SSC 0125 – Verificação Validação e Teste de Software

Teste de Mutação

Prof. Marcio E. Delamaro

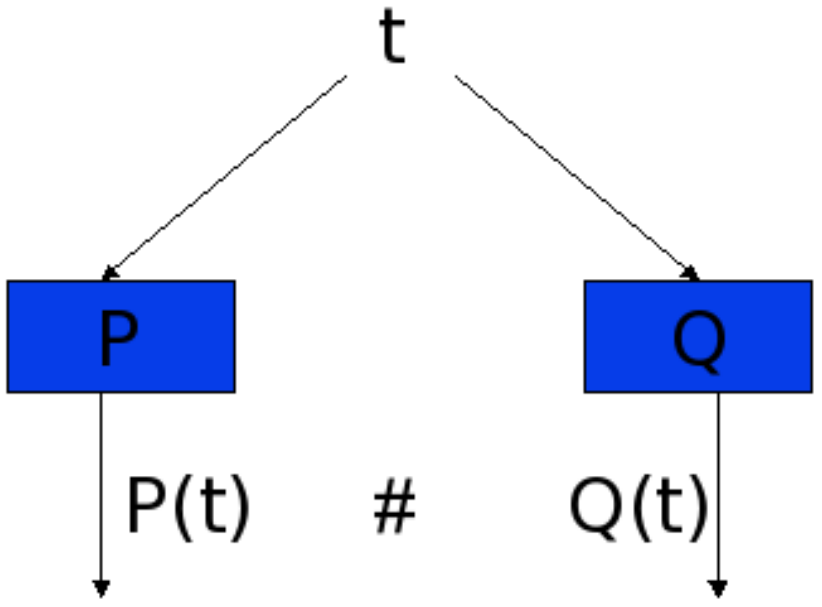
delamaro@icmc.usp.br

Teste baseado em defeitos

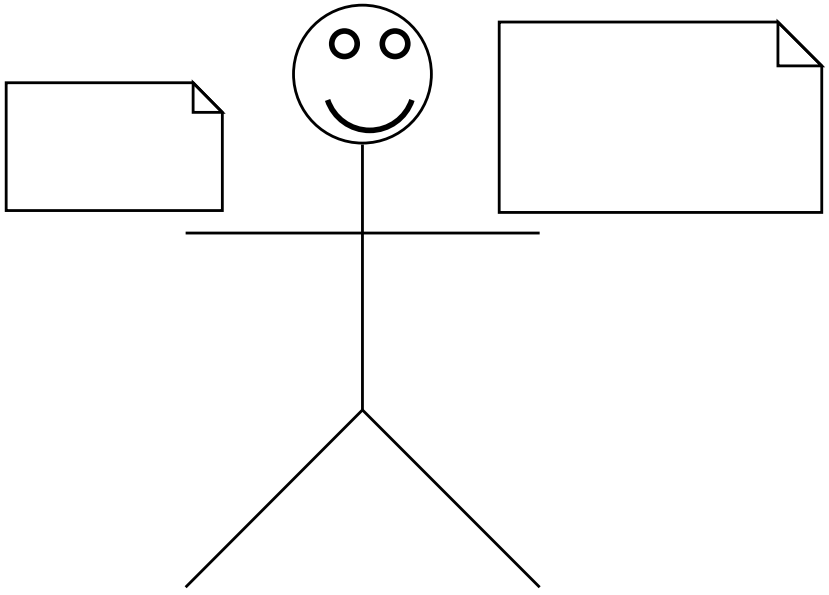
- Utiliza informação sobre defeitos comuns
- Teste de mutação é o critério mais conhecido

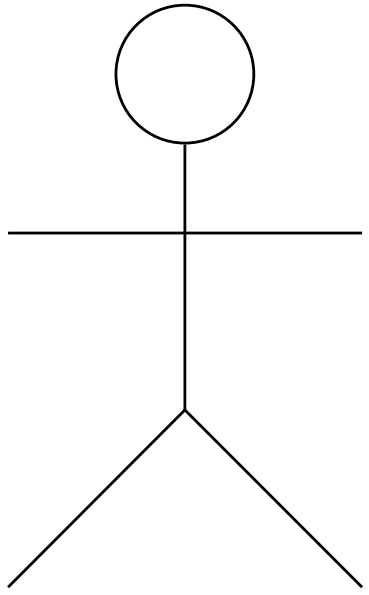
Teste de mutação

- A idéia por trás do teste de mutação é mostrar que o programa em teste não possui determinados tipos de defeitos

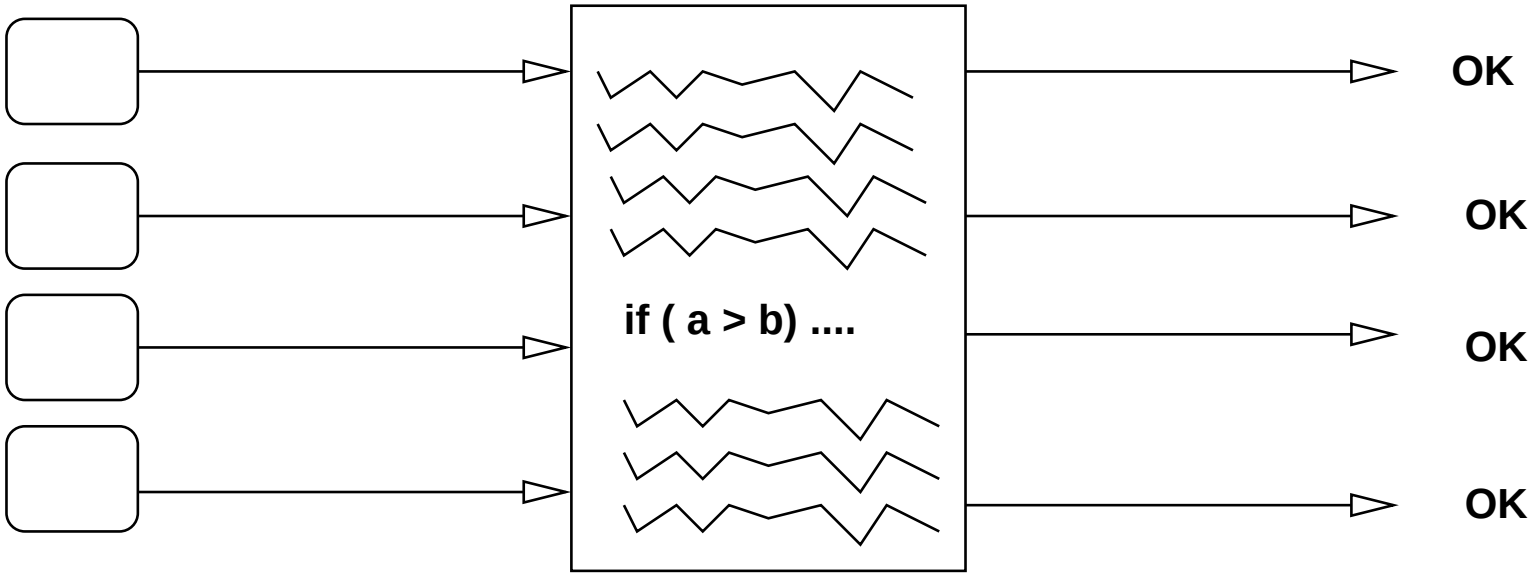


Qual a idéia?

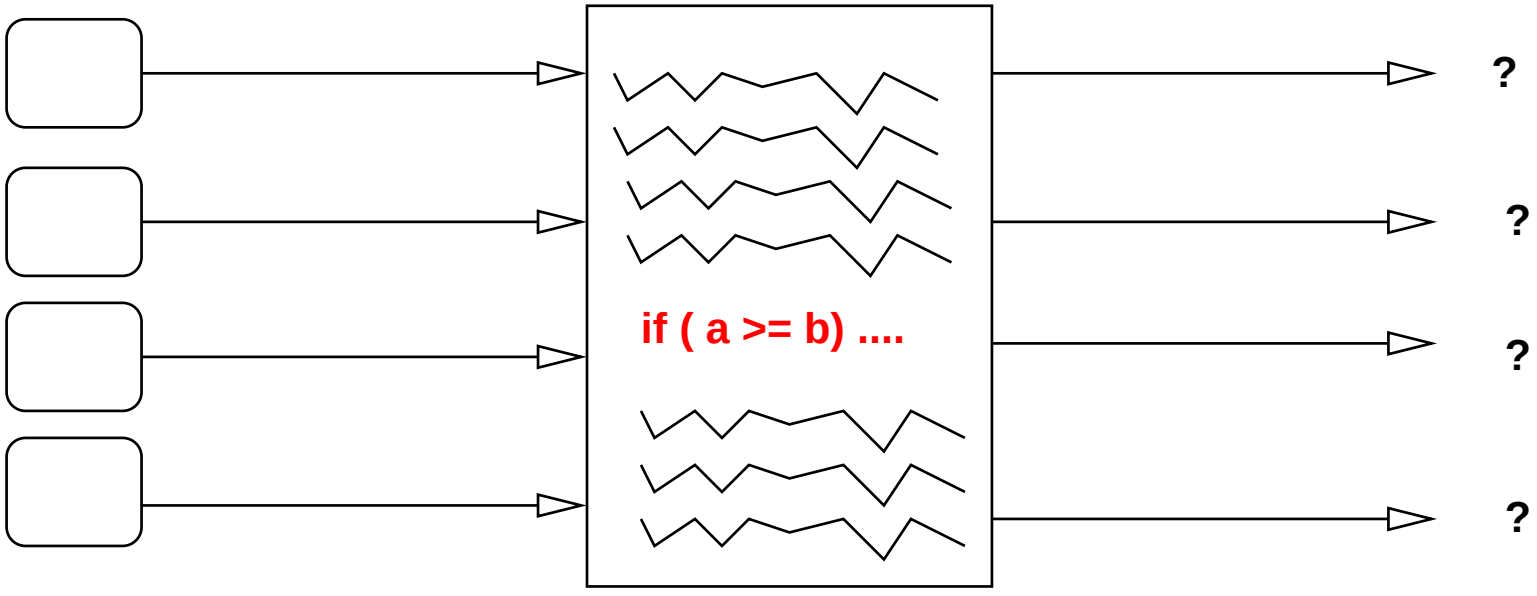




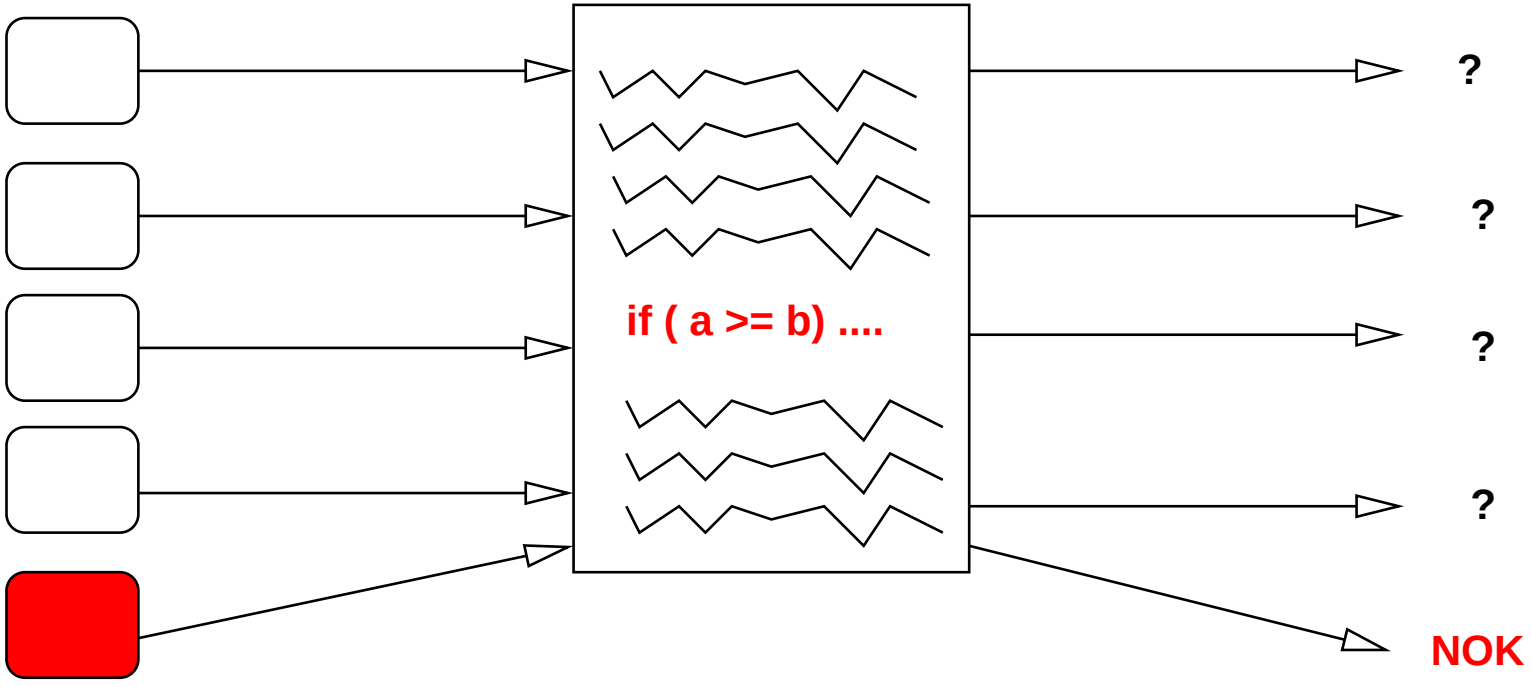
Qual a idéia?



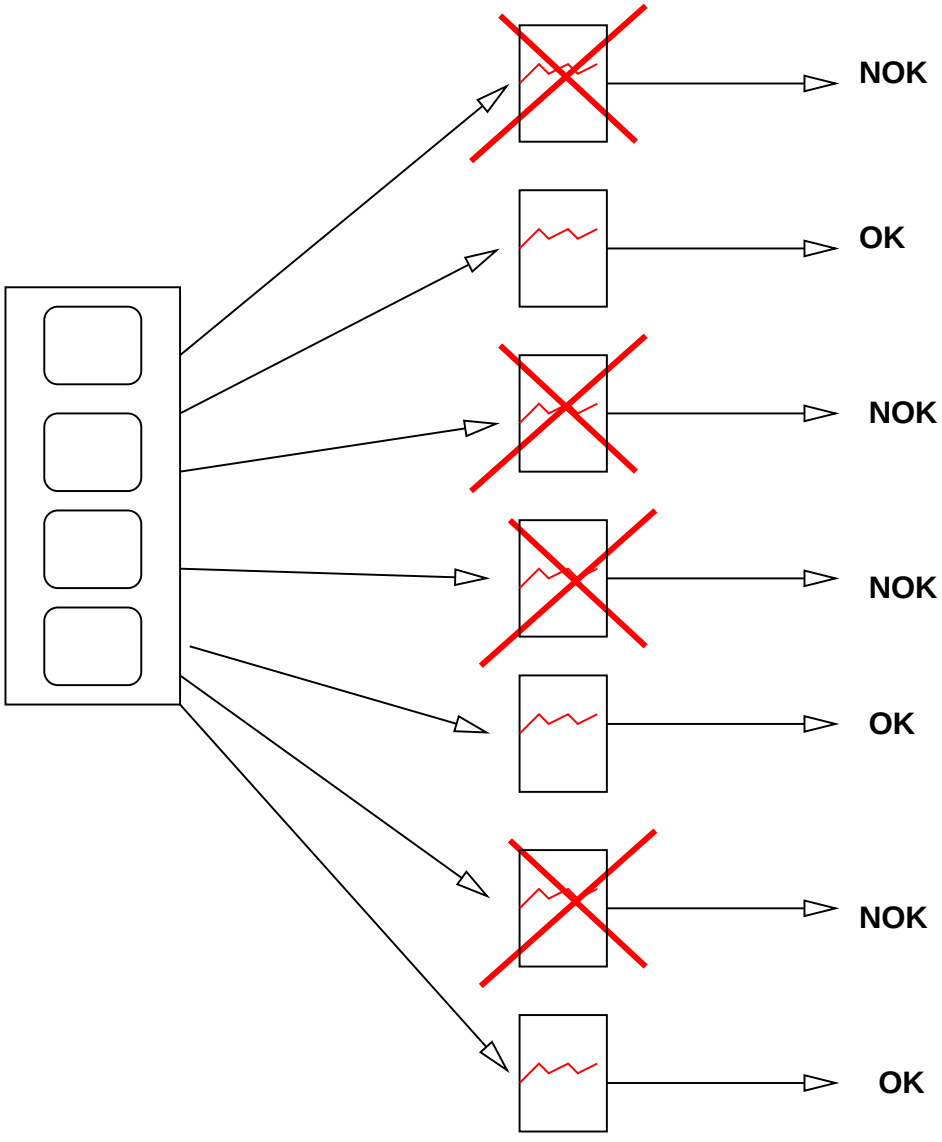
Qual a idéia?



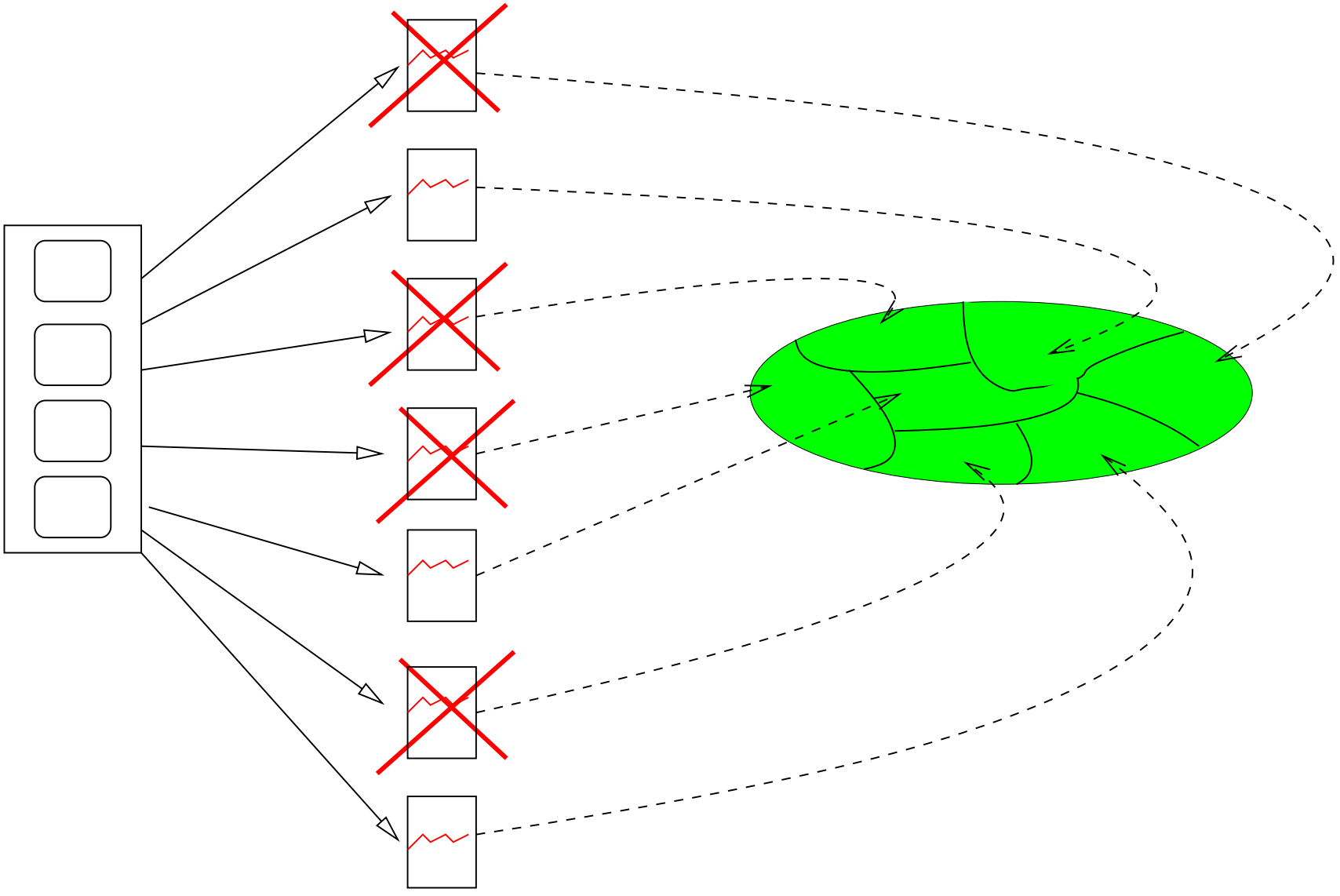
Qual a idéia?



Qual a idéia?

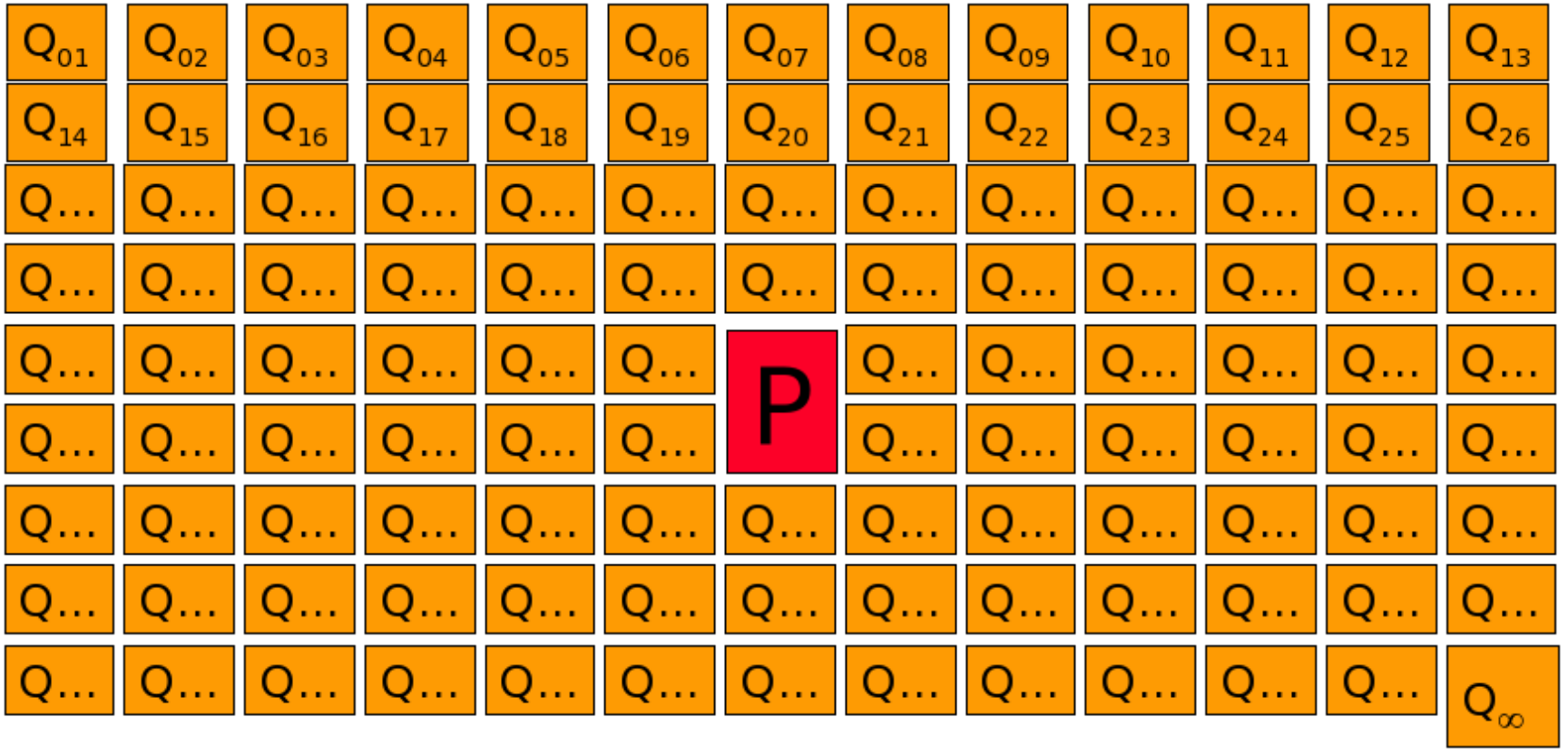


Teste de subdomínio



Correção absoluta

- Tomando todos os programas Q, é possível provar a correção de P

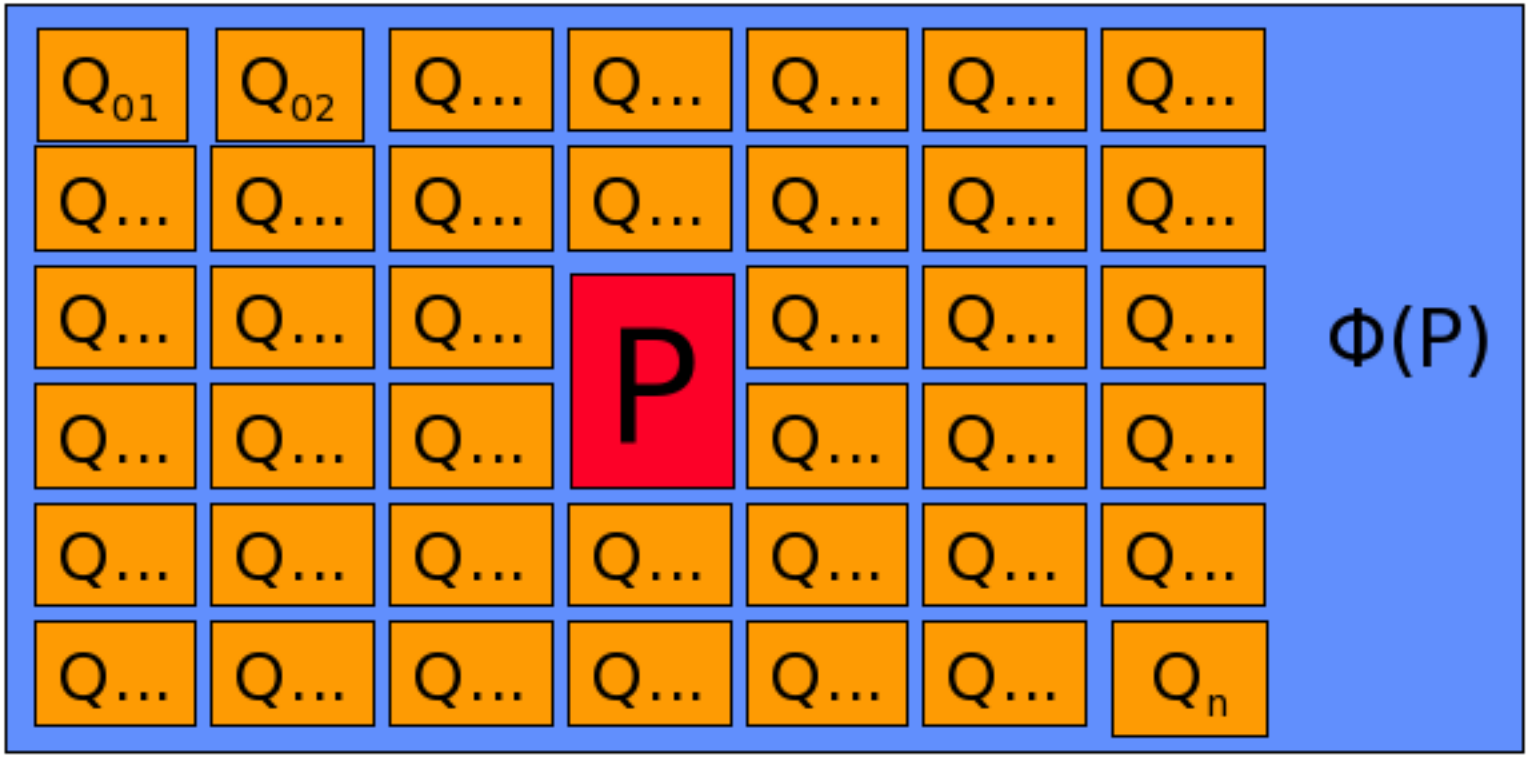


Correção relativa

- O problema com esta vizinhança é que ela é infinita
- Torna-se impossível executar e comparar cada programa Q_i
- Estabelece-se então uma vizinhança $\Phi(P)$ que contém apenas um conjunto finito de programas
- Esses programas contêm pequenos desvios sintáticos que representam defeitos simples

Correção relativa

- Casos de teste que distingam esses mutantes mostram que P está livre de determinados tipos específicos de defeitos



Hipóteses

- Hipótese do programador competente
  - P está correto ou próximo do correto
- Efeito de acoplamento
  - Casos de teste capazes de distinguir mutantes que possuem pequena diferença semântica em relação a P devem também revelar outros tipos de defeitos

Hipóteses

```
read x, y, z
m := x
if m < y
    m := y
if m < z
    m := z
print m
```

4

X = 4
Y = 2
z = 1

```
read x, y, z
m := x
if m ≠ y
    m := y
if m < z
    m := z
print m
```

2

```
read x, y, z
m := y
if m > z
    m := z
print m
```

1

Operadores de mutação

- Os operadores de mutação determinam o tipo de alteração sintática que deve ser feita para a criação dos mutantes
- Esses operadores buscam introduzir pequenas alterações semânticas através de pequenas alterações sintáticas que representam defeitos típicos
- Existem também os mutantes instrumentados que não modelam defeitos típicos
- Operadores dependem da linguagem alvo
  - FORTRAN 22 operadores
  - C 75 operadores

Operadores para C

- Conjunto de 75 operadores divididos em 4 classes
  - Mutação de comandos
  - Mutação de operadores
  - Mutação de variáveis
  - Mutação de constantes
- Implementados na ferramenta Proteum

Mutação de comandos

SMVB - Move Brace Up and Down

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
c[a] = 0;
```

```
while (a < 10)
{
   c[a] = a+b;
}
a++;
c[a] = 0;
```

```
while (a < 10)
{
   c[a] = a+b;
   a++;
   c[a] = 0;
}
```

Mutação de comandos – SSDL

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
 /* --> */          ; // comando removido
                x[i+1] = aux;
            }
        }
    }
}
```

Mutação de operadores

ORRN - Replace relational operator by relational operator

```
while (a < 10)
{
  c[a] = a+b;
  a++;
}
c[a] = 0;
```

```
while (a > 10)
```

```
while (a <= 10)
```

```
while (a >= 10)
```

```
while (a != 10)
```

```
while (a == 10)
{
  c[a] = a+b;
  a++;
}
c[a] = 0;
```

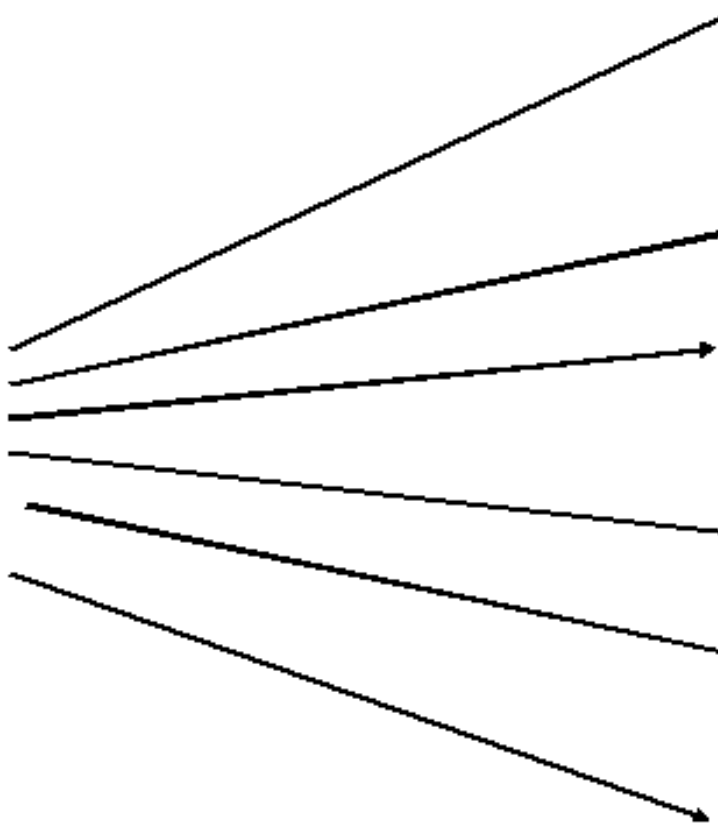
Mutação de operadores – ORRN

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
 /* --> */  if (x[i] != x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Mutação de variáveis

Vssr - scalar variable replacement

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
c[a] = 0;
```



- `while (b < 10)`
- `c[b] = a+b;`
- `c[a] = b+b;`
- `c[a] = a+a;`
- `b++;`
- `c[b] = 0;`

Mutação de variáveis – VSSR

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
 /* --> */         int aux = x[last];
                   x[i] = x[i+1];
                   x[i+1] = aux;
            }
        }
    }
}
```

Mutação de constantes

Cccr - Constant for constant replacement

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
c[a] = 0;
```

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
c[a] = 10;
```

```
while (a < 0)
{
   c[a] = a+b;
   a++;
}
c[a] = 0;
```

Mutação de constantes – CCCR

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
  /* --> */     x[i] = x[i+0];
                x[i+1] = aux;
            }
        }
    }
}
```

Mutantes instrumentados

STRP - trap on statement execution

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
c[a] = 0;
```

```
while (a < 10)
{
   trap();
   a++;
}
c[a] = 0;
```

```
trap();
c[a] = 0;
```

```
while (a < 10)
{
   c[a] = a+b;
   trap();
}
c[a] = 0;
```

```
while (a < 10)
{
   c[a] = a+b;
   a++;
}
trap();
```

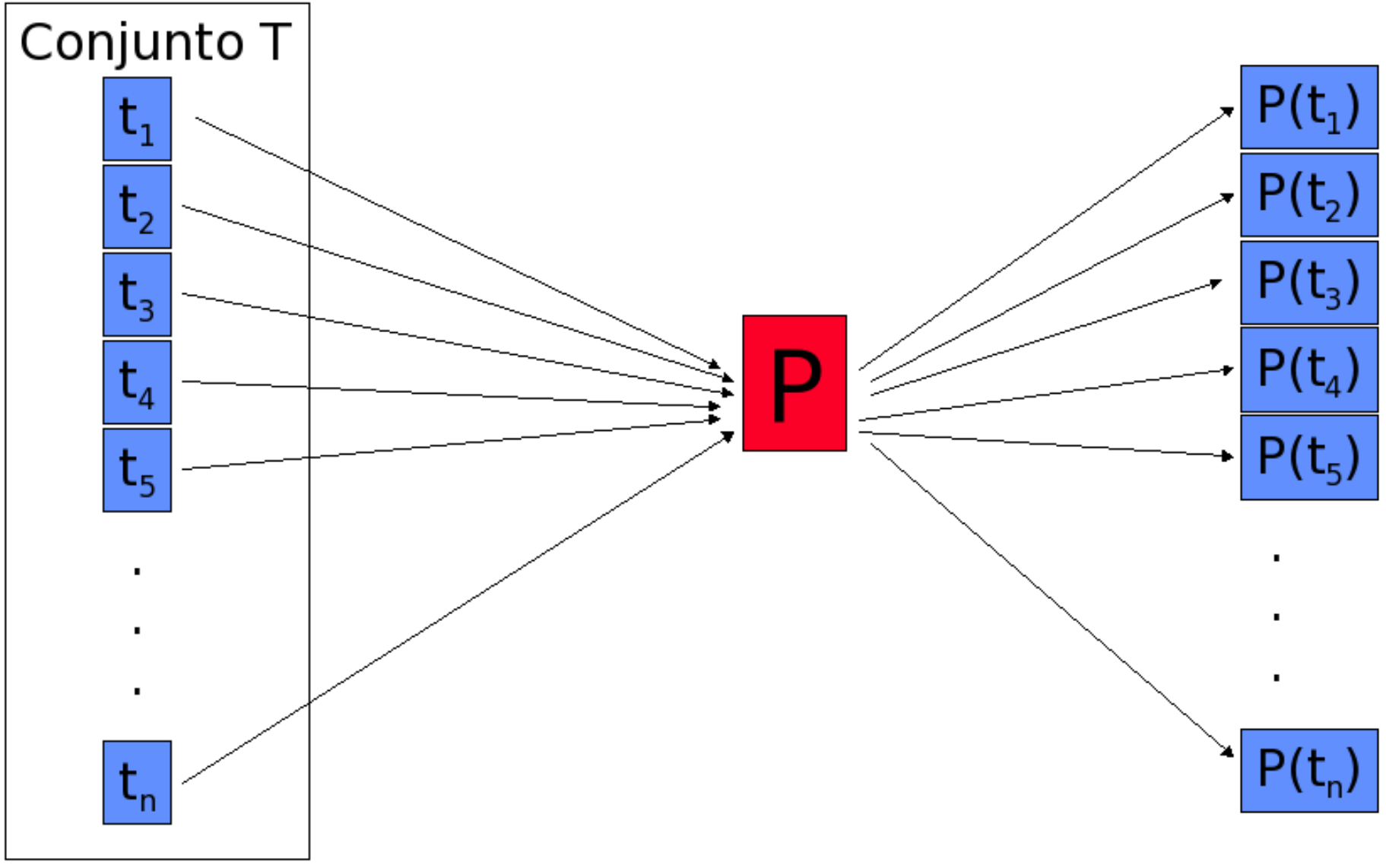
Mutante instrumentado

```
void bubbleSort(int x[], int n) {
    for (int last = 1; TRAP_ON_TRUE(last < n);
         last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

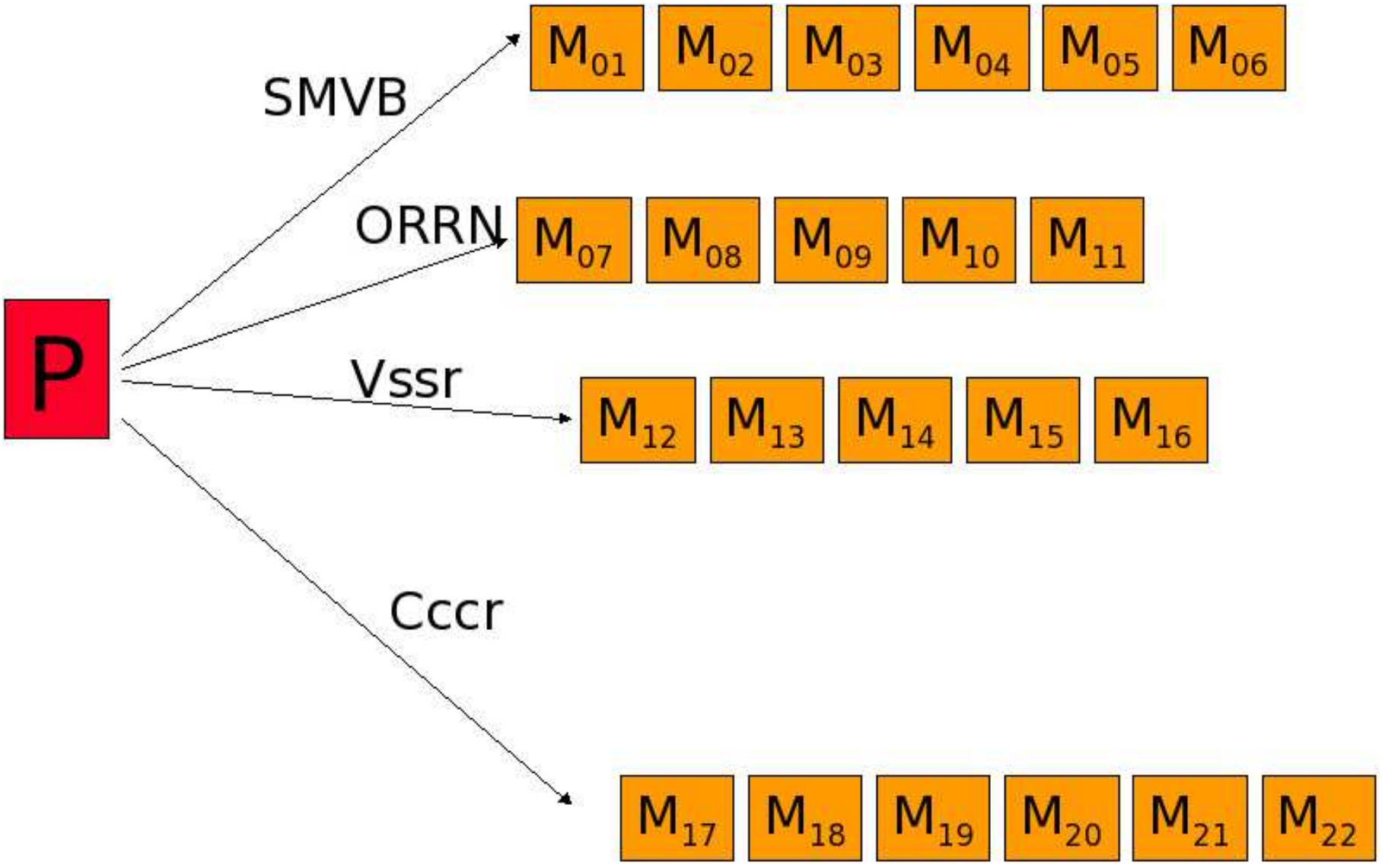
Aplicação do critério

- P é executado com os casos de teste de T
- Mutantes são gerados
- Mutantes são executados com os casos de teste de T
- Mutantes são analisados

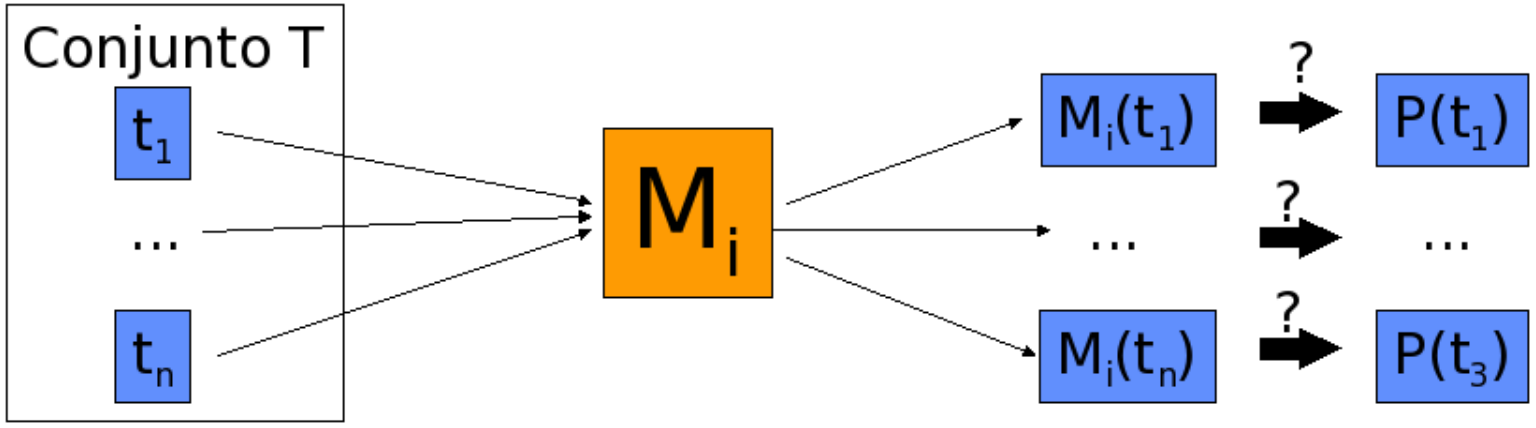
Execução de P



Geração dos mutantes

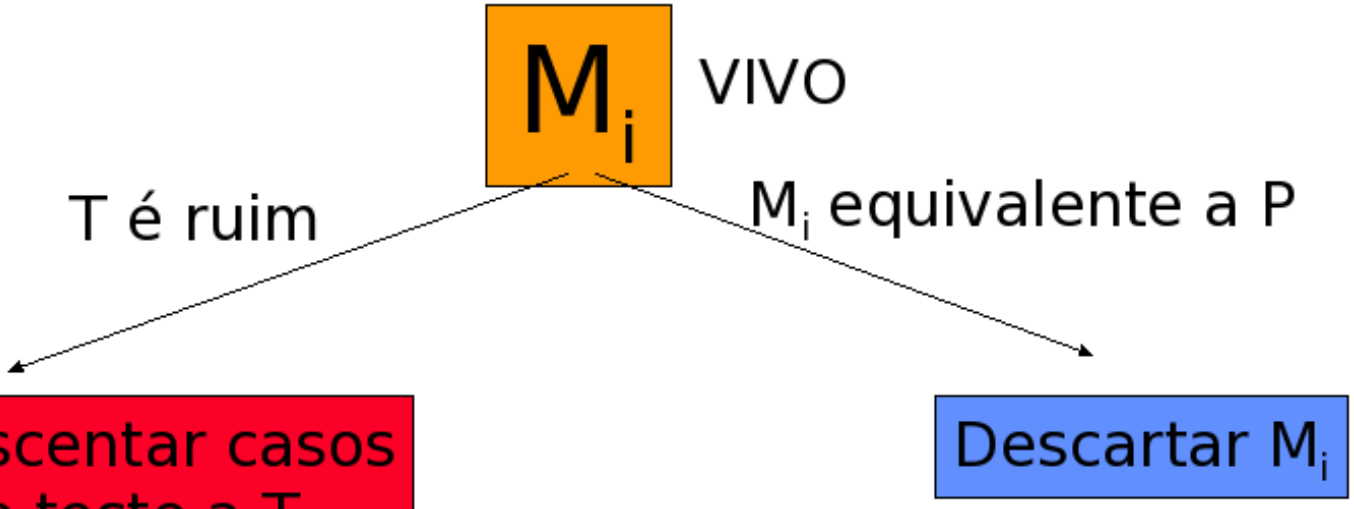


Execução dos mutantes



- $M_i(T) = P(T)$ então M_i está vivo
- senão M_i está morto
- escore de mutação $= \frac{\text{\# mutantes mortos}}{\text{\# total de mutantes}}$

Análise dos mutantes

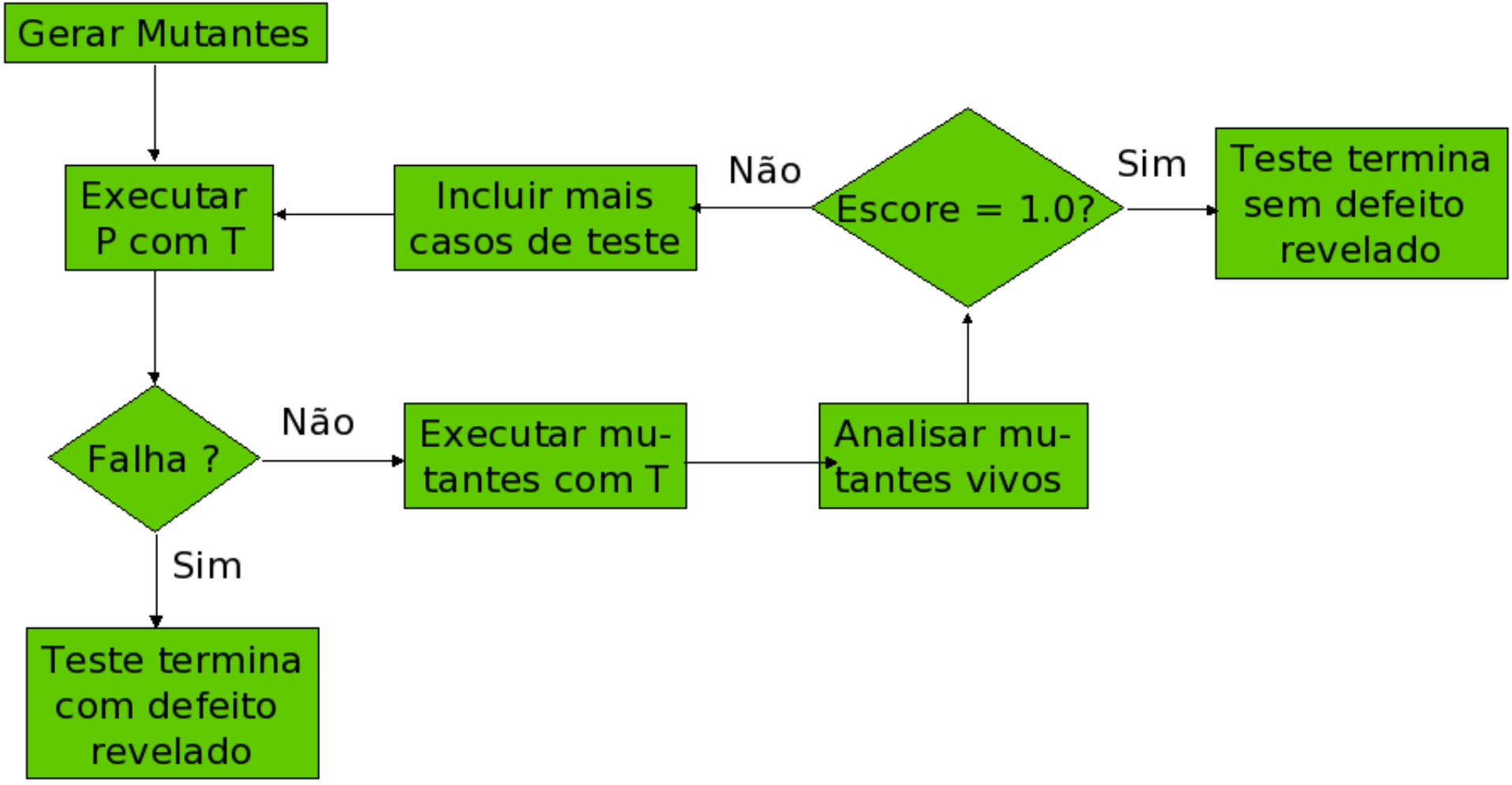


- escore de mutação $= \dfrac{\#\text{ mutantes mortos}}{\#\text{ mutantes não equivalentes}}$

Condições de suficiência

- Para que um mutante **M** seja morto por um caso de teste **t**, três condiçoes devem ser satisfeitas:
  - Alcançabilidade: o ponto de mutação deve ser executado pelo caso de teste
  - Infecção: a execução da mutação deve fazer com que o programa mutante assuma um estado diferente do programa original
  - Propagação: como apenas os resultados dos mutantes são comparados, essa diferença de estado deve fazer com que algum comportamento visível do mutante seja diferente daquele do programa original
- RIP (Reachability, Infection, Propagation)
- Defeito, erro, falha

Aplicação do critério



Considerações

- Mutantes equivalentes
  - Assim como associações de fluxo de dados não executáveis, a equivalência é indecidível
- Custo
  - O número de mutantes a serem executados pode ser muito alto
  - Exemplo: Cal.c gera 4257 mutantes para 106 LOC

Custo de aplicação

- Número de mutantes é alto

Custo de aplicação

- Número de mutantes é alto
- Tempo de execução é baixo

Custo de aplicação

- Número de mutantes é alto
- Tempo de execução é baixo
- Número de equivalentes é alto

Custo de aplicação

- Número de mutantes é alto
- Tempo de execução é baixo
- Número de equivalentes é alto
- Problema indecidível

Custo de aplicação

- Número de mutantes é alto
- Tempo de execução é baixo
- Número de equivalentes é alto
- Problema indecidível
- Alguns casos é possível decidir automaticamente

Custo de aplicação

- Número de mutantes é alto
- Tempo de execução é baixo
- Número de equivalentes é alto
- Problema indecidível
- Alguns casos é possível decidir automaticamente
- Vale a pena?

Equivalência

- Problema indecidível

```
read x, y, z
m := x
if m < y
    m := y
if m < z
    m := z
print m
```

```
read x, y, z
m := x
if m <= y
    m := y
if m < z
    m := z
print m
```

Padrões de equivalência

- Triviais: `a = 0;` por `a *= 0;`
- Quase triviais

Comando original	Mutante equivalente
`for (i = 0; i < n; i++)`	`for (i = 0; i != n; i++)`
`if ( (a > b) || (c < d) )`	`if ( (a > b) + (c < d) )`
`if ( (a > b) && (c < d) )`	`if ( (a > b) * (c < d) )`
`if ( (a < 0) || (a > 10) )`	`if ( (a < 0) - (a > 10) )`

Mutantes quase equivalentes

- No comando (1) `last` deveria ser inicializada com 1
- No comando (2) pode ocorrer um erro pois está sendo acessada posição que não existe

```
void bubbleSort(int x[], int n) {
(1) for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
(2)         if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Custo de execução

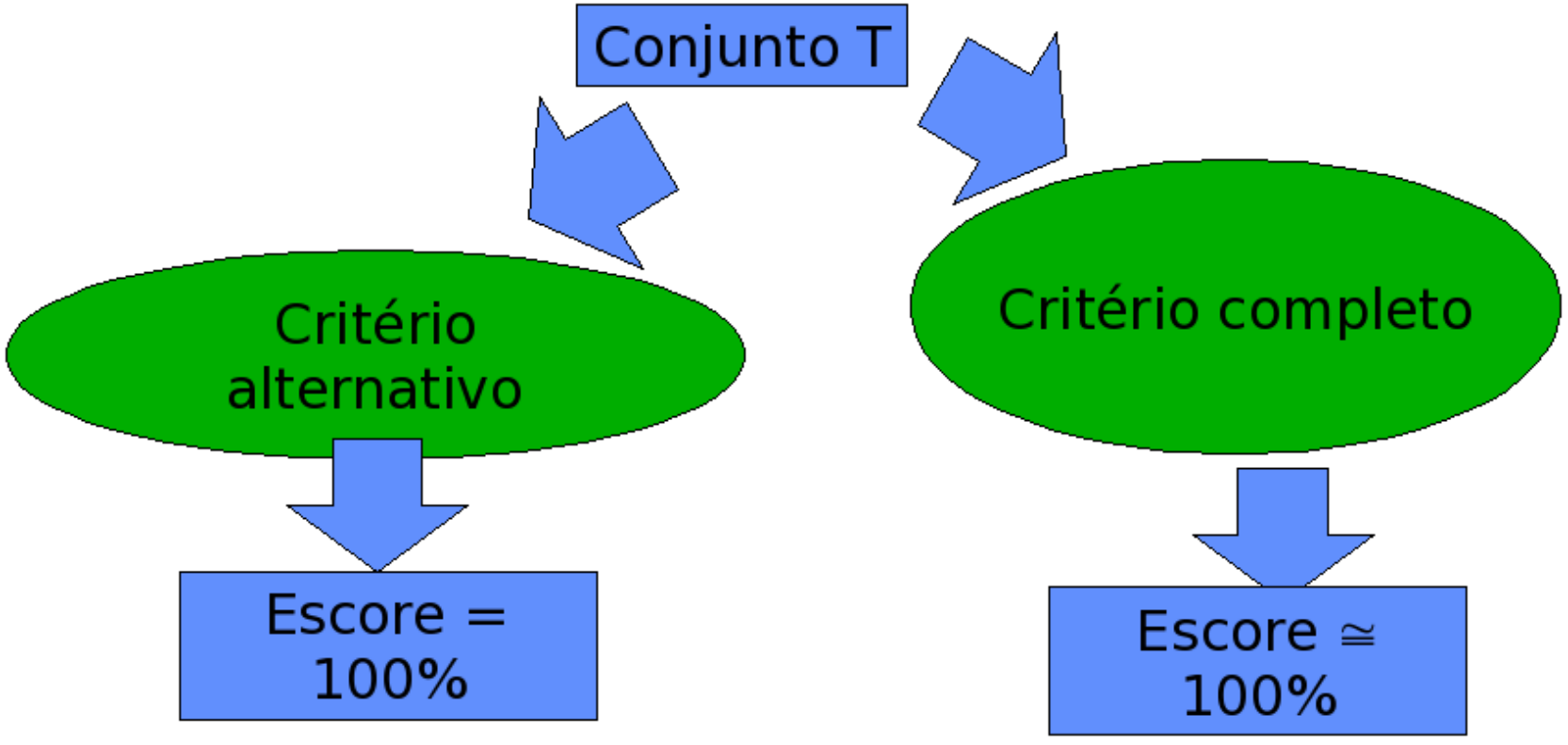
- Uma das soluções para o problema de custo envolve a utilização de hardware de alta performance ou paralelo.
- Nesse caso, o custo em termos de tempo diminui mais o custo total do teste é alto
- Soluções são mais teóricas do que práticas

Redução dos mutantes

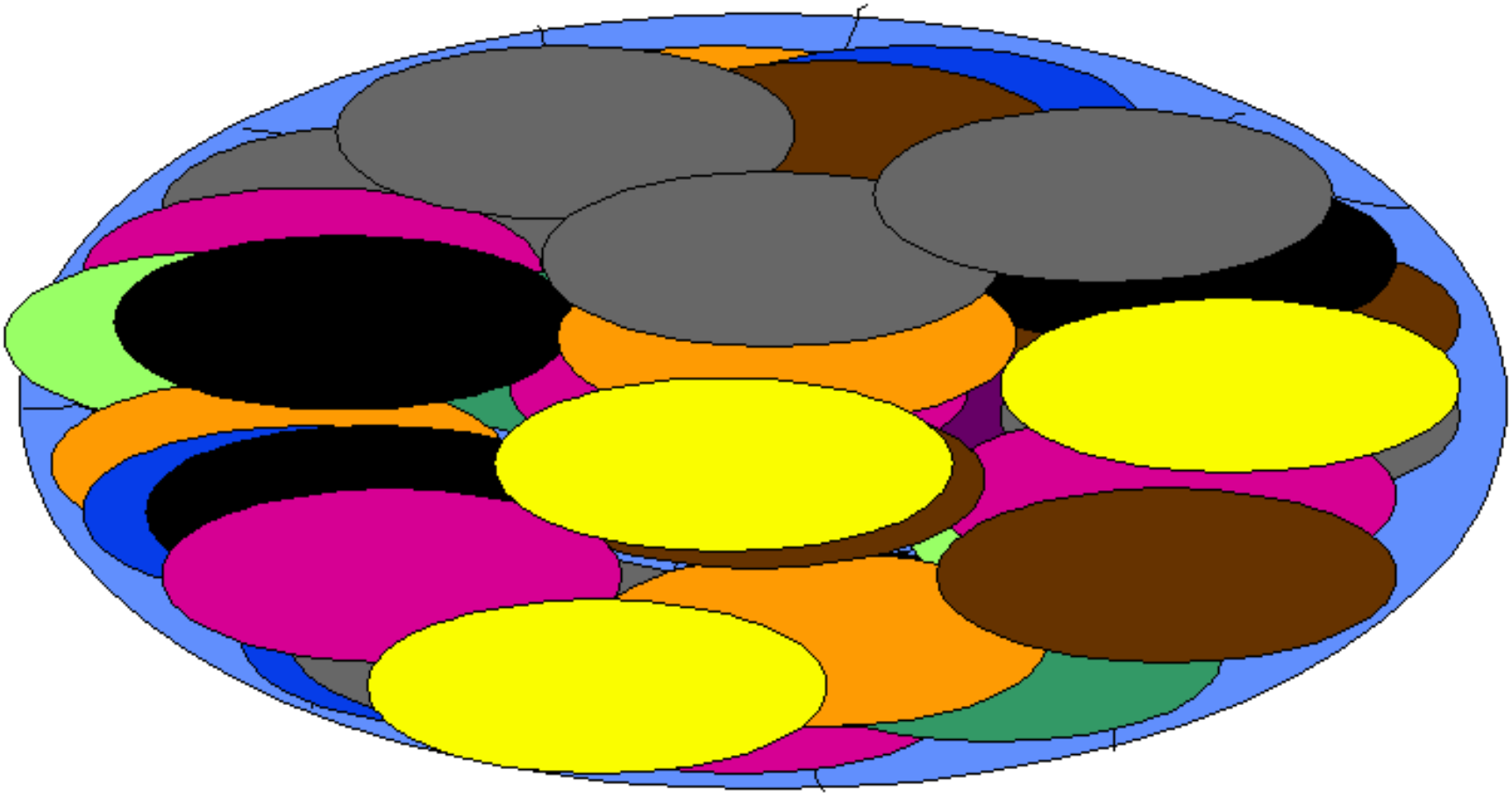
- Outro tipo de solução tenta reduzir o número de mutantes através de algumas abordagens
- Critérios de mutação alternativos
  - Mutação restrita
  - Mutação randômica
- Estratégias podem ser combinadas

Mutação alternativa

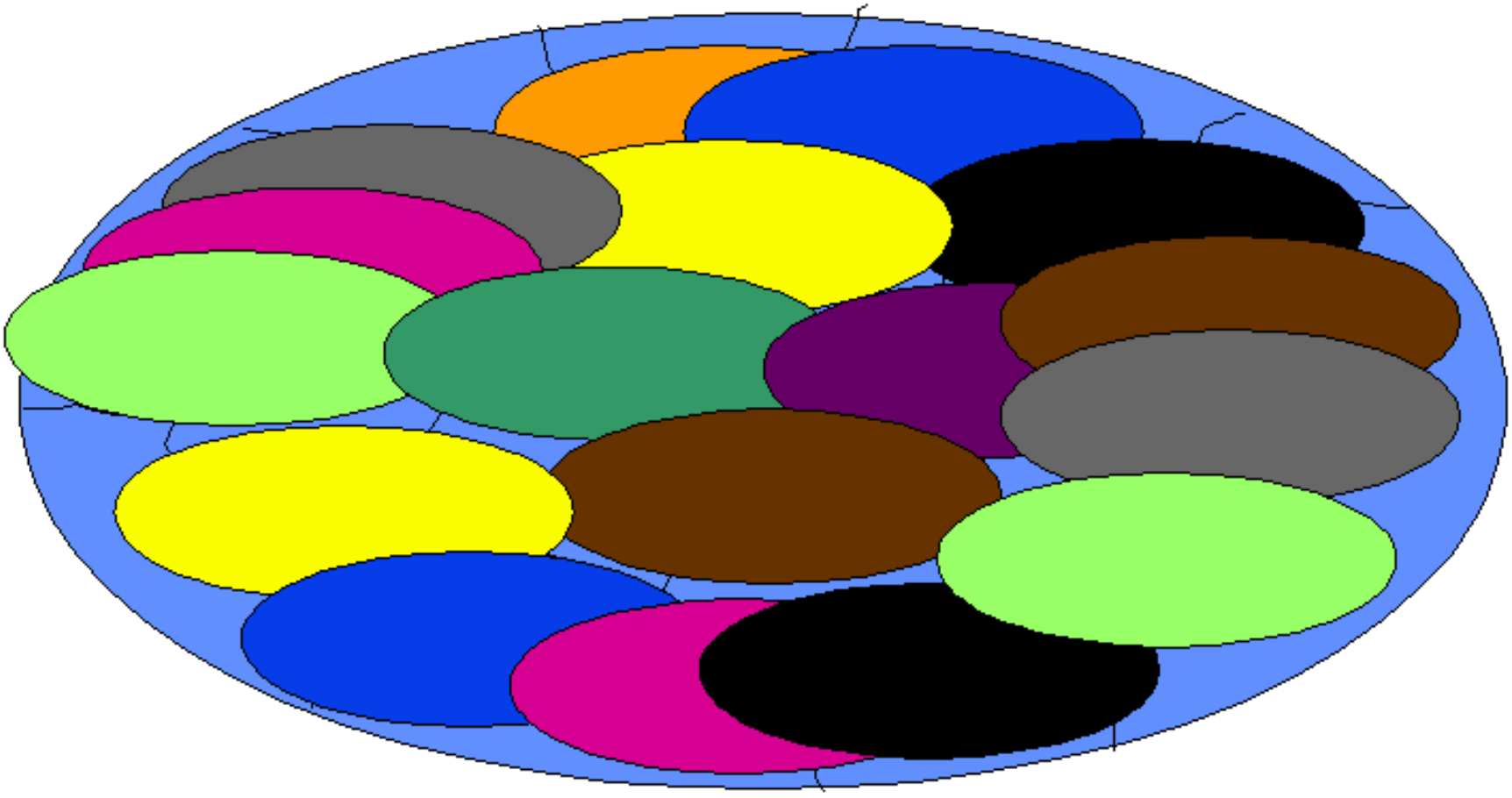
- Utilizando critérios alternativos obtém-se conjuntos quase tão adequados em relação ao critério completo



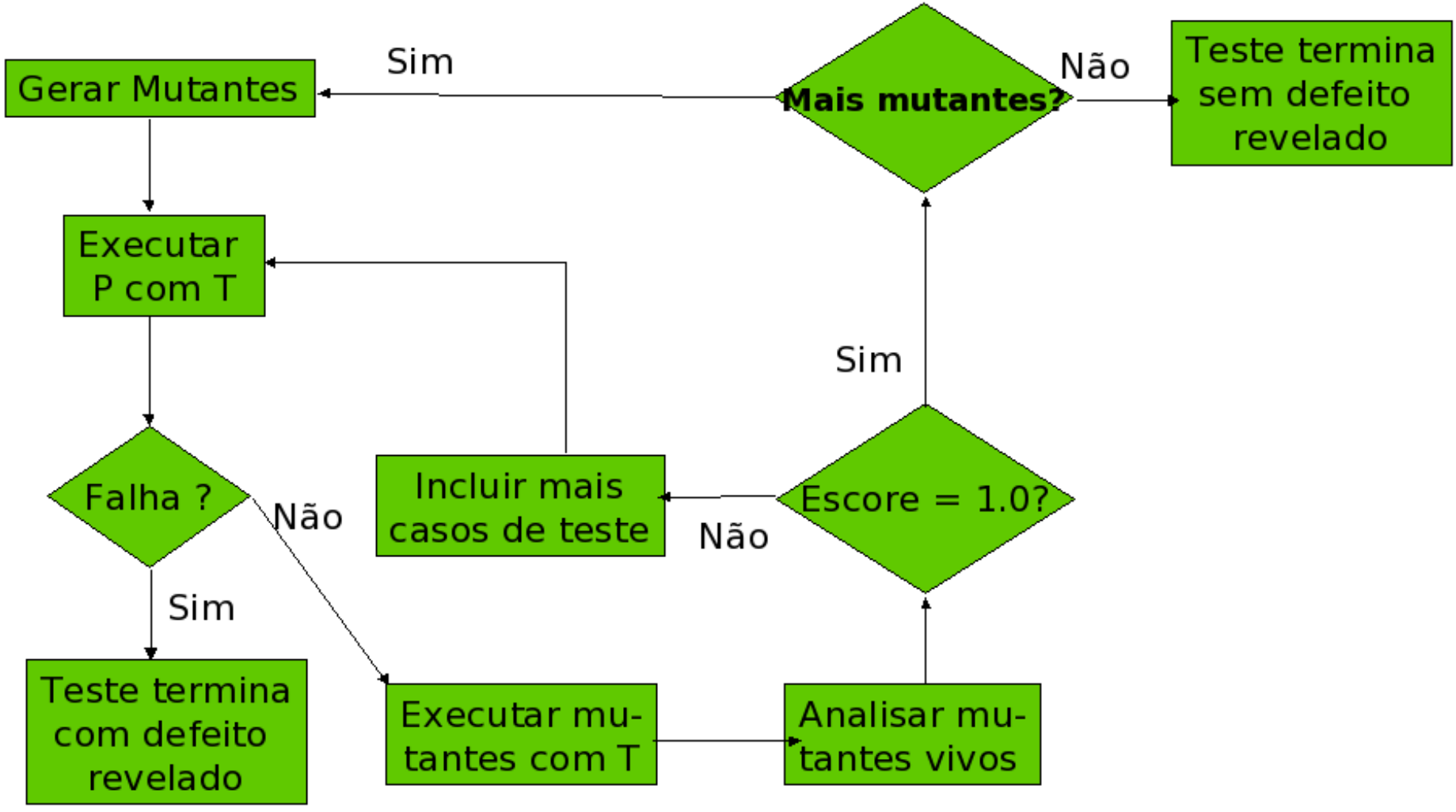
Mutação alternativa



Mutação alternativa



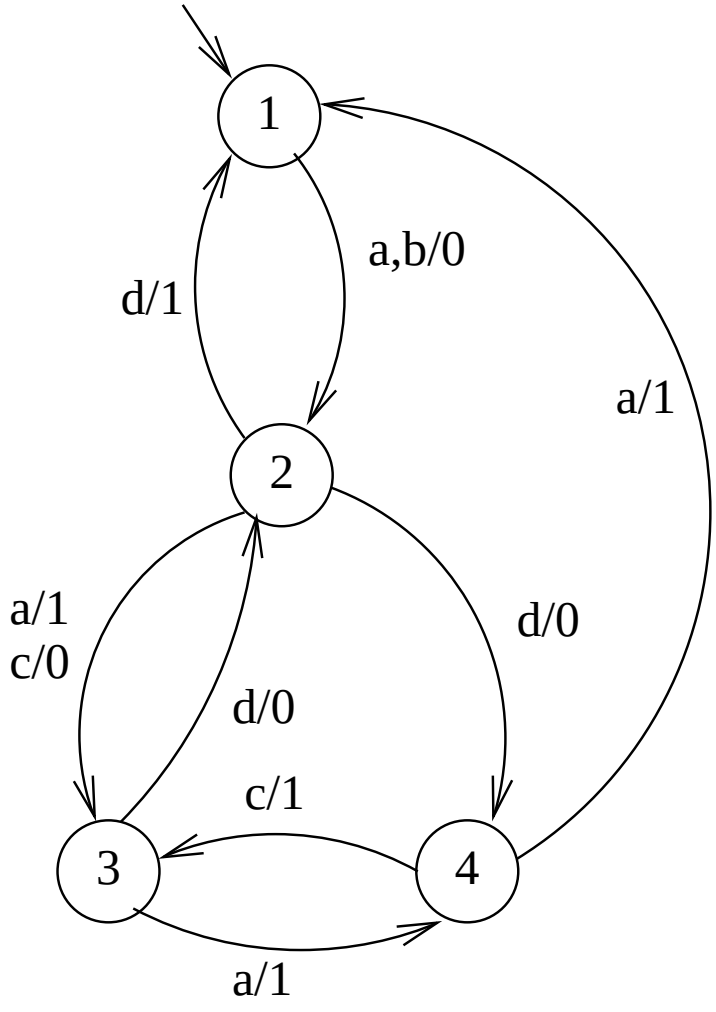
Estratégia incremental



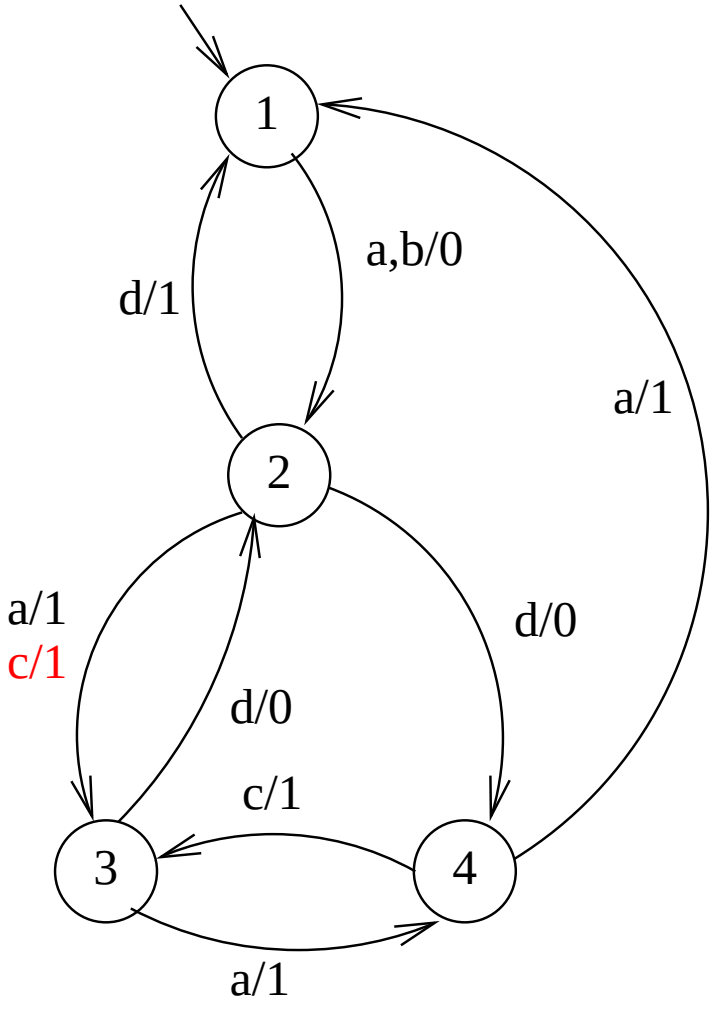
Teste de modelos

- Uma das vantagens do teste de mutação é que ele não requer a identificação de determinadas estruturas como as de fluxo de controle ou de fluxo de dados
- Por isso ele pode ser aplicado em outros tipos de “entidades”
  - Máquinas de estado finito
  - Redes de Petri
  - Statecharts

Máquina de estados finitos



adbca
01001



adbca
01011

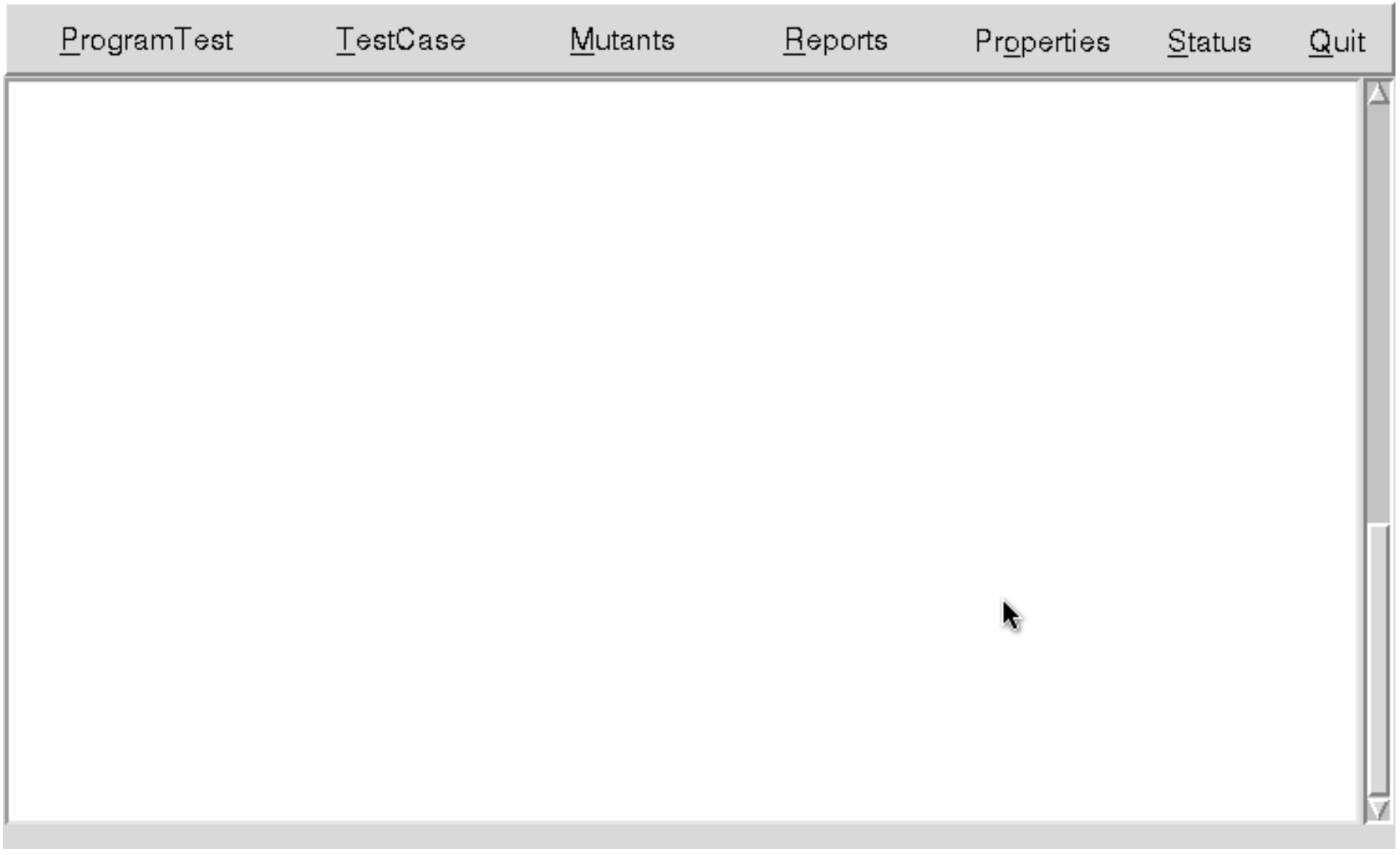
Mutação de interface

- Teste de mutação tem sido utilizado no nível de unidade
  - Operadores exercitam aspectos algoritmicos da unidade
- Mutação de interface visa as interações entre unidades
  - chamadas de funções
  - passagem de parâmetros
  - valores de retorno

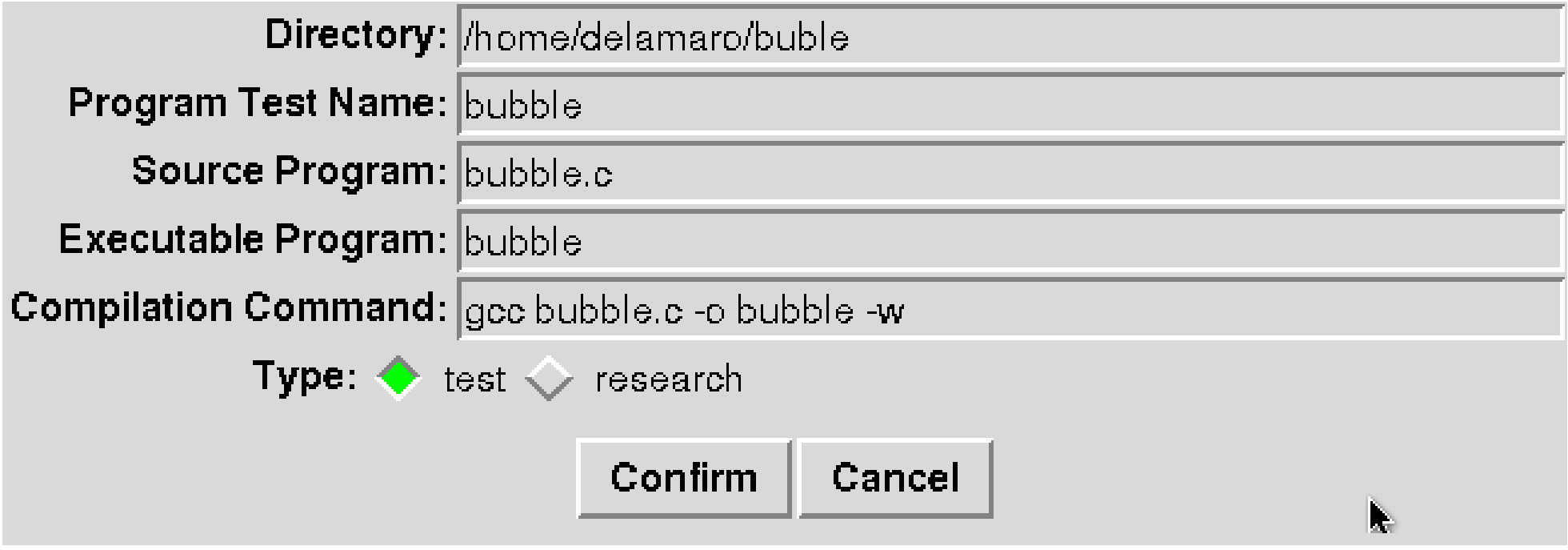
Ferramenta – Proteum

- Linguagem C
- Muito usada no meio acadêmico
- Características que facilitam experimentação
- Facilidade de execução rápida
- Execução via GUI ou via scripts

Menu principal



Criação sessão de teste

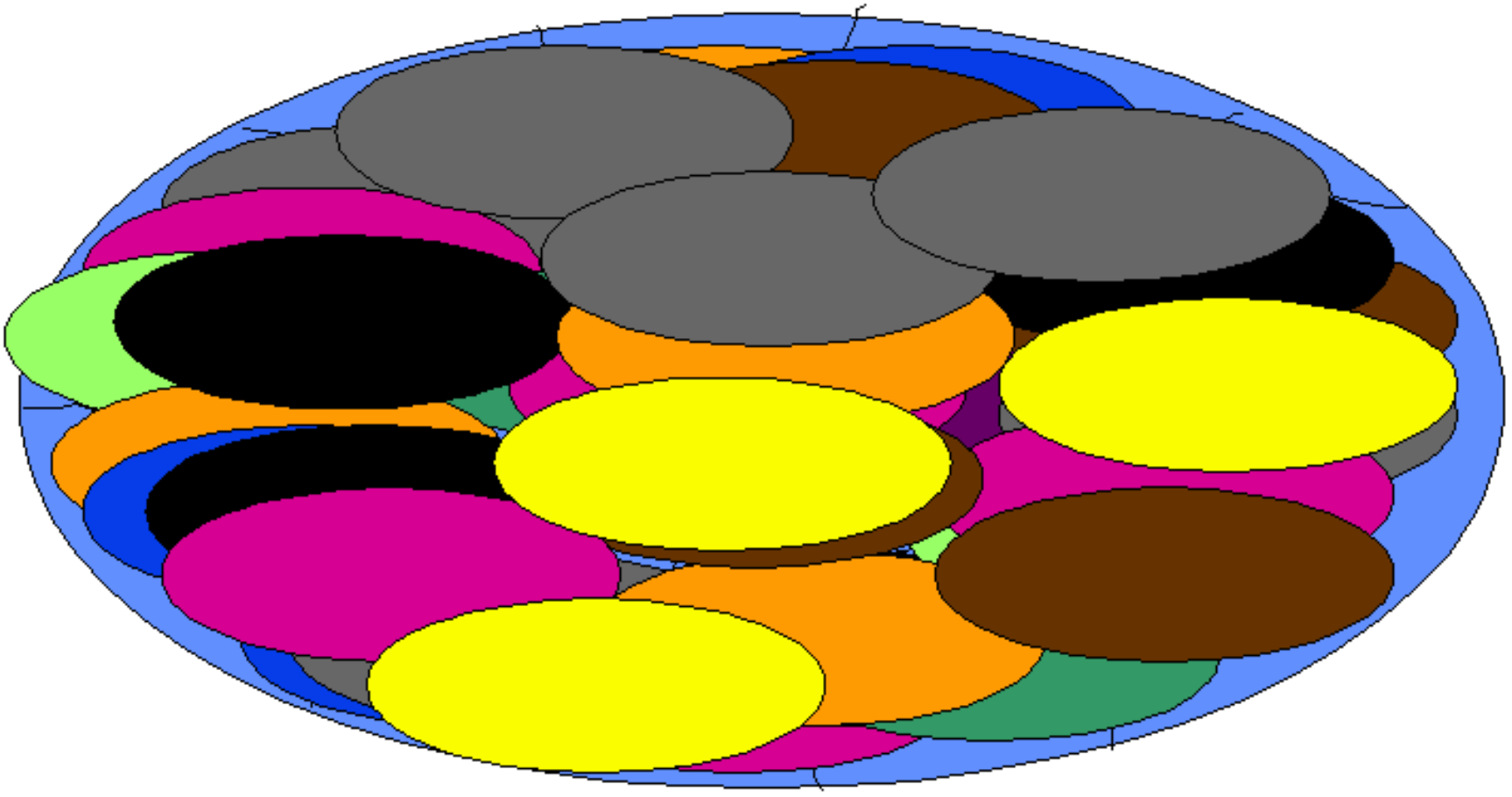


Escolha dos operadores

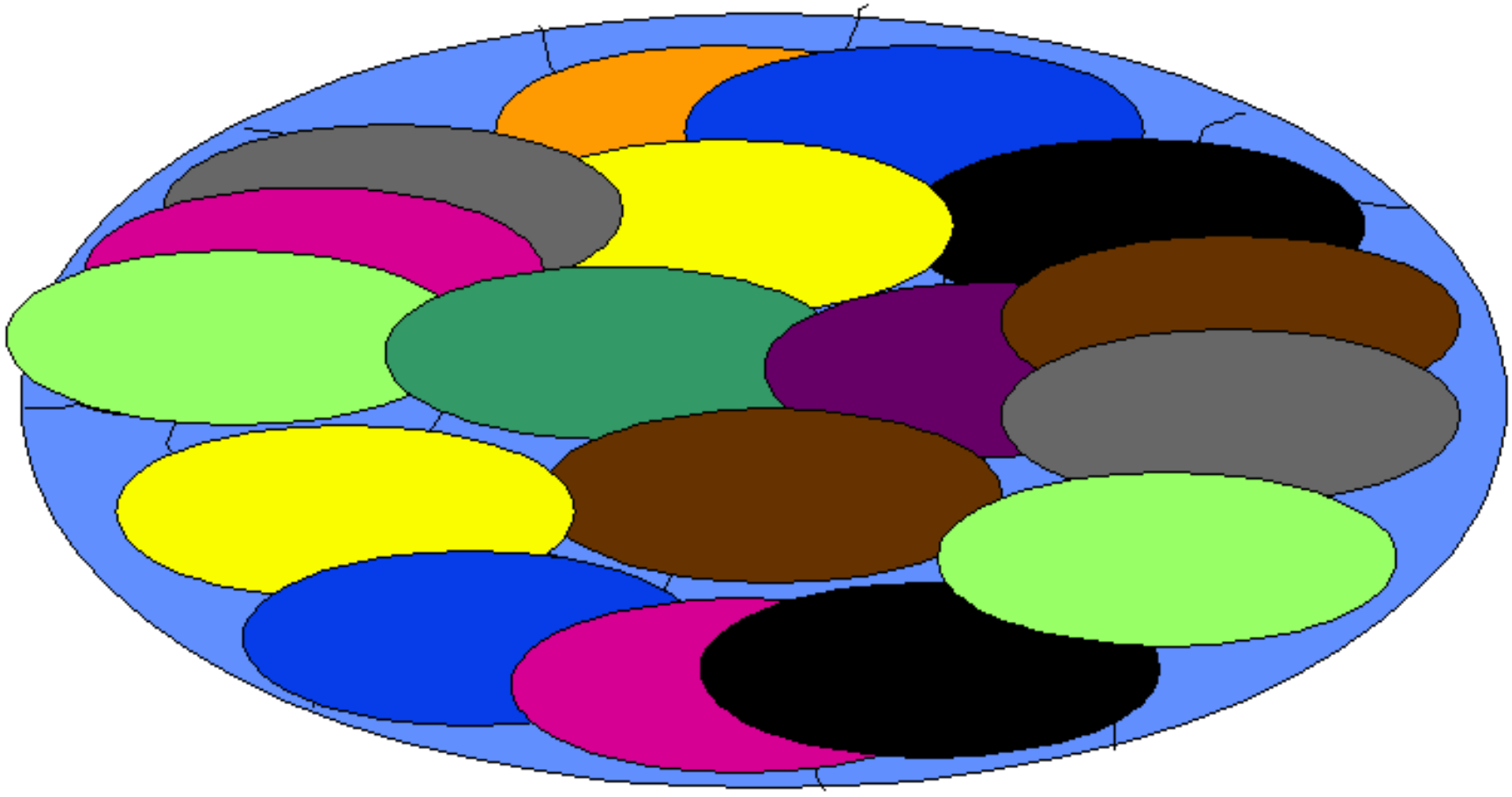
Apply Default | Percentage: 100 | Limit: 0

OLAN - Logical Operator by Arithmetic Operator	0	0	0
OLBN - Logical Operator by Bitwise Operator	0	0	0
OLLN - Logical Operator Mutation	0	0	0
OLNG - Logical Negation	0	0	0
OLRN - Logical Operator by Relational Operator	0	0	0
OLSN - Logical Operator by Shift Operator	0	0	0
ORAN - Relational Operator by Arithmetic Operator	0	0	0
ORBN - Relational Operator by Bitwise Operator	0	0	0
ORLN - Relational Operator by Logical Operator	0	0	0
ORRN - Relational Operator Mutation	100	0	0
ORSN - Relational Operator by Shift Operator	0	0	0
OSAA - Shift Assignment by Arithmetic Assignment	0	0	0

Mutação alternativa



Mutação alternativa



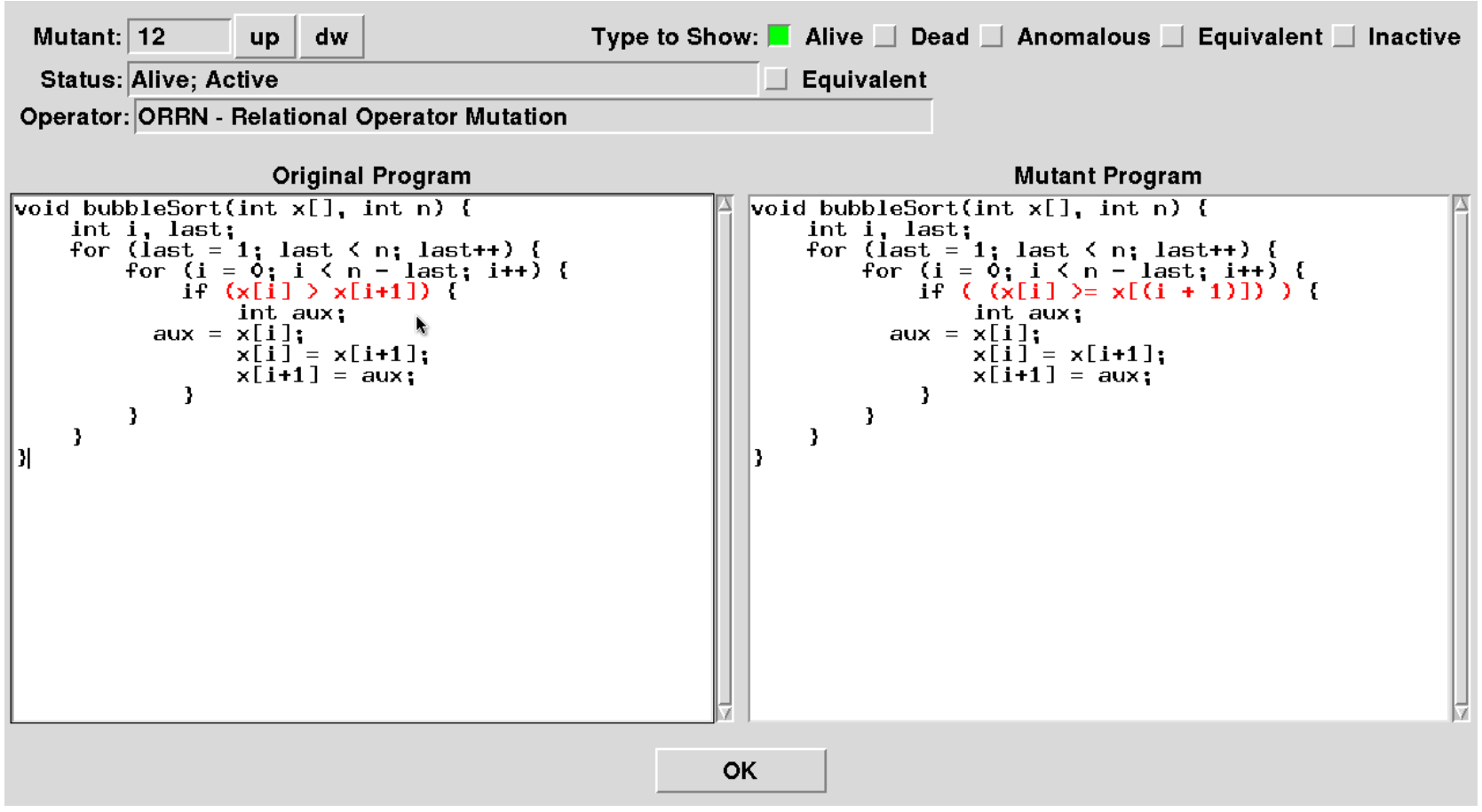
Execução inicial

Directory:	/home/delamaro/buble
Program Test Name:	bubble
Source Program:	bubble
Executable Program:	bubble
Compilation Command:	gcc bubble.c -o bubble -w

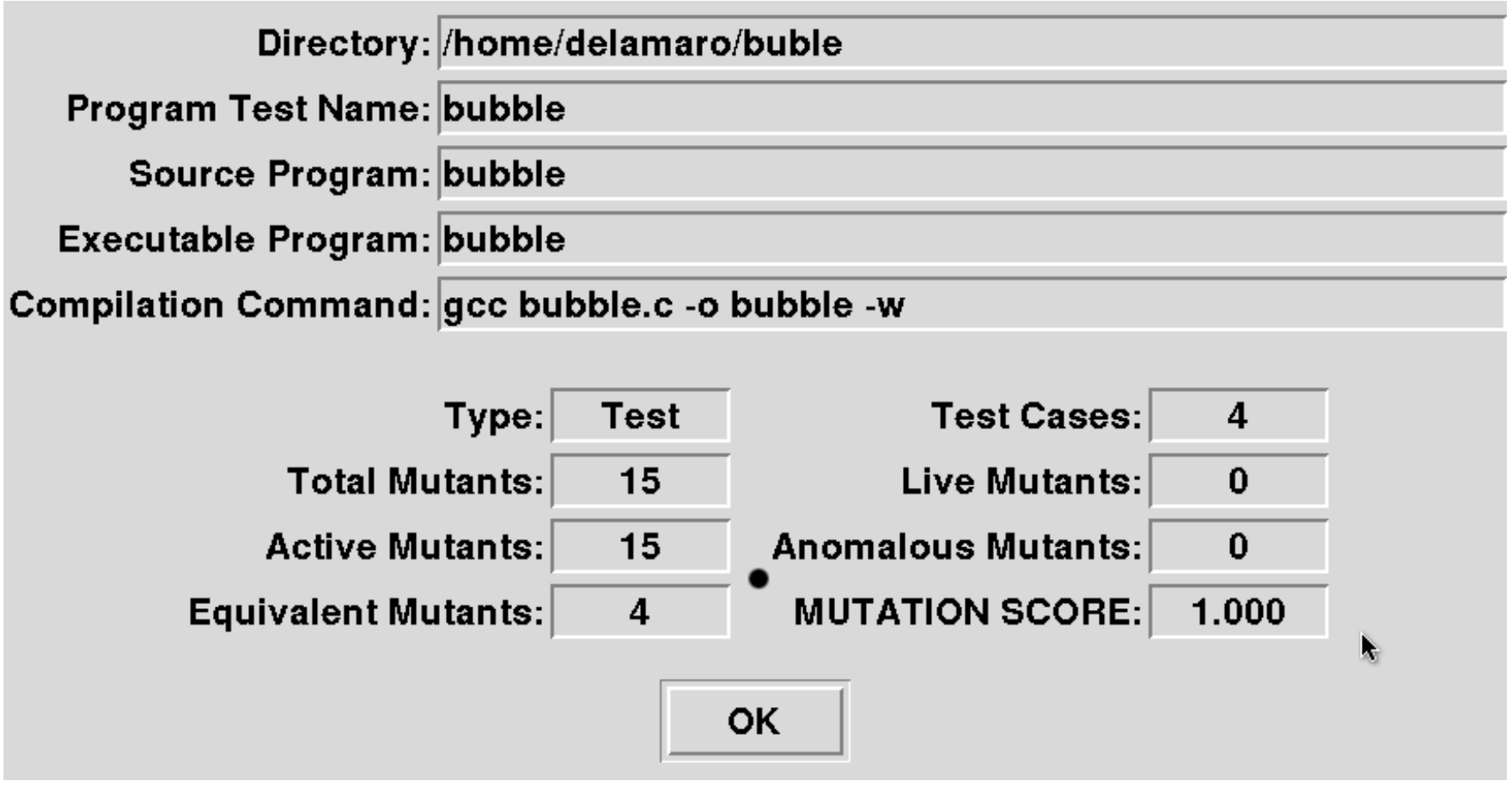
Type:	test	**Test Cases:**	4
Total Mutants:	15	**Live Mutants:**	4
Active Mutants:	15	**Anomalous Mutants:**	0
Equivalent Mutants:	0	**MUTATION SCORE:**	0.733

OK

Análise dos mutantes



Equivalentes identificados



Estratégia incremental

Directory:	/home/delamaro/buble
Program Test Name:	bubble
Source Program:	bubble
Executable Program:	bubble
Compilation Command:	gcc bubble.c -o bubble -w

Type:	Test	Test Cases:	4
Total Mutants:	469	Live Mutants:	51
Active Mutants:	469	Anomalous Mutants:	0
Equivalent Mutants:	4	MUTATION SCORE:	0.890

OK