DIM0436

27. Testes de caixa branca
Mutações

20141106

Sumário

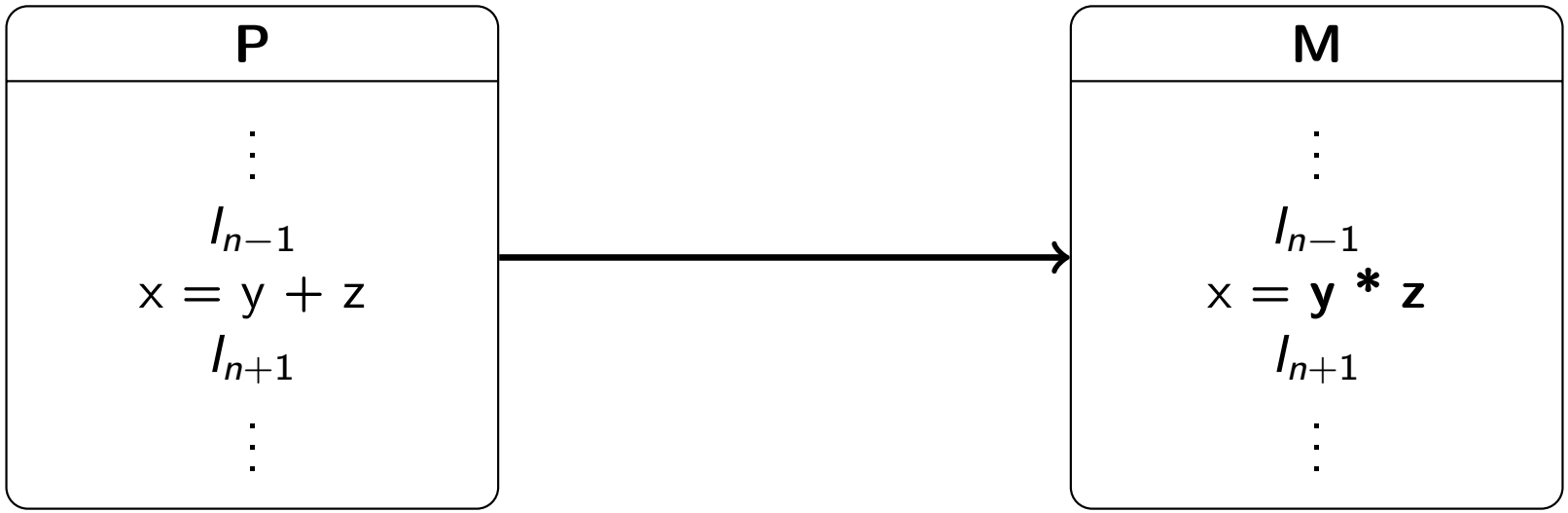
Ideia

Princípio (mutações fortes)

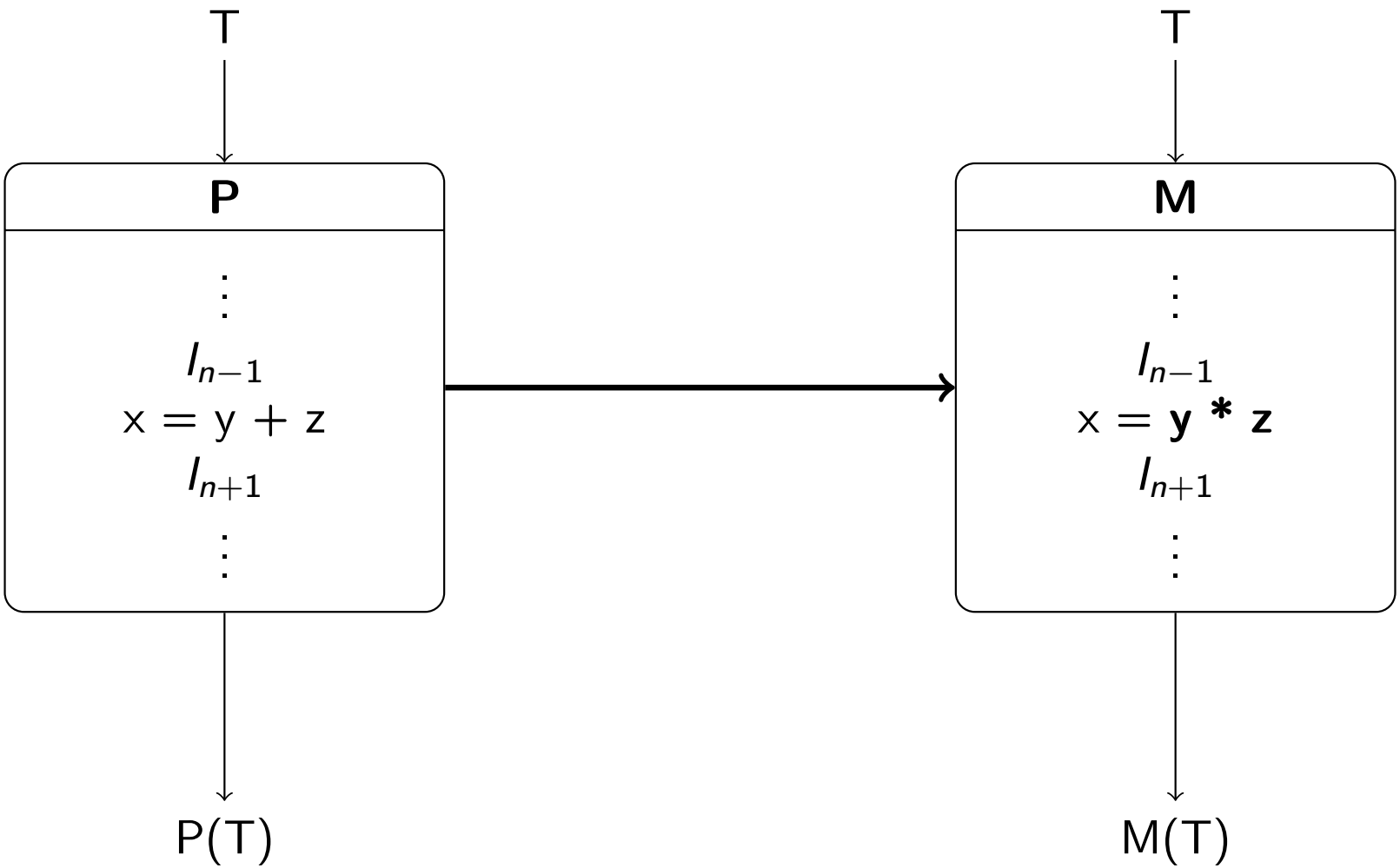
- Modificar o programa P em um programa P' injetando uma modificação **sintaticamente correta**, i.e. P' ainda compila.
- Idealmente, o comportamento de P' é diferente do de P
- Selecionar um dado de teste que destaque essa diferença de comportamento.

eliminar o mutante P'

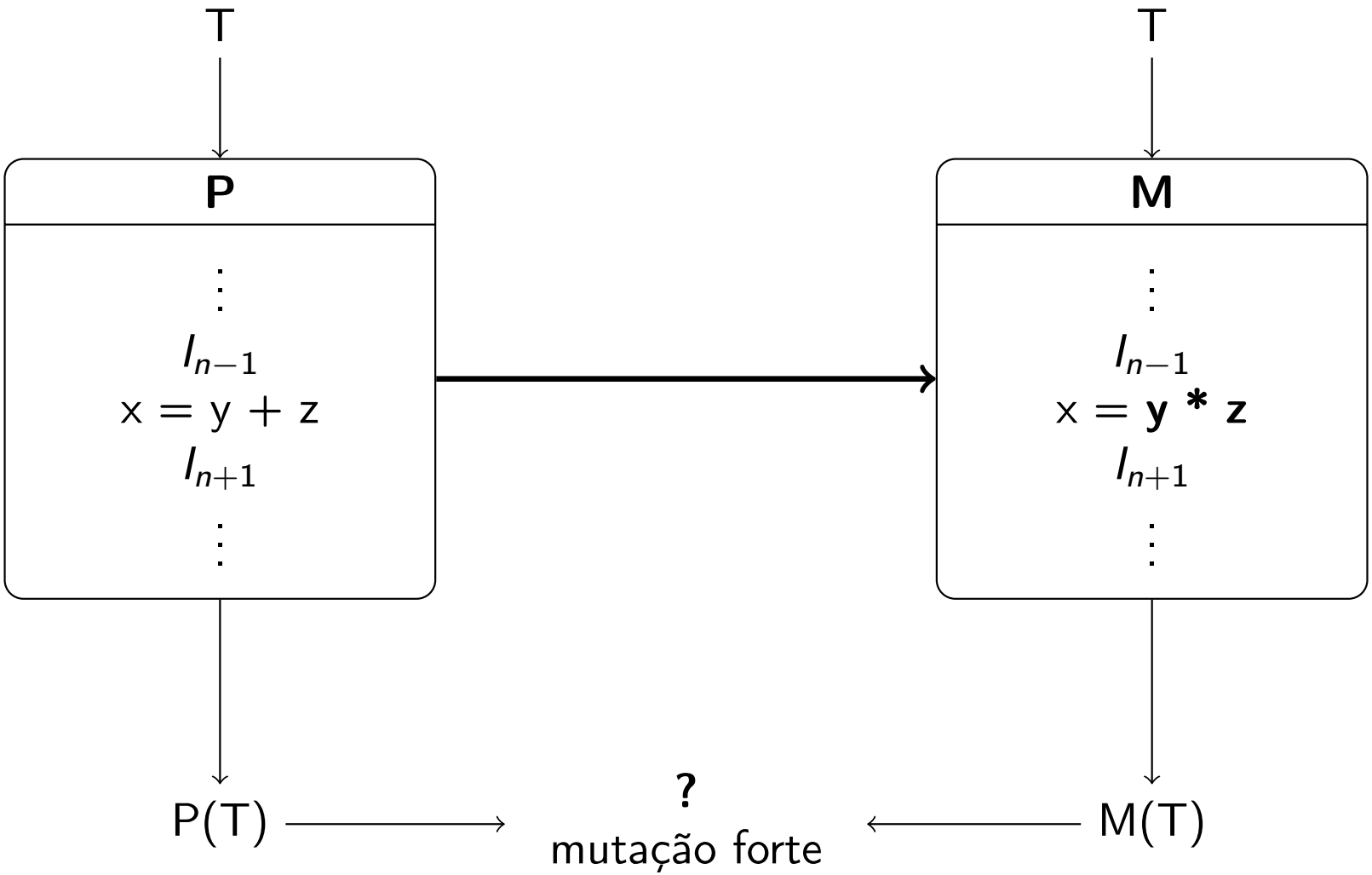
Ilustração



Ilustração



Ilustração



Vocabulário

- **Mutação** de P: modificação sintática de P
- **Mutante** de P: programa P' obtido por mutação de P
- TS **elimina** P: $\exists t \in TS, P(t) \neq P(t')$
- Resultado de mutação de TS: # mutantes eliminados / # mutantes

Detectar um mutante

Para detectar um mutante, precisamos cumprir as 3 condições seguintes:

Alcançabilidade P(t) alcança a mutação

Infeção P(t) $\neq$ P'(t) logo depois da mutação

Propagação P(t) $\neq$ P'(t) até o fim do programa

Exemplos de mutações

Seja a instrução `if (a > 8) x = y + 1`
Pode-se considerar os mutantes abaixo:

1. `if (a < 8) x = y + 1`
2. `if (a >= 8) x = y + 1`
3. `if (a > 8) x = y - 1`
4. `if (a > 8) x = y`

Observação

Para um programa dado, consideraremos um grande número de mutantes.

Operadores de mutação

Operadores clássicos de mutação [AO08]

- **bomb** : adicionar `assert(false)` após uma instrução
- **ABS**: modificar um expressão aritmética e em $|e|$
- **ROR**: substituir um operador relacional por um outro
- **AOR**: substituir um operador aritmético por um outro
- **COR**: substituir um operador lógico por um outro
- **UOI**: inverter uma expressão aritmética/lógica por $-$ ou $\neg$
- substituir o nome de uma variável por um outro
- substituir o nome de uma variável por uma constante (do bom tipo)
- substituir uma constante por uma outra constante (do bom tipo)
- ...

Algumas regras para mutantes

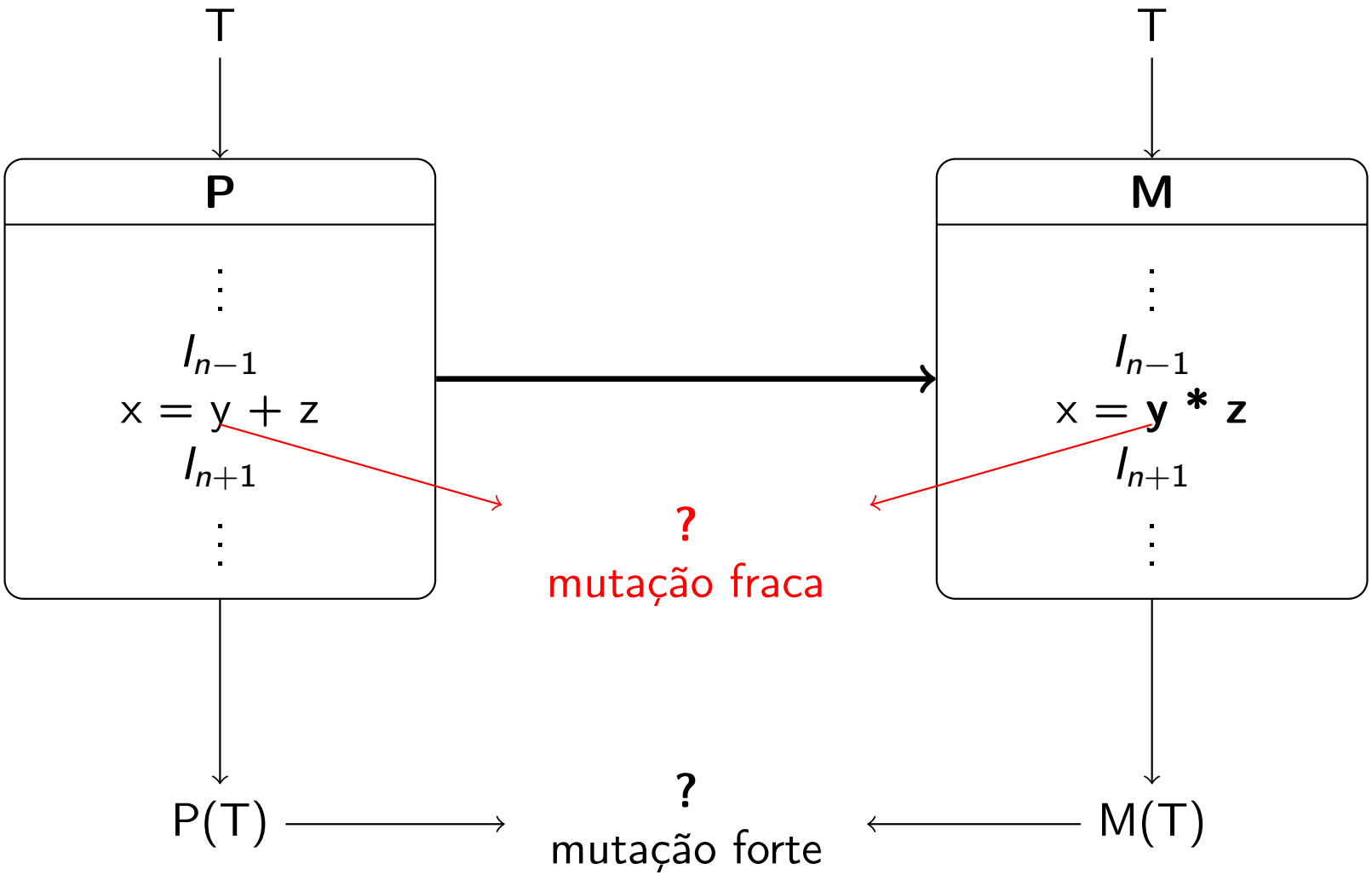
Correção

- Operadores de mutação devem preservar a correção sintática é a tipagem
- Caso contrário, o mutante é trivialmente distinguido : o programa não compila!

Diferença simples

Consideramos mutantes que diferem do programa original por um instrução só (mutação atômica, mutantes de ordem 1)

Mutações fracas (wM)



Interesse das mutações fracas

Ponto de vista teórica

Mais simples para comparar aos critérios estruturais usuais

Ponto de vista prática

- Mais simples de achar os dados de teste
- Só precisa respeitar a fase de alcançabilidade e de infeção
- Não precisamos seguir a fase de infeção

Comparação

- teórica $sM \succ wM$
- prática $sM \approx wM$

Expressividade

Notação

- Seja O um operador de mutação $P \rightarrow P'^{+}$
- $M(O)$ = cobertura dos mutantes criados por O
- $sM(O)$ / $wM(O)$

Teorema

Para todo critério $\mathcal{C} \in \{I, D, C, DC, MC\}$,

1. $\exists O(\mathcal{C}), sM(O(\mathcal{C})) \succeq \mathcal{C}$
2. $\exists O(\mathcal{C}), wM(O(\mathcal{C})) = \mathcal{C}$

Propriedades

Mutações fracas permitem simular critérios estruturais usais.

- wM $\succeq$ MC
- wM $\neq$ all-paths
- wM $\succ$ MCDC
- wM $\succ$ all-def
- wM $\succ$ all-use
- wM $\neq$ all-du-all-paths
- wM $\neq$ all-du-one-path ($\succ$ em prática)

Em prática

Criar os mutantes

$|M| \approx |P|$

Compilar os mutantes

- $|M|$ compilações de um programa de tamanho $|P|$
- 1 compilação / mutante
- D: 1 compilação

Calcular o resultado

- $(|M| + 1) \times |T|$ execuções
- Cada teste dever executado contra cada mutante e o programa inicial
- D: $|T|$ execuções

Melhoria

Representação baixo nível

- Aplicar mutações à representação baixo nível
- 1 compilação só!
- limitada ao *bytecode* (Java, .NET, . . .)

Meta-mutantes

- Agrupar múltiplas mutações numa só
- 1 compilação mas código grande
- sempre funciona

Exemplo de meta-mutantes

Programa original

```
1  void foo(t1 arg1, t2 arg2) {
2    /* ... */
3    x += y;
4    a = b + 1;
5    /* ... */
6  }
```

Meta-mutante

```
1  void foo(t1 arg1, t2 arg2, int mid)
2  {
3          /* ... */
4          switch (mid) {
5          case 1: x *= y; break;
6          case 2: x -= y; break;
7          default: x += y;
8          };
9          switch (mid) {
10         case 1: a = b - 1; break;
11         case 2: a = b; break;
12         default: a = b + 1;
13         }
14         /* ... */
15 }
```

Melhoria: cálculo do resultado

Cálculo incremental

- Um mutante eliminado não é reusado contra os testes restantes
- Em prática, menos reusos
  - os primeiros testes eliminam muitos mutantes "fáceis"
- Informação perdida
  - só o resultado importa
  - não o número de mutantes eliminados por teste
  - torna-se impossível minimizar o conjunto de testes

Achar os dados de teste

- Muitas vezes, 90% dos mutantes simples a serem cobertos ($\approx$ D)
- `x = y + 1` $\mapsto$ `x = y` sempre eliminado se mutação alcançada

Dificuldade

É difícil cobrir o resto . . . mas são eles os mais importantes e que constituem a riqueza da abordagem.

Escolha dos operadores de mutação

Operadores de mutação baseados em geral sobre:

- critérios de cobertura estrutural (bomb)
- cobrir domínios das variáveis do programa (abs)
- modelos de falhas simples usuais

O que fazer ?

- Usar só mutantes de ordem 1
- Privilegiar os operadores os mais eficientes

Eficiência dos operadores

$\succeq$

Um operador O é mais potente que um outro operador O' se $M(O) \succeq M(O')$

Dados experimentais

- ABS + ROR : 97% da eficiência
- ABS + ROR + AOR + COR + UOI : 99% da eficiência

Equivalência de mutantes

Às vezes, a mutação não pode ser distinguida, i.e. $\nexists t\ P(t) \neq P(t')$

- mutação não acessível
- infeção impossível ($x + 0 \mapsto x - 0$)
- propagação impossível (valor não usado)

Nesse caso, falamos de **mutantes equivalentes**

Problema

Mutantes equivalentes atrapalham o método

- resultado de mutação baixo
- esforço para cobrir um mutante sem poder fazê-lo

Detecção

- Detectar mutantes equivalentes é **indecidível**
- Análises podem detectar alguns (código morte, análise valor, dependências, prova de teoremas, ...)

Sobre mutantes

Vantagens

- Teóricas: mais potente que os critérios de cobertura estrutural
- Práticas: boa correlação com descoberta de bugs

Custo

- Muitos mutantes necessários ($\approx |P|$)
  - muitos fáceis ($\leq 90\%$)
  - eficiência vem da cobertura dos últimos
- Difícil mas qualidade dos testes
- Avaliar o resultado é custoso
- Melhor como critério de qualidade

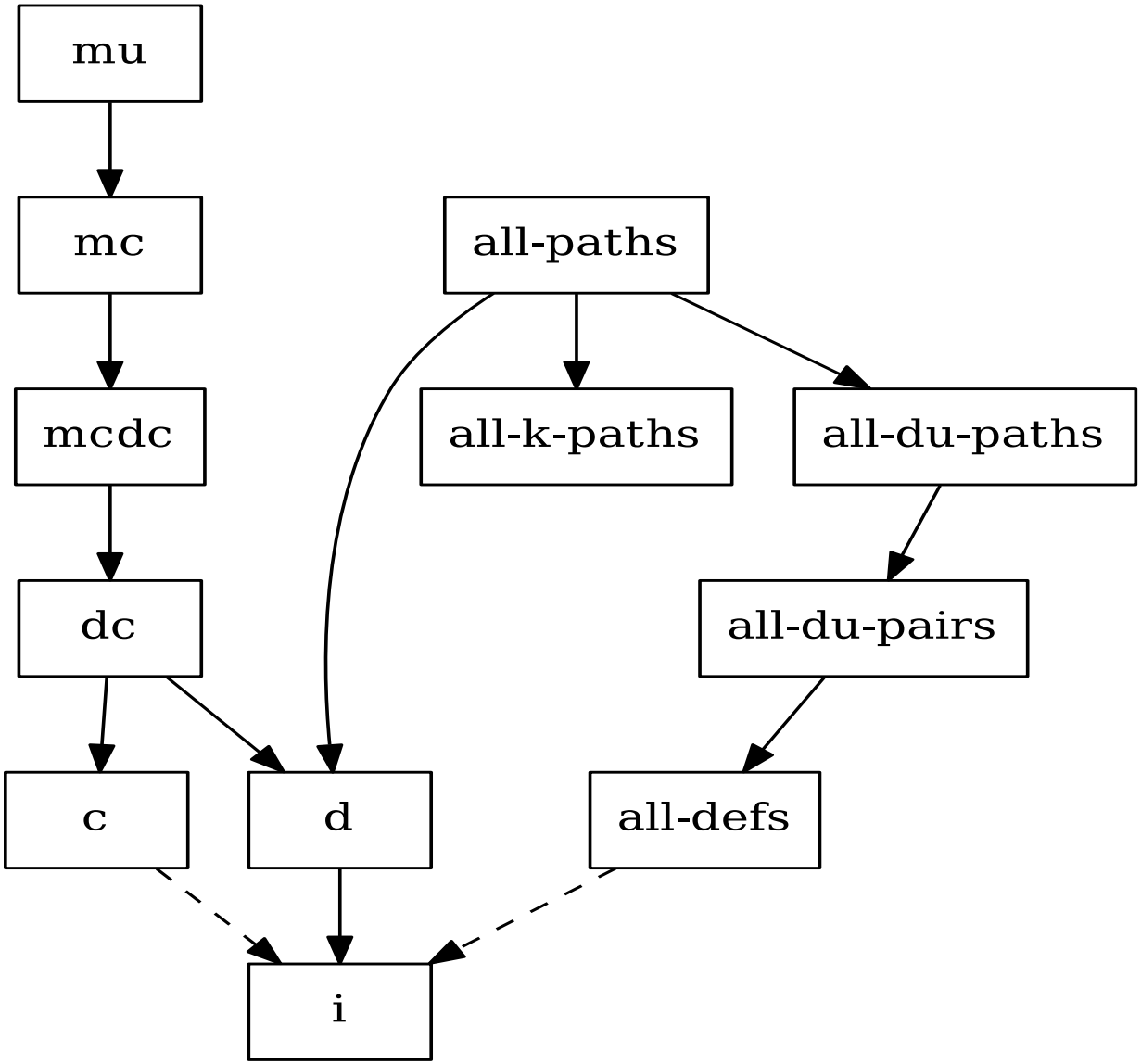
Mutações em prática

- Raramente usada hoje
- Critério fino mas custoso
- Ferramentas (calcular o resultado):
  - Proteum (C)
  - mujava/muclipse (Java)

Uso prático

- Começar por I, D e/ou domínios de entrada
- Já eliminará muitos mutantes

Hierarquia dos critérios



Resumo

1. Princípios
2. Expressividade
3. Aspeitos práticos
4. Operadores de mutação

Referências

Paul Ammann and Jeff Offutt, *Introduction to software testing*, 1 ed., Cambridge University Press, New York, NY, USA, 2008.

Glenford J. Myers and Corey Sandler, *The art of software testing*, 3 ed., John Wiley & Sons, 2004.

Perguntas ?

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