



静态测试

Static Testing

提纲

- ◆ 静态测试概述
- ◆ 静态测试的方法
- ◆ 静态白盒测试工具
 - C++ test
 - PC-Lint
 - soot

提纲

- ◆ 静态测试概述
- ◆ 静态测试的方法
- ◆ 静态白盒测试工具
 - C++ test
 - PC-Lint
 - soot

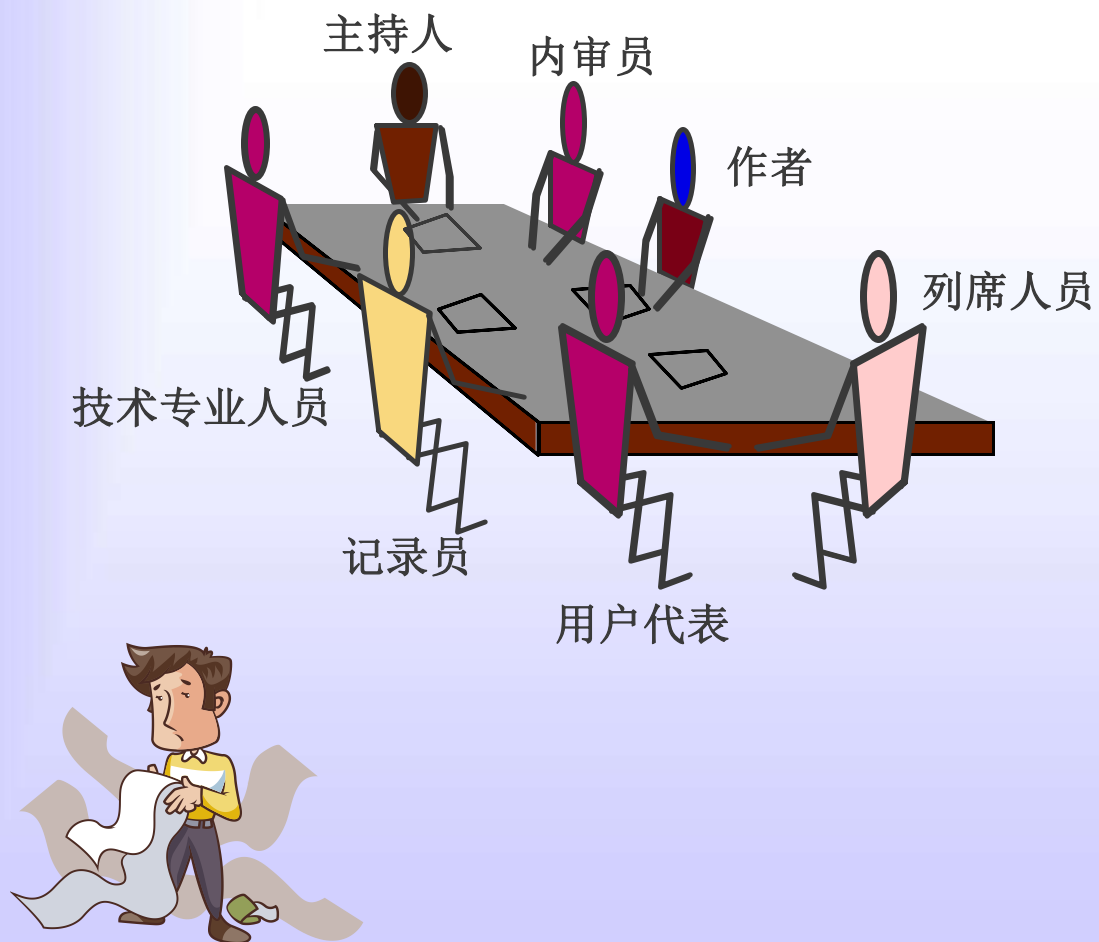
1. 静态测试概述

- ◆ 静态测试和动态测试的概念
- ◆ 为什么需要静态测试
- ◆ 静态测试的重要性

1.1静态测试的概念

- ◆ 根据是否需要执行被测软件，分为静态测试和动态测试
- ◆ **静态测试（static testing）** 不执行被测软件，对需求规格说明书和设计说明书进行评审，对源程序代码进行审查和静态分析，等。
 - 对源程序代码进行静态测试，可以找出程序中的欠缺和可疑之处，例如不匹配的参数、不允许的递归、未使用过的变量、空指针的引用等。
- ◆ **动态测试（dynamic testing）** 通过运行软件的组件或系统来测试软件。通过观察代码运行过程，来获取系统行为、变量实时结果、内存、堆栈、线程以及测试覆盖等各方面的信息，来判断系统是否存在问题；或者通过运行测试用例，来发现缺陷。

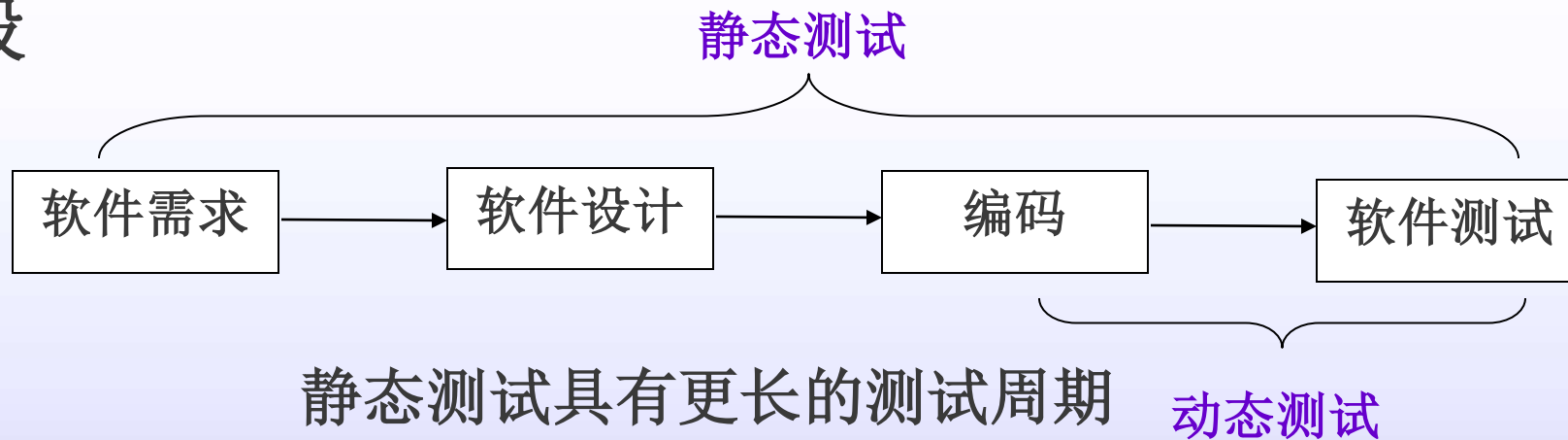
静态的和动态的



运行程序

1.2 为什么需要静态测试

- ◆ 狭隘的软件测试只对可运行的软件进行测试，广义的软件测试是将测试遍布于软件生命周期的各个阶段，包括需求、设计、编码、测试及维护等阶段



- ◆ 静态测试不仅具有更长的生命周期，而且由于其大多数情况下是对软件系统**高层次的测试评审**，能够在软件开发的**早期**找出软件缺陷，更能体现测试的经济学原则。

1.3 静态测试的重要性

- ◆ 发现设计的方向性问题
- ◆ 更早的发现问题
- ◆ 避免杀虫剂现象
- ◆ 引起程序设计人员的重视
- ◆ 静态测试可以训练程序员

提纲

- ◆ 静态测试概述
- ◆ 静态测试的方法
- ◆ 静态白盒测试工具
 - C++ test
 - PC-Lint
 - soot

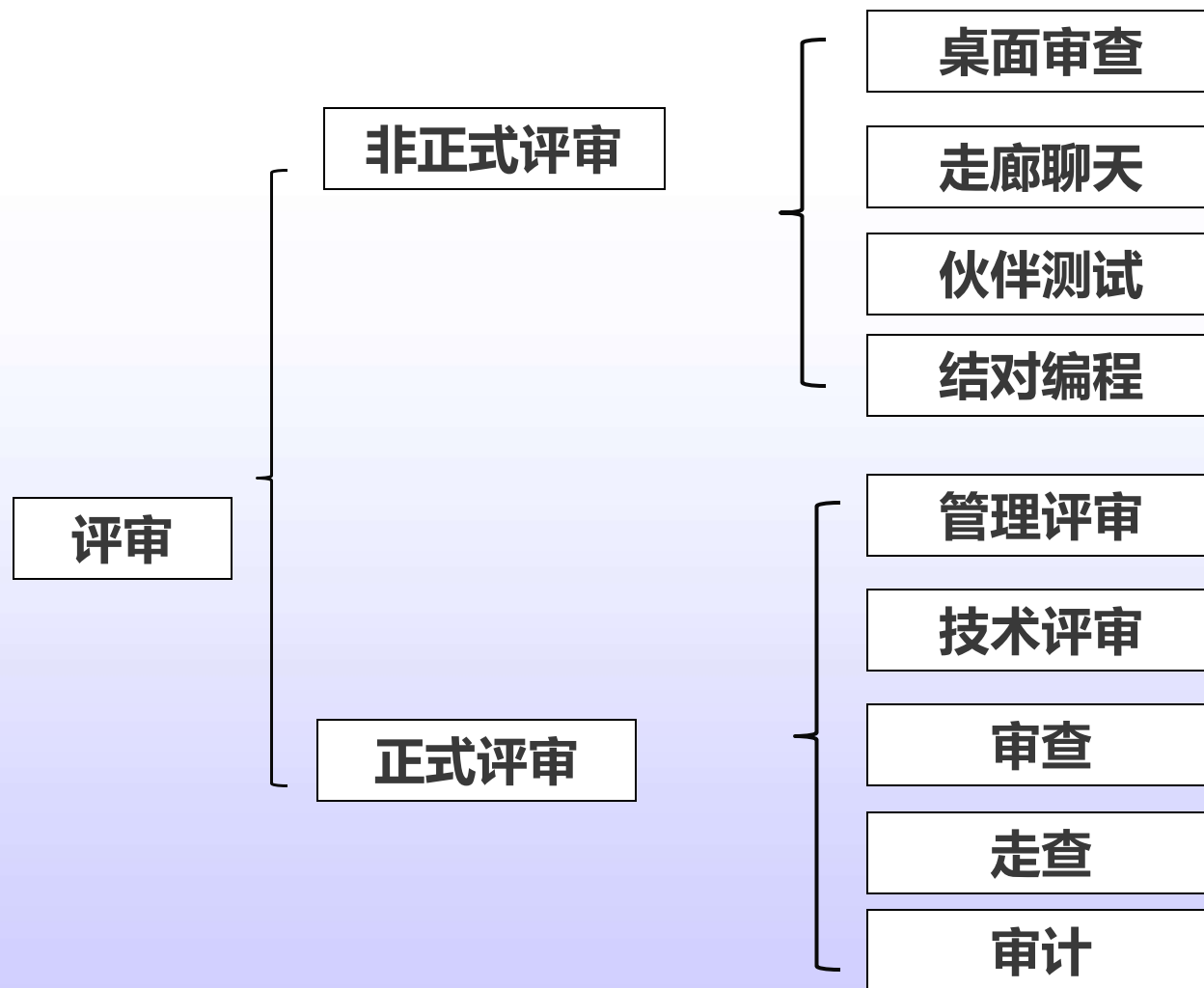
2. 静态测试的方法

- ◆ 对于需求规格说明、设计说明等相关文档，一般采用**评审**的方法进行静态测试
- ◆ 对于源代码，目前一般采用**自动化工具**进行静态白盒测试
 - **C++Test**: 针对C/C++
 - **PC-Lint** : 针对C/C++
 - **FindBug** : 针对Java, 开源
 -

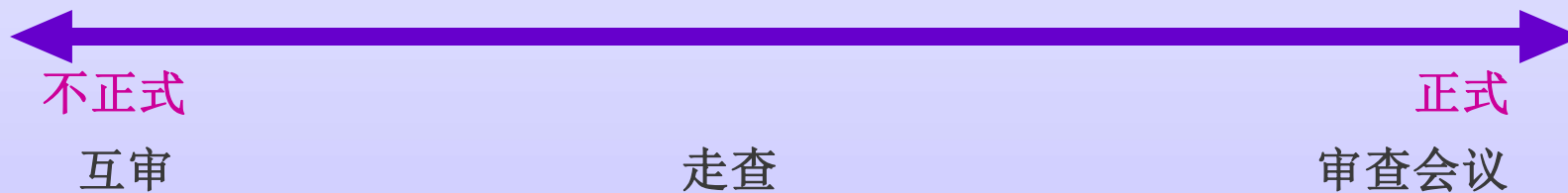
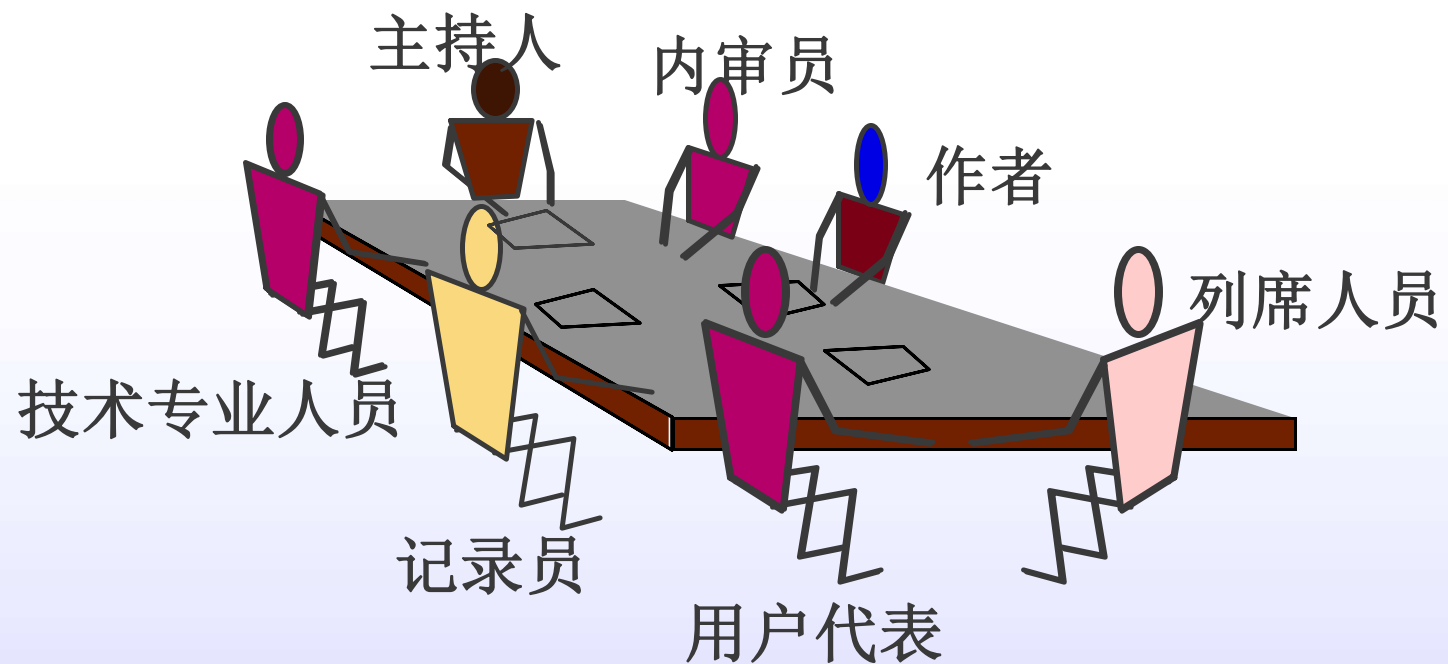
2.1 评审的概念

- ◆ **评审（review）**：对软件元素或项目状态的一种评估手段，以确定与计划的结果所存在的误差，并提供改进建议。（IEEE Std1028-1988）
- ◆ **评审是主要的静态测试技术**
- ◆ **评审是一个过程或会议**，将软件产品或软件过程呈现给工程人员、管理者、使用者、使用者代表、审计人员或其他感兴趣的人员进行检查、评价或建议

2.2 评审的形式



评审



2.3 评审的对象

- ◆ **需求评审**：审查需求说明书是否完整、正确、清晰，是否是用户的真正需求。包括对**文档的评审**：文档格式、术语、内容等的检查。
- ◆ **设计评审**：对所设计的系统结构的合理性、可测试性、安全性、可用性、.....等进行评审；利用关系数据库规范化理论对数据库模式进行审查；等等
- ◆ **代码评审**：人工阅读代码，审查程序的结构、代码风格、算法等

提纲

- ◆ 静态测试概述
- ◆ 静态测试的方法
- ◆ 静态白盒测试工具
 - C++ test
 - PC-Lint
 - soot

3. 静态白盒测试工具

- ◆ 对源代码进行分析，检查程序逻辑的各种缺陷和可疑的程序构造
 - 如：错误的局部变量和全局变量、不匹配的参数、潜在的死循环、空指针、等
- ◆ 还可以用符号代替数值求得程序结果，以便发现缺陷
 - 比如符号化执行工具JPF(Java Path Finder)

3.1 C++ test

- ◆ 商业工具，是Parasoft针对C/C++的一款自动化测试工具。

<https://www.parasoft.com/>

- ◆ 支持静态分析、全面代码走查、单元与组件的测试、.....
- ◆ 可以申请试用版

The screenshot displays the Parasoft website's product offerings. The header includes the Parasoft logo, a 'TRY PARASOFT' button, and links to 'Customer Portal', 'Forums', 'Marketplace', 'Products', 'Solutions', and 'Partners'. The main content is organized into two columns: 'DEVELOPMENT TESTING' and 'FUNCTIONAL TESTING'. Under 'DEVELOPMENT TESTING', there are three sub-sections: 'Parasoft C/C++test' (listing Static Analysis, Unit Testing, Coverage & Traceability, Runtime Analysis, Security Testing, and Functional Safety & Compliance), 'Parasoft Jtest' (listing Static Analysis, Unit Testing, Test Impact Analysis, Coverage & Traceability, and Security Testing), and 'Parasoft Insure++' (listing Memory Debugging). Under 'FUNCTIONAL TESTING', there are two sub-sections: 'Parasoft SOAtest' (listing API Testing, Microservices Testing, Web UI Testing, Mobile Testing, Load Testing, and Security Testing) and 'Parasoft dotTEST' (listing .NET Static Analysis, .NET Coverage & Traceability, Test Data Management, Test Orchestration & Reuse, and Reporting & Analytics).

DEVELOPMENT TESTING	FUNCTIONAL TESTING	
Parasoft C/C++test <ul style="list-style-type: none">C/C++ Static AnalysisC/C++ Unit TestingC/C++ Coverage & TraceabilityC/C++ Runtime AnalysisC/C++ Security TestingFunctional Safety & ComplianceReporting & Analytics	Parasoft Jtest <ul style="list-style-type: none">Java Static AnalysisJava Unit TestingJava Test Impact AnalysisJava Coverage & TraceabilityJava Security TestingReporting & Analytics	Parasoft SOAtest <ul style="list-style-type: none">API TestingMicroservices TestingWeb UI TestingMobile TestingLoad TestingSecurity TestingTest Data ManagementTest Orchestration & ReuseReporting & Analytics
Parasoft Insure++ <ul style="list-style-type: none">C/C++ Memory Debugging	Parasoft dotTEST <ul style="list-style-type: none">.NET Static Analysis.NET Coverage & Traceability	

C++ test 静态检测

- ◆ 静态测试方面：在不需要执行程序的情况下识别运行时缺陷，包括未初始化的内存、空指针引用、除零、内存泄漏等缺陷。

可以进行C/C++ 编程规范自动检查，内置了很多业界规则，也可以定制自己的规则。

- BugDetective

潜在的缺陷：避免访问数组越界.....

资源：确保资源已释放.....

- 业界编码规范

不要在for内使用break.....

Parasoft C/C++test: Static Analysis

Parasoft C/C++test

3.2 PC-Lint

- ◆ PC-Lint 是GIMPEL SOFTWARE公司开发的C/C++代码静态分析工具；
- ◆ 在全球拥有广泛的客户群，许多大型的软件开发组织都把PC-Lint 检查作为代码走查的第一道工序。
- ◆ PC-Lint不仅能够对程序进行全局分析，检验数组下标、报告未初始化变量、警告使用空指针以及冗余的代码等，还能够提出在空间利用、运行效率上的改进点。
- ◆ 网上试用版本 <http://www.gimpel-online.com/OnlineTesting.html>

几个静态代码测试的例子

```
int a[10];
```

```
int array1()
```

```
{ int k=10;
```

```
  return a[k]; }
```

访问a[10]越界

```
int array2( int n )
```

```
{ int k;
```

```
  if ( n ) k = 10; else k = 0;
```

```
  return a[k]; }
```

在某些执行路径下，a[k]可能越界。并不是每一条执行路径都会发生越界访问

```
int array3( int k, int n )  
{return a[k]; }
```

程序中没有什么语句可以
推测出来k是否会 ≥ 10

```
int array4( int k, int n )  
{if ( k  $\geq$  10 ) a[0] = n;  
  return a[k]; }
```

根据程序来看, k还是有可能 ≥ 10

```
int array5()  
{ int k;  
  k = -1;  
  return a[k];  
}
```

访问a[-1]越界

```
int array6( int k)
{ int m = 2;
  if( k >= 10 ) k=9;
  while( m-- ) { k++; }
  return a[k];
}
```

根据程序来看, 这里的k应该>10

```
#include <stdlib.h>
void memleak()
{ int *p;
  p = (int *)malloc(10*sizeof(int));
  if (p!=NULL)
    p[0]=12;
  return;
}
```

内存泄漏

```
int *pointer1( int *p )
```

```
{
```

```
    if ( p ) printf( "\\n" );
```

```
    printf( "%d", *p );
```

```
    return p + 2;
```

```
}
```

看来，p可能为NULL

```
int *pointer2( int *p )
```

```
{    printf( "%d", *p );
```

```
    return p + 2;
```

```
}
```

之前没有针对p是否为NULL的判断

```
#include <string.h>
```

```
void f1(int n, int m)
```

```
{  if( n > 0 ) n = 5;  
    else if ( n <= 0 ) n = -1;  
    if( m ) m = 0;  
    else if(!m) m--; }
```

else后面的if条件冗余，可以不写

```
void f2()
```

```
{  char buf[4], *p;  
    strcpy(buf, "abc" );  
    p = buf + strlen( buf );  
    p++;  *p = 'd';  
}
```

p所指位置不能访问。

静态测试没有找出这个缺陷。

缺陷漏报：系统不能够确切知道变量的具体值，不能判断p运行时的具体值。

你来试试?

```
void foo1(int a)
```

```
{
    int k;

    k = 100+a*a;
    if(k=200)
        printf("k=%d", k);
    else
        printf("hello");
}
```

```
void foo2(char grade)
```

```
{
    switch (grade)
    {
        case 'A': printf("85~100\n");
        case 'B': printf("70~84\n");
        case 'C': printf("60~69\n");
        case 'D': printf("<60\n");
        default:printf("error\n");
    }
}
```

```
void foo3( )
```

```
{
    char str1[10], str3[10], *str2="0123456789";
    int i;

    strcpy(str1, str2);
    for(i=0;i<10;i++) str3[i]='a';
    strcpy(str1, str3);
}
```

```
void foo4()
```

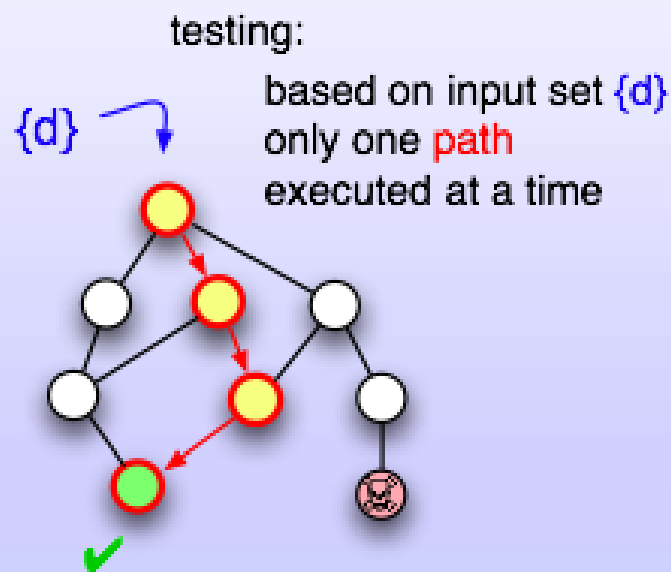
```
{
    int *p1, *p2, *p3;
    p1=(int *)malloc(2*sizeof(int));
    if(!p1) return;
    p2 = p1;
    p2[0]=1;
    p3=(int *)malloc(2*sizeof(int));
    if(!p3) return;
    p3[0]=4;
}
```

3.3 soot

- ◆ Soot是McGill大学的Sable研究小组自1996年开始开发的Java字节码分析工具，它提供了多种字节码分析和变换功能，可以进行过程内和过程间的分析优化，以及程序流图的生成.

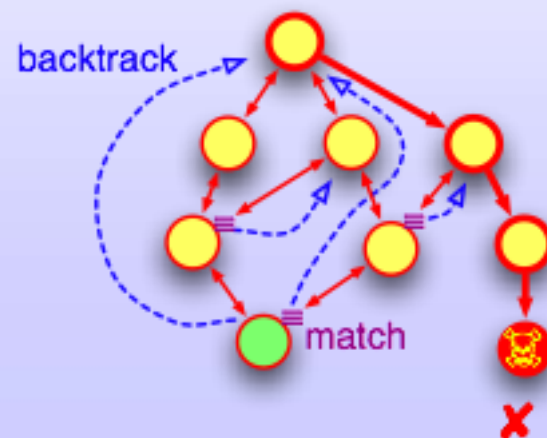
静态测试中的研究问题

- ◆ 目标代码检查（较难）
- ◆ 模型检测（model checking）技术
- ◆ 符号化执行（symbolic execution）技术



model checking:

all program state are explored
until none left or defect found



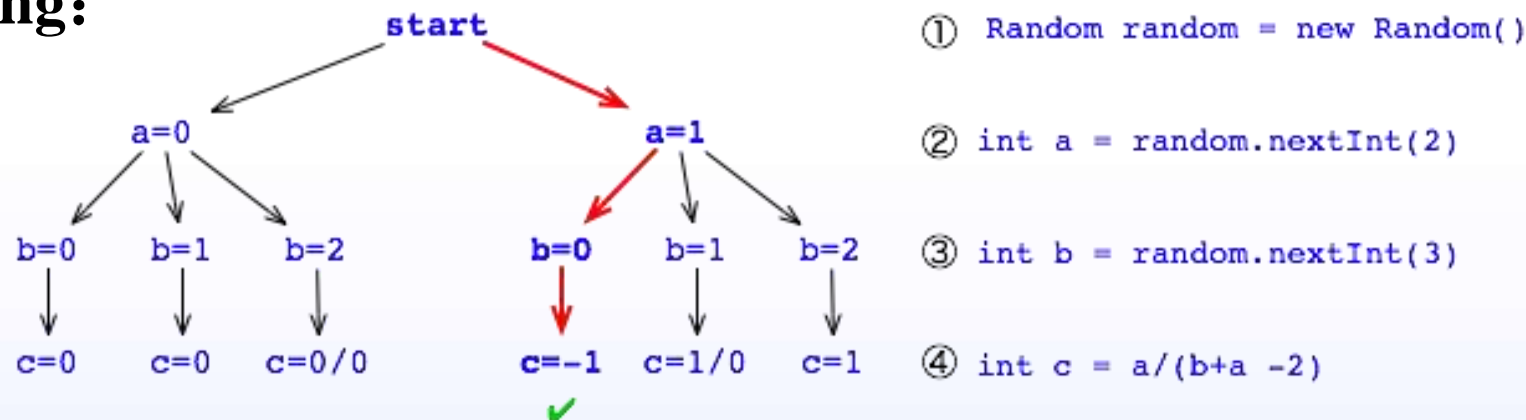
模型检测技术

- **JPF (Java Path Finder)**是一个典型的使用模型检测技术的检测工具，也有符号化执行能力；
- 它可以验证Java程序，可以对Java程序的不同方面进行检测、验证。
- 是一个开源项目

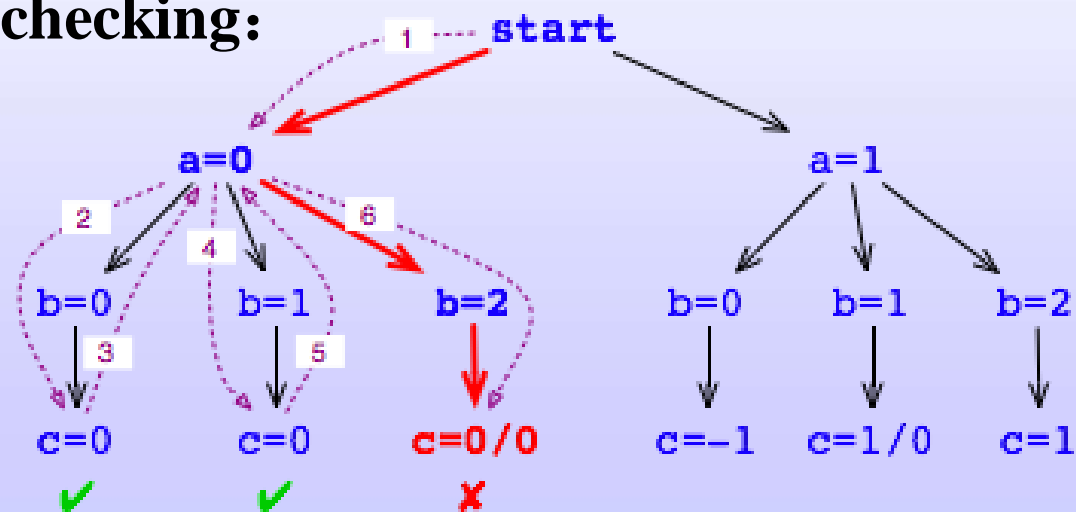
<http://javapathfinder.sourceforge.net/>

使用JPF 进行检测，发现程序缺陷的一个例子：

Testing:



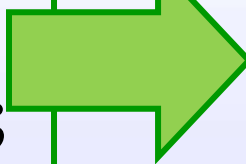
Model checking:



符号化执行技术

```
1. int a, b, c;  
2.  
3. int x = 0, y = 0, z = 0;  
4. if (a) x = -2;  
5. if (b < 5) {  
6.     if (!a && c) y = 1;  
7.     z = 2;  
8. }  
9. assert(x+y+z!=3)
```

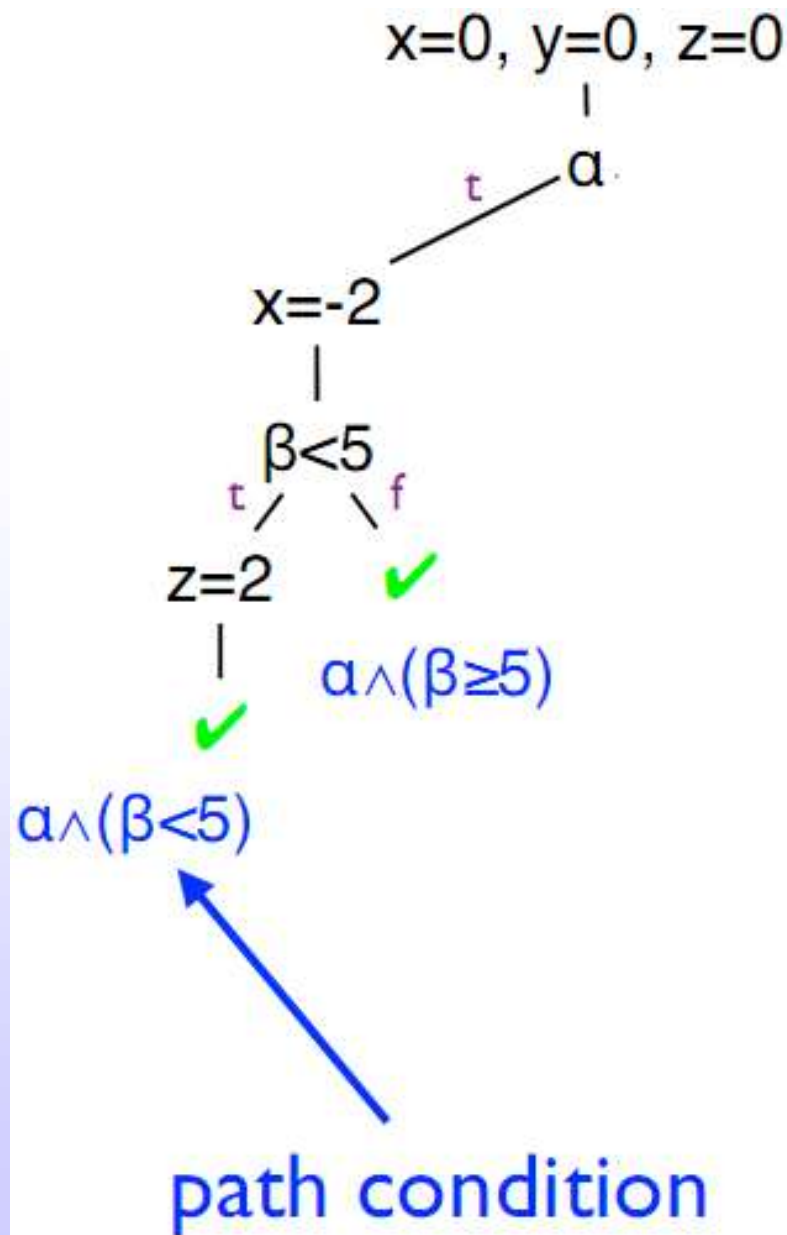
符号化



```
1. int a =  $\alpha$ , b =  $\beta$ , c =  $\gamma$ ;  
2.           // symbolic  
3. int x = 0, y = 0, z = 0;  
4. if (a) x = -2;  
5. if (b < 5) {  
6.     if (!a && c) y = 1;  
7.     z = 2;  
8. }  
9. assert(x+y+z!=3)
```

符号化执行技术

```
1. int a =  $\alpha$ , b =  $\beta$ , c =  $\gamma$ ;  
2.           // symbolic  
3. int x = 0, y = 0, z = 0;  
4. if (a) x = -2;  
5. if (b < 5) {  
6.     if (!a && c) y = 1;  
7.     z = 2;  
8. }  
9. assert(x+y+z!=3)
```





The End
Any Question?

