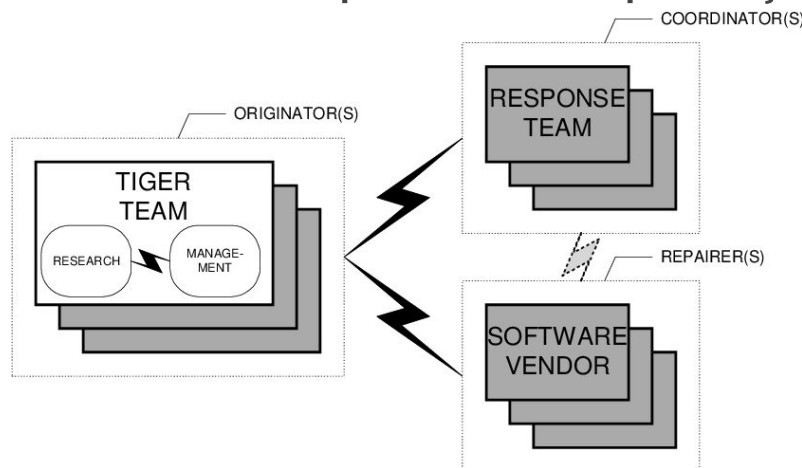
Exemplo de Checklist de Problemas de Proteção

Checklist de proteção

1. Todos os arquivos criados na aplicação têm permissões de acesso apropriadas? Quando erradas, as permissões de acesso podem levar ao acesso desses arquivos por usuários não autorizados.
2. O sistema automaticamente encerra as sessões de usuário depois de um período de inatividade? Sessões que ficam ativas podem permitir acesso não autorizado através de um computador não usado.
3. Se o sistema for escrito em uma linguagem de programação sem verificação de limites de vetor, existem situações em que o *overflow* de *buffer* pode ser explorado? O *overflow* de *buffer* pode permitir que invasores enviem e executem sequências de código para o sistema.
4. Se as senhas forem definidas, o sistema verifica se elas são 'fortes'? Senhas fortes consistem de pontos, números e letras misturados e não são entradas de dicionário normal. Elas são mais difíceis de quebrar do que senhas simples.
5. As entradas do ambiente do sistema são sempre verificadas por meio da comparação com uma especificação de entrada? O tratamento incorreto de entradas mal formadas é uma causa comum de vulnerabilidades de proteção.

Equipes tigras

- É uma forma de testes baseada em experiências, no qual designamos uma **equipe tigre** para testar a proteção do sistema
 - **Equipe tigre:** simula as ações de um atacante, para descobrir novas maneiras de comprometer a proteção do sistema



Testes baseados em ferramentas

- Várias ferramentas de proteção, como verificadores de senha, são usadas para analisar o sistema
 - **Ex:** verificadores de senha (ex: cracklib-check)
 - **Ex:** verificadores de *leak* ou falha na alocação de memória (ex: valgrind)
 - **Ex:** debugadores de código (ex: gdb)



cracklib-check

A library used to enforce strong passwords

Exercício

Considerando os testes de proteção de sistemas, marque a alternativa que contém **somente** as assertivas VERDADEIRAS.

I - A técnica das *equipes tigras* é uma das abordagens disponíveis para verificar a proteção de um sistema. Nela, a equipe simula as ações de um atacante visando maneiras de comprometer a proteção do sistema. **V**

II - A proteção de um sistema pode ser verificada através de técnicas de verificação formal, que são raramente utilizadas devido à alta complexidade da especificação formal. **V**

III - Nos testes baseados em experiência, checklists de verificação de problemas de proteção são construídos baseados nos tipos de ataques conhecidos pela equipe de validação. **V**

IV - Os testes baseados em ferramentas também podem ser utilizados para verificação de proteção de um sistema. Exemplos disso são os verificadores de senha, de *leaks* e debugadores de código. **V**

 ☐ Todas as assertivas são verdadeiras.

☐ Somente II e III.

☐ Somente II, III e IV.

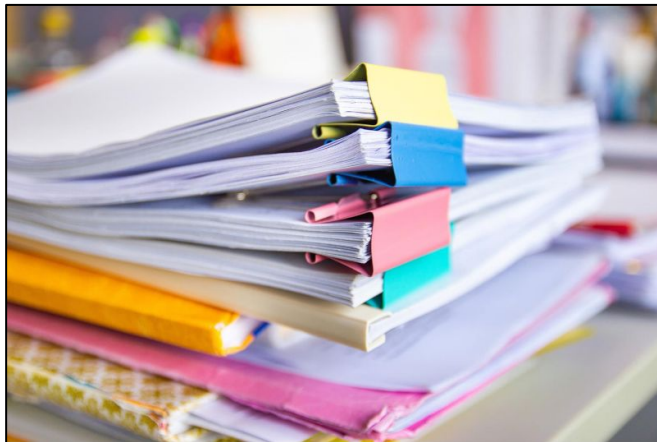
☐ Somente I, II e III.

☐ Somente I, III e IV.

Seguem as assertivas com os ajustes necessários para torná-las VERDADEIRAS:

Casos de segurança e confiança

- São documentos estruturados que unem todas as provas que demonstram que um sistema é confiável
 - Estabelecem **argumentos** e **evidências** de que um sistema é seguro (alcançou o nível exigido de proteção ou confiança)



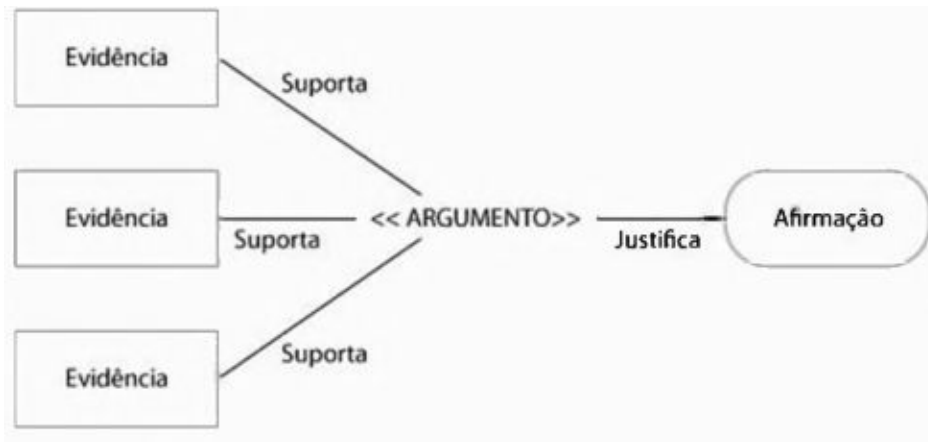
Casos de segurança e confiança

- Relaciona as falhas do software com as do sistema
- Demonstra que as falhas de software não ocorrerão ou não serão propagadas de forma que possam ocorrer falhas perigosas
 - **Ex:** explosão de uma usina nuclear
- Para muitos tipos de sistema críticos, a produção de um caso de segurança é um requisito legal
 - **Ex:** sistema de controle de uma aeronave comercial

Capítulo	Descrição
Descrição de sistema	Uma visão geral do sistema e uma descrição de seus componentes essenciais.
Requisitos de segurança	Os requisitos de segurança abstraídos da especificação de requisitos de sistema. Detalhes de outros requisitos relevantes de sistema também podem ser incluídos.
Análise de perigos e riscos	Documentos que descrevem os perigos e os riscos identificados e as medidas tomadas para reduzir os riscos. Análises de risco e logs de perigos.
Análise de projeto	Um conjunto de argumentos estruturados (ver Seção 15.5.1) que justifique por que o projeto é seguro.
Verificação e validação	Uma descrição dos procedimentos de V & V usados e, se for o caso, os planos de teste para o sistema. Resumos dos resultados de testes mostrando defeitos que foram detectados e corrigidos. Se foram usados métodos formais, uma especificação formal de sistema e quaisquer análises dessa especificação. Registros de análises estáticas do código-fonte.
Relatórios de revisão	Registros de todas as revisões de projeto e segurança.
Competências de equipe	Evidência da competência de todos da equipe envolvida no desenvolvimento e validação de sistemas relacionados com a segurança.
Processo de QA	Registros dos processos de garantia de qualidade (ver Capítulo 24) efetuados durante o desenvolvimento de sistema.
Processos de gerenciamento de mudanças	Registros de todas as mudanças propostas, ações tomadas e, se for o caso, justificativa da segurança dessas mudanças. Informações sobre os procedimentos de gerenciamento de configuração e logs de gerenciamento de configuração.
Casos de segurança associados	Referências a outros casos de segurança que podem afetar o processo de segurança.

Argumentos estruturados

- Para podermos afirmar que um sistema é seguro, precisamos ter **argumentos lógicos**
 - **Argumento:** é um relacionamento entre o que acreditamos ser verdade (a afirmação) e um corpo de evidências que prova essa afirmação



Exemplo de Argumentos estruturados

- **Ex:** sistema de bomba de insulina
 - **Afirmação:** o sistema não excederá a dose segura para um paciente em particular.
 - **Evidência:** conjuntos de testes feitos com o sistema
 - **Evidência:** relatório de análise estática do software

Hierarquia de Afirmações

- É possível termos uma afirmação dependendo de outras.
 - Isto é, uma afirmação só é verdadeira **se todas as outras** às quais ela depende **também são verdadeiras**
- Nesse caso, precisamos construir uma **hierarquia de afirmações** para verificar a validade dessas afirmações
 - *As afirmações de nível superior são válidas, se todas as afirmações inferiores (“folhas da árvore”) também forem válidas*

Exemplo de Hierarquia de Afirmações do Sistema da Bomba de Insulina

As afirmações de nível superior são válidas, se **todas** as afirmações inferiores também forem.

Afirmações de nível alto

A bomba de insulina não vai entregar uma única dose de insulina não segura

A dose máxima única, calculada pelo software da bomba, não deve exceder a maxDose

A maxDose é definida corretamente quando a bomba está configurada

A maxDose é uma dose segura para o usuário da bomba de insulina

Afirmações de nível inferior

Em uma operação normal, a dose máxima calculada não excede a maxDose

Se o software falhar, a dose máxima calculada não excederá a maxDose

Argumentos estruturados

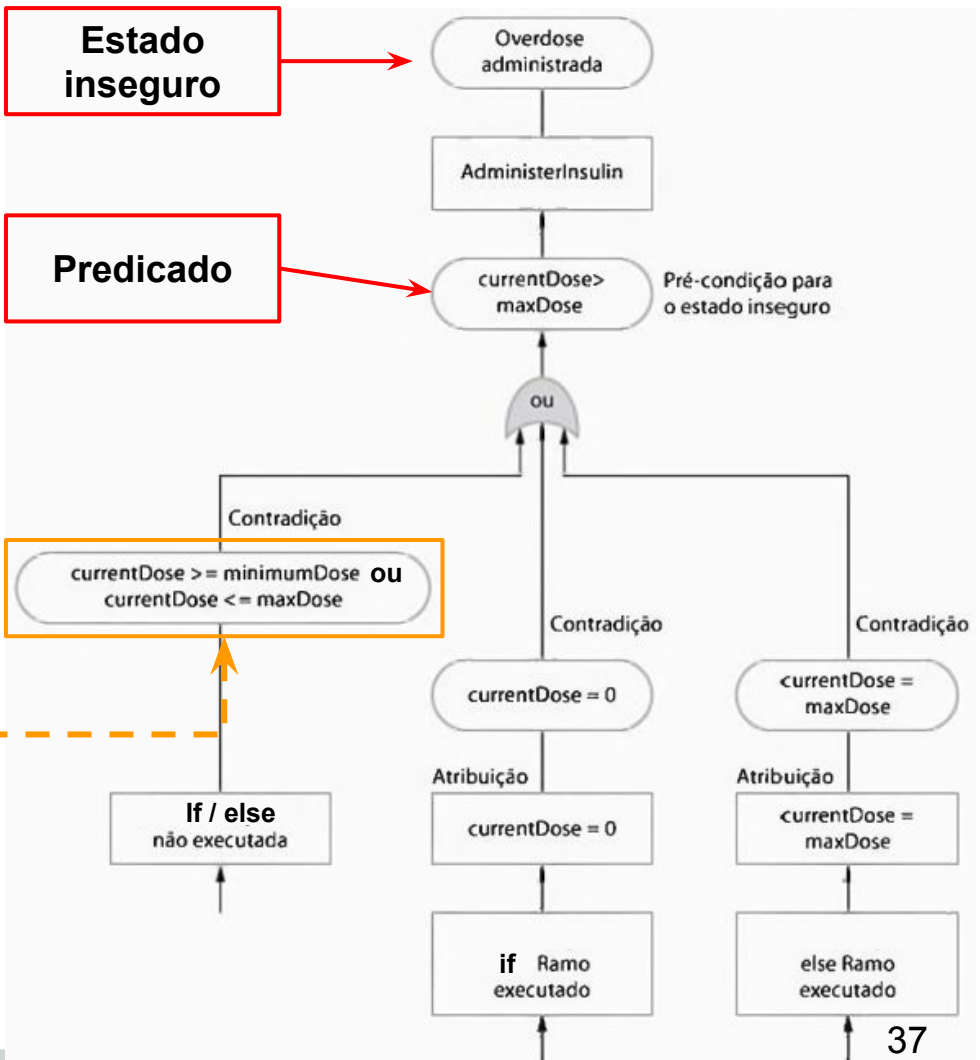
- O argumento estruturado é construído como uma **prova por contradição** da matemática, através dos seguintes passos:
 - Identificar um estado inseguro do sistema
 - Escrever um **predicado** (expressão lógica) que defina esse estado inseguro
 - Analisar sistematicamente o modelo do sistema/programa e tentar mostrar que existe pelo menos um caminho do programa que leva o sistema até esse estado
 - Se não conseguimos mostrar que existe ao menos um caminho, sabemos que a hipótese do estado inseguro está incorreta

Exemplo de Argumento Estruturado para o Sistema de Bomba de Insulina

- **Afirmação:** o sistema nunca entrega uma dose maior que maxDose, considerada segura para os pacientes diabéticos
- **Prova:** demonstrar que o sistema nunca causa overdose no paciente
 - **Predicado (estado inseguro):** $\text{currentDose} > \text{maxDose}$
 - Temos que demonstrar que todos os caminhos do programa conduzem a uma contradição a essa asserção não segura

Código-fonte do sistema

```
if (currentDose < minDose) {
    currentDose = 0
} else if (currentDose > maxDose) {
    currentDose = maxDose
} -----
administerInsulin(currentDose)
```

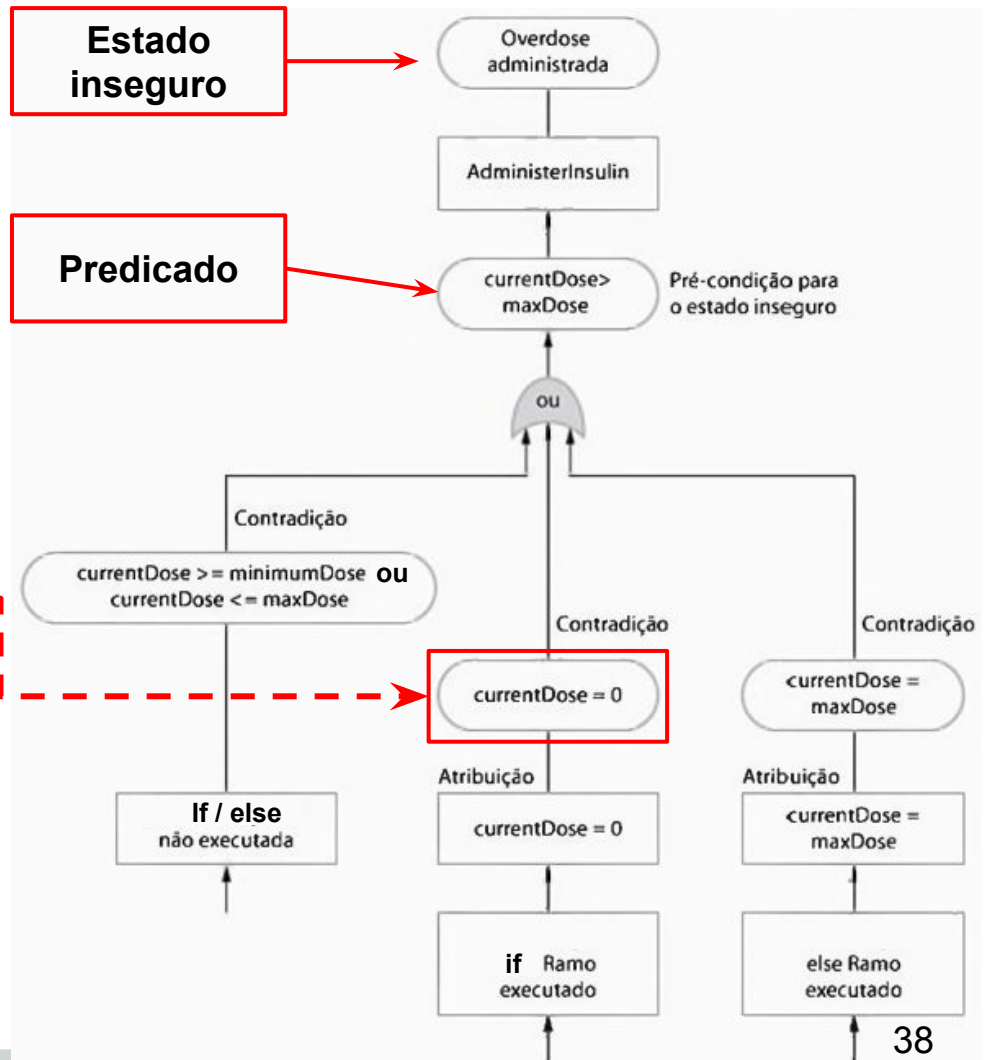


Código-fonte do sistema

```
if (currentDose < minDose) {  
    currentDose = 0  
} else if (currentDose > maxDose) {  
    currentDose = maxDose  
}  
administerInsulin(currentDose)
```

Estado
inseguro

Predicado

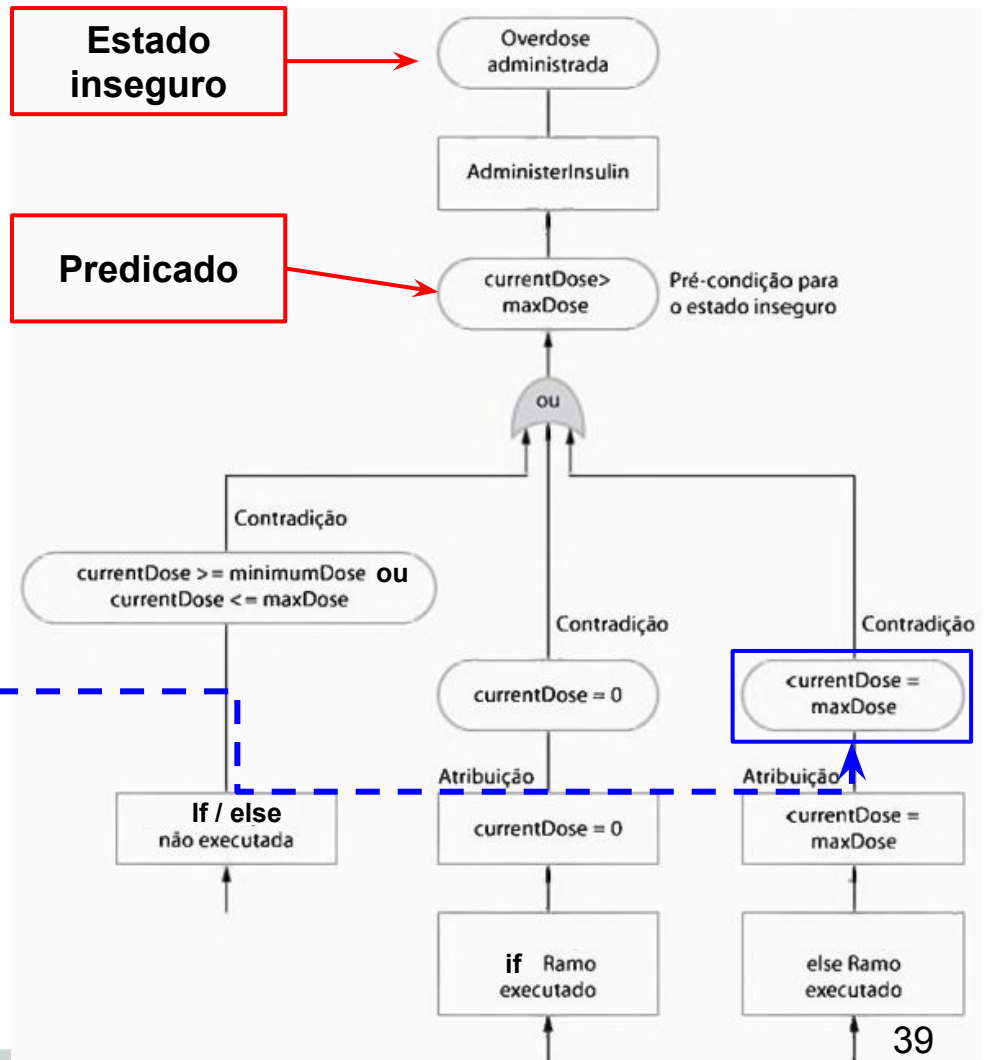


Código-fonte do sistema

```
if (currentDose < minDose) {  
    currentDose = 0  
} else if (currentDose > maxDose) {  
    currentDose = maxDose  
}  
administerInsulin(currentDose)
```

Estado inseguro

Predicado



Exercício

Marque a alternativa que contém **somente** as assertivas VERDADEIRAS.

I - Uma hierarquia de afirmações deve ser construída quando temos uma afirmação dependendo de outras. **V**

II - Em uma hierarquia de afirmações, as afirmações superiores são válidas somente se todas as afirmações inferiores também forem. **V**

III - Um argumento estruturado é concebido como uma prova por contradição da matemática, que visa demonstrar que um estado inseguro nunca é alcançado pelo sistema. **V**

IV - Um argumento estruturado deve demonstrar que todos os caminhos do programa conduzem a uma contradição ao predicado, que leva o sistema um estado inseguro. **V**

 ☐ Todas as assertivas são verdadeiras.

☐ Somente I, II e III.

☐ Somente I, II e IV.

☐ Somente I e IV.

☐ Somente III.

Seguem as assertivas com os ajustes necessários para torná-las VERDADEIRAS:

Referencial Bibliográfico

- SOMMERVILLE, Ian. **Engenharia de Software**. 6. ed. São Paulo: Addison-Wesley, 2003.
- PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 1995.
- JUNIOR, H. E. **Engenharia de Software na Prática**. Novatec, 2010.

Obrigado!

- Perguntas?

