



软件分析

# 错误定位技术

熊英飞  
北京大学



# 自动化调试

- 静态分析、测试等技术都是为了发现缺陷
- 当一个缺陷被发现之后，如何修复该缺陷？
  - 如何知道缺陷是因为哪行程序的错误导致的？
  - 当定位到有缺陷的语句之后，如何修复该语句的缺陷？



# 基于测试的错误定位

- 输入：
  - 软件系统的源码
  - 一组测试，至少有一个没有通过
- 输出：
  - 一个可能有错误的程序元素列表，根据出错概率排序
- 程序元素可以定义在不同级别上
  - 表达式
  - 语句
  - 方法
  - 类
  - 文件
  - .....



基于测试的错误定位

# 程序切片



# 程序切片

- 给定程序中的一条语句 $S$ ，找到所有可能影响 $S$ 或者 $S$ 可能影响的语句
  - 语句 $S$ 称为切片准则
  - 切片准则也可能是执行到某个位置的某个变量
- 切片分类
  - 后向切片：找到所有影响 $S$ 的语句
  - 前向切片：找到所有 $S$ 可能影响的语句
  - 静态切片：找到在任何输入下可能被影响的语句
  - 动态切片：给定特定输入，只考虑该输入下可能被影响的语句



# 程序切片示例：静态切片

```
a=0;  
b=3;  
If (x>0)  
  {a=1; b++;}  
else  
  {a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...  
...
```

后向  
切片

```
a=0;  
b=3;  
If (x>0)  
  {a=1; b++;}  
else  
  {a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...
```

前向  
切片



# 程序切片示例：静态切片

```
a=0;  
b=3;  
if (x>0)  
  {a=1; b++;  
else  
  {a=2; b--;  
z=a;  
b=b+z;  
b=b-a;  
if (b>0)  
...  
...
```

后向  
切片

```
a=0;  
b=3;  
if (x>0)  
  {a=1; b++;  
else  
  {a=2; b--;  
z=a;  
b=b+z;  
b=b-a;  
if (b>0)  
...  
...
```

前向  
切片



# 程序切片示例：动态切片

```
a=0;  
b=3;  
If (x>0)  
  {a=1; b++;}  
else  
  {a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...  
...
```

后向  
切片

```
a=0;  
b=3;  
If (x>0)  
  {a=1; b++;}  
else  
  {a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...
```

前向  
切片

假设x>0



# 程序切片示例：动态切片

```
a=0;  
b=3;  
If (x>0)  
{a=1; b++;}  
else  
{a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...  
...
```

后向  
切片

```
a=0;  
b=3;  
If (x>0)  
{a=1; b++;}  
else  
{a=2; b--;}  
z=a;  
b=b+z;  
b=b-a;  
If (b>0)  
...
```

前向  
切片

假设x>0



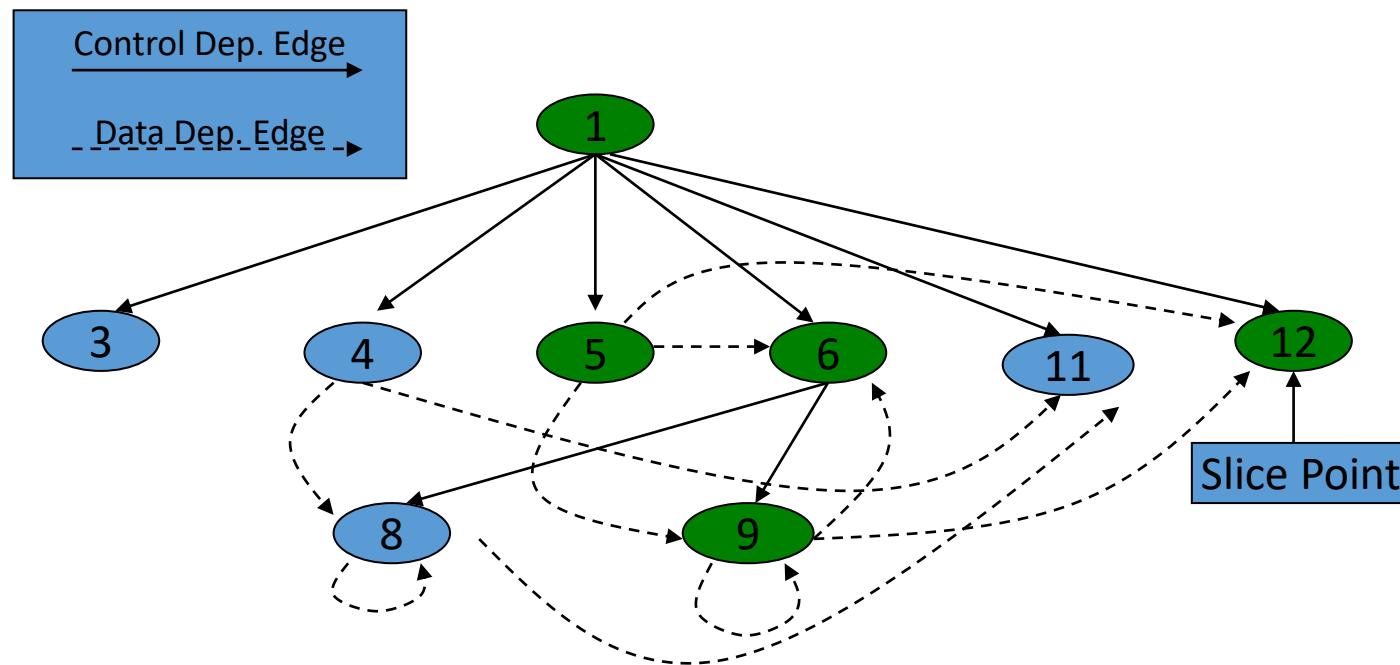
# 实现程序切片

- 三种依赖关系
- 数据依赖：语句a读取了由语句b写入的一个变量
- 控制依赖：语句a是否被执行由语句b的执行结果决定
- 同步依赖：在多线程程序同步时引入的依赖



# 程序依赖图

- 结点为语句，边为依赖关系
- 切片：从出发点开始求图可达性



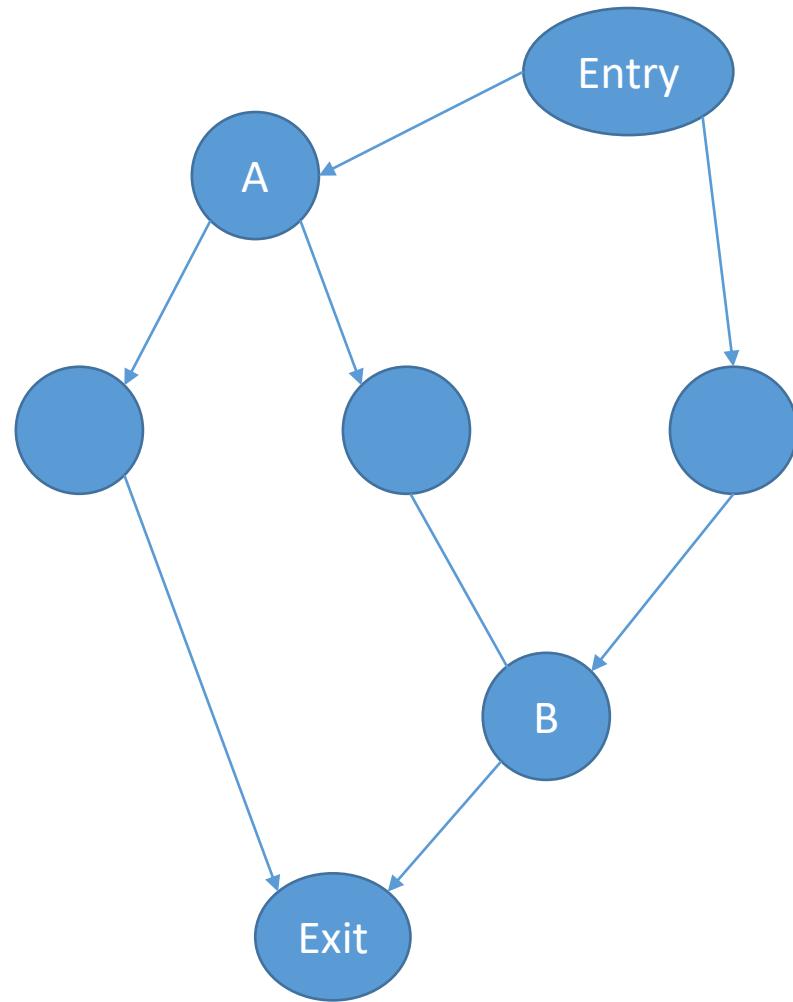


# 构造过程内静态依赖关系

- 数据依赖
  - 假设已经有了指针分析的结果
  - 流非敏感：s1写某个内存位置，s2读该内存位置，则有依赖关系
  - 流敏感：可达定值分析
- 控制依赖
  - 结构化程序
    - If, switch每个分支中的语句依赖if条件
    - While, for循环体中的语句依赖循环条件



# 非结构化程序的例子： A和B有控制依赖吗？





# 复习支配关系/边界

- 结点A支配 (dominate) 结点B: 所有从Entry到B的路径都要通过A
- 结点A严格支配 (Strictly dominate) 结点B: A支配B并且A和B不是一个结点
- 结点A的支配边界中包括B, 当且仅当
  - A支配B的某一个前驱结点—至少有一条路径经过A
  - A不严格支配B—至少有一条路径没有经过A, 且两条在B处汇合
- 存在快速算法可以计算支配边界



# 非结构化程序的控制依赖

- 结点B反向支配 (postdominate) 结点A: 所有从A到Exit的路径都要通过B
- 结点B严格反向支配结点A: B支配A并且B和A不是一个结点
- 结点B的反向支配边界中包括A, 当且仅当
  - B支配A的某一个后继结点
  - B不严格反向支配A
- 将支配边界算法反过来就是反向支配边界算法
- B控制依赖A: B的反向支配边界中包括A



# 构造过程间静态依赖关系

- 数据依赖——不考虑堆上的数据
  - 被调函数入口语句 依赖 调用语句的
  - 调用语句 依赖 被调函数返回语句



# 构造过程间静态依赖关系

- 数据依赖——考虑堆上的数据
  - 计算过程P读的内存位置的集合 $P_r$ 
    - 过程内的语句读x:  $x \in P_r$
    - P调用了Q:  $Q_r \subseteq P_r$
  - 类似可计算过程P写的内存位置集合 $P_w$
  - $P_r$ 当参数处理,  $P_w$ 当返回值处理



# 构造过程间静态依赖关系

- 控制依赖
  - 函数中所有顶层语句依赖函数入口
  - 函数入口依赖调用该函数的语句
- 上下文敏感的过程间依赖关系
  - 通过上下文无关文法的可达性实现



# 构造动态依赖关系

- 打印程序的运行的追踪信息
  - 执行的语句编号
  - 每个语句读写的内存地址
    - 具体位置而非抽象位置
- 数据依赖
  - 语句执行a依赖于语句执行b  $\Leftrightarrow$  a读了一个最近由b写的内存位置
- 控制依赖
  - 实际在执行中发生的静态控制依赖



# 切片与错误

- 出错的语句只可能存在于动态反向切片中
  - 切片准则：发现出错的位置
    - 失败的assert语句
    - 抛出的未捕获的异常
    - 通常为运行追踪信息中的最后一条语句
  - 动态反向切片可以有效的缩小错误语句的范围



基于测试的错误定位

基于测试覆盖的错误定位



# 基于频谱的错误定位

- 使用最广泛的自动化错误定位方法
  - 形式简单，效果较好
- 程序频谱(Program Spectrum)
  - 最早由威斯康星大学Tom Reps于1997年在处理千年虫问题时发明
  - 指程序执行过程中的统计量
- 基于频谱的错误定位
  - 佐治亚理工James Jone, Mary Jean Harrold等人2002把Tom Reps的方法通用化成通用调试方法
  - 主要用到的频谱信息为测试覆盖信息



# 基于频谱的错误定位

- 基本思想
  - 被失败的测试用例执行的程序元素，更有可能有错误
  - 被成功的测试用例执行的程序元素，更有可能没有错误
- 程序元素可以定义在不同的粒度上
  - 基本块
  - 方法
  - 类
  - 文件
  - 可以是语句、表示式吗？



# 例子

		T15	T16	T17	T18
1	int count; int n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	●	●	●	●
2	{return;}	●			
3	src_queue = prio_queue[prio]; dest_queue = prio_queue[prio+1]; count = src_queue->mem_count; <i>if (count &gt; 1) /* Bug */ /* supposed : count&gt;0 */ {</i>		●	●	●
4	n = (int) (count*ratio + 1); proc = find_nth(src_queue, n); if (proc) {		●	●	
5	src_queue = del_ele(src_queue, proc); proc->priority = prio; dest_queue = append_ele(dest_queue, proc); }		●	●	
Pass/Fail of Test Case Execution :					
Pass Pass Pass Fail					



# 计算程序元素的怀疑度

- $a_{ef}$ : 执行语句a的失败测试的数量,  $a_{nf}$ : 未执行语句a的失败测试的数量
- $a_{ep}$ : 执行语句a的通过测试的数量,  $a_{np}$ : 未执行语句a的通过测试的数量
- Tarantula: 
$$\frac{a_{ef}}{a_{ef}+a_{nf}} / \left( \frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}} \right)$$
- Jaccard: 
$$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$$
- Ochiai: 
$$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$$
- D\*: 
$$\frac{a_{ef}^*}{a_{nf}+a_{ep}}, \text{ *通常设置为2或者3}$$
- Naish1: 
$$\begin{cases} -1 & a_{nf} > 0 \\ a_{np} & a_{nf} = 0 \end{cases}$$



# 哪个公式是最好的公式？

- 实验验证
  - 在不同对象上的实验结果并不一致
  - 早期实验认为Ochiai最好，D\*论文认为D\*最好
  - 最新在Java的真实缺陷上的研究认为不同公式之间并无统计性显著差异
    - 语句级别Top-5能平均能定位准18%，Top10为27%
- 理论研究
  - 武汉大学谢晓园等人理论上证明了Naish1优于Ochiai, Ochiai优于Jaccard, Jaccard优于Tarantula，但不存在单一最佳公式
  - 新加坡管理大学David Lo等人做实验验证出和谢晓园不一致的结论



# 其他可能的覆盖信息

- 之前考虑的是对程序元素的直接覆盖
- 也可以考虑其他覆盖类型
- 分支覆盖：对于分支语句的每个选择的覆盖
- 数据流覆盖：对于每个变量定义-使用对的覆盖
- 映射其他覆盖到程序元素的错误
  - 分支覆盖：分支语句的条件是出错语句
  - 数据流覆盖：定义语句是出错语句
- 平均来说效果更好，但并不稳定



# 数据流覆盖的例子

- `a=abs(a);`
- `....`
- `If (...) {`
- `b=sqrt(a);`
- `}`
- 如果只有失败的测试用例覆盖`sqrt (a)`，会认为是`sqrt (a)`错误，但其实错误出现在对`a`取绝对值的时候



# 数据流覆盖的例子

- a=1;
  - ....
  - If (...) {
  - b=sqrt(-1);
  - }
- 
- 但是正确性是没有保障的



基于测试的错误定位

基于状态覆盖的错误定位



# 程序元素的粒度如何选择？

- 粒度越细
  - 缺陷定位的结果越精细，对测试信息的利用越精确
  - 单个元素上覆盖的测试数量越少，统计显著性越低
- 常见情况举例
  - 方法级别
  - 基本块级别



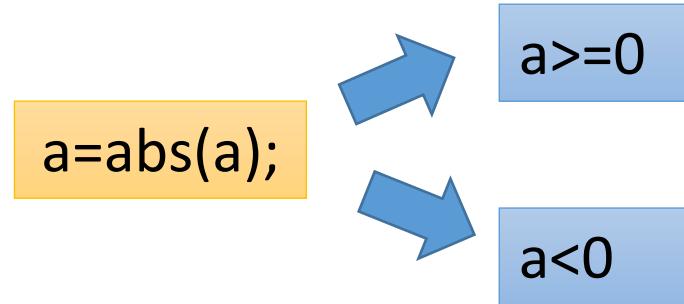
# 能否比语句更精细?

- 状态级别：程序的每个执行状态作为一个元素
  - 定位结果最精细，对测试的利用最充分
  - 几乎不会有两个测试覆盖同样的状态
- 能否找到一个折中方案？



# 基于状态覆盖的错误定位

- `a=abs(a);`
- ....
- `If (...) {`
- `b=sqrt(a);`
- `}`



- 该语句执行完系统的状态可以分成两组抽象状态
  - 通过的测试只有 $a>=0$ 的状态。
  - 只有失败的测试有 $a<0$ 的状态。
- 可以判断出 $a<0$ 是缺陷状态，引入该状态的语句为缺陷语句。



# 状态抽象方案1:预定义谓词

- 定义一些常见谓词(predicate)，每个谓词的不同状态  
把具体状态划分成抽象状态
  - 不同谓词形成的划分可以重叠
- 常见谓词
  - 对整形变量a
    - $a>0$
    - $a<0$
    - $a==0$
  - 对布尔变量b
    - $b==true$
    - $b==false$
  - 对对象o
    - $o==null$
    - $o!=null$



# 状态抽象方案2:谓词挖掘

- 从通过的测试执行中挖掘谓词而不是自定义
- Daikon: 从程序的运行过程中挖掘不变式
  - 给定不变式模板和程序中的位置, Daikon监控程序的执行, 并尽量多地挖掘出不变式
- 常见模板
  - 假设 $x, y, z$ 是变量,  $a, b, c$ 是常量
  - $x \geq c$
  - $ax + by + c = 0$
  - 某数组是按升序排列



# 状态抽象方案3:定值位置

- 根据变量定值的位置定义抽象状态
- 给定语句 $s$ , 令该语句读取的变量集合为 $V$
- 该位置的抽象状态为
  - $\prod_{v \in V}$  执行到 $s$ 时 $v$ 可能的定值位置的集合
- 和基于数据流覆盖的情况有什么不同?
  - 考虑Def-Use关系的组合而不是单个Def-Use关系



# 如何给抽象状态的出错可能性打分？

- 打分方法1：
- 统计性调试公式
  - 假设令predicate为真的状态为a，为假的状态为b
  - $$\frac{\frac{1}{a_{ef}} - \frac{a_{ef} + b_{ef}}{a_{ep} + a_{ef} + b_{ep} + b_{ef}} + \frac{\log F}{\log a_{ef}}}{2}$$
- 各种spectrum的公式
  - 谓词为真看做覆盖，谓词为假看做不覆盖
  - 最新研究表明效果显著好于统计性调试公式



# 打分方法2:状态迁移的概率

- 给定程序依赖图上的两个结点  $S_a$  和  $S_b$ ，  $S_a$  直接依赖于  $S_b$ ， 则  $S_a$  的任意抽象状态  $a$  直接依赖  $S_b$  的抽象状态  $b$ ， 记为  $a \mapsto b$
- 对于任意满足  $a \mapsto b$  的抽象状态  $a$  和  $b$ ， 基于通过的测试统计条件概率
  - $P(a \mid b)$
- 对于失败测试执行中的每一个状态  $a$ ， 计算出错可能性
  - $\frac{1}{\min_{a \mapsto b} P(a|b)}$



# 打分方法3：机器学习

- 基本思路：不同类型的谓词具有不同的预测能力
  - $n!=\text{null}$ 可能有很强的预测能力
  - $n>=25$ 不一定
- 采用机器学习来给不同的谓词分的类打分
- 具体方法：
  - 把不同的谓词分类
    - Daikon已经分好了311个类别
  - 令 $a$ 为对应某个谓词 $P$ 为真的抽象状态,  $b$ 为 $P$ 为假的抽象状态, 假定 $P$ 的类别为 $S$ , 添加一系列的二元特征
    - $S_{p \neg f}$ , 为真表示 $b_{ep} = 0, a_{ep} > 0$ , 且 $b_{ef} > 0$ , 即 $P$ 在通过测试中恒成立, 但在失败测试中不一定成立
    - $S_{\neg pf}$ , 为真表示 $b_{ef} = 0, a_{ef} > 0$ , 且 $b_{ep} > 0$ , 即 $P$ 在通过测试中不一定成立, 但在失败测试中恒成立
  - 通过机器学习用这组特征预测语句出错的概率。



# 基于状态覆盖的错误定位代 表方法

- 统计性调试：预定义谓词+统计调试公式
  - 斯坦福Ben Liblit 和Alex Aiken等人于2003年提出
- PredFL：预定义谓词+基于频谱错误定位公式
  - 北京大学姜佳君和王然等提出，效果显著超过原始统计性调试
- 概率依赖图：定值位置+状态迁移概率
  - 佐治亚理工Mary Jean Harrold等人于2010年（Mary去世前三年）提出
- Savant：谓词挖掘+机器学习
  - 新加坡管理大学David Lo等人于2016年提出



基于测试的错误定位

基于变异的错误定位



# 变异分析

- 变异：对程序的任意随机修改
- 变异分析：收集变异后程序上原测试用过与否的信息的分析
- 变异分析被广泛应用于测试领域来衡量一个测试集的好坏
  - 如果一个测试集中任意测试在一个变异后的程序上执行失败，称为该变异体被这个测试杀死
  - 能杀死越多变异体的测试集越好



# 常见变异测试工具

- C
  - Milu
- Java
  - MuJava：基础变异测试工具，支持变异算子较完善
  - Javalanche：支持Mutation Schemata的加速
  - Major：支持预先过滤测试执行的加速，支持变异算子较少
  - PIT：商业工具，功能最完善速度最快



# 关于变异的假设

- 假设1：变异错误语句时
  - 失败测试用例输出发生变化的概率>通过测试用例输出发生变化的概率
- 假设2：变异正确语句时
  - 失败测试用例输出发生变化的概率<通过测试用例输出发生变化的概率
- 假设3：导致失败测试变成通过的概率
  - 变异错误语句时>变异正确语句时
- 假设4：导致通过测试变成失败的概率
  - 变异正确语句时>变异错误语句时

Metallaxis

MUSE



# Metallaxis

- 卢森堡大学的Yves Le Traon教授等人2012年提出
- $m$ : 变异体,  $m_f$ : 输出发生变化的失败测试数,  $m_p$ : 输出发生变化的通过测试数,  $F$ : 原失败测试总数
- 变异体可疑度公式
  - $$\frac{m_f}{\sqrt{F*(m_f+m_p)}}$$
  - 与Ochiai类似, 但主要考虑输出的变化
- 程序元素可疑度为该元素上的最高变异体的可疑度



# MUSE

- 韩国科学技术院Moonzoo Kim和英国UCL大学Shin Yoo于2014年提出
- $m$ : 变异体,  $m_{f2p}$ :  $m$ 上从失败变成通过的测试数,  $m_{p2f}$ :  $m$ 上从通过变成失败的测试数
- 变异体可疑度公式
  - $m_{f2p} - m_{p2f} \frac{\sum_m m_{f2p}}{\sum_m m_{p2f}}$
- 程序元素可疑度为该元素上的变异体的可疑度平均



# 基于变异vs基于频谱

- 在基于频谱的错误定位中，不同测试只要覆盖了语句，对结果的效果就是相同
- 但不同测试受同一个语句的影响是不同的
  - 不同测试触发错误的概率不同
  - 不同测试传播错误的概率不同
  - 不同测试捕获错误的概率不同
- 基于变异的错误定位实际依靠变异捕获了测试和语句之间的关系



基于测试的错误定位

构造正确执行状态



# 动机

- 在MUSE中，如果有一个变异让失败的测试通过，同时通过的测试仍然通过，那么该变异有最高的怀疑度
- 换句话说，该变异很可能是正确的补丁
- 动机：直接分析出这样的变异，然后将能产生出的语句当作怀疑度最高的语句
- 困难：直接分析出比较困难
- 解决方案：不分析出变异本身，只分析出该变异对系统状态的影响



# 谓词翻转 Predicate Switching

- 2006年由普度的张翔宇教授提出
- 假设出错的是一个布尔表达式
  - 不考虑表达式的副作用
- 该表达式修改后，必然在原失败测试中至少一次求值返回翻转的结果
  - $\text{true} \rightarrow \text{false}$
  - $\text{false} \rightarrow \text{true}$
- 依次翻转失败测试中表达式求值结果，如果测试通过，则说明对应表达式可能有错误



# 天使调试 Angelic Debugging

- 2013年由华盛顿大学的Emina Torlak提出
- 如何把谓词翻转从布尔表达式扩展到任意表达式上？如int, float, double等
- 天使性条件：存在常量c（天使值）把表达式的求职结果替换成c，失败的测试变得通过
- 是否满足天使性条件就代表表达式很可能有缺陷呢？



# 天使性条件

$f(a)$ :

$b = a + 1;$

$c = b + 1;$

$d = c ++;$

失败测试:

$f(1);$

`assert(d=4);`

以上每个表达式都满足条件



# 完整天使调试

- 基础天使调试条件对应原来目标的前一半：失败的测试变得通过
- 利用后一半：通过的测试仍然通过
- 假设：对表达式进行修改后，表达式在所有测试中都会得到不同的结果
  - 比较强的假设，但对数值型表达式有较大概率成立
- 灵活性条件：对于所有通过的测试中的每一次表达式求值，都可以把求值结果换成一个不同的值，并且测试仍然通过。
- 可疑语句需要同时具有天使性和灵活性



# 完整天使调试

f(a):

b = a+1;

c = b+1;

d = c++;

为什么谓词翻转不需要灵活性条件?

失败的测试:

f(1);

assert(d=4);

通过的测试:

f(2);

assert(c=5);

只有c++是可疑的表达式



# 如何判断天使性和灵活性？

- 采用符号执行
- 首先选定表达式
- 将表达式的返回值用符号 $v$ 替换
- 从该表达式所在语句开始符号执行，考虑所有路径（循环最多执行 $n$ 次），并收集路径约束和最后test oracle形成的约束求解
- 对于通过的测试，还要添加约束 $v \neq c$ ，其中 $c$ 是原来运行的结果
- 由于符号执行的开销，天使调试无法应用到大型程序上



# 算法式调试



# 算法式调试

# Algorithmic Debugging

- 之前的所有方法都是试图直接找出错误位置
- 交互式调试：通过询问程序员来定位错误位置
- 算法式调试
  - 1983年由Ehud Shapiro在《算法式调试》一书中提出
  - 主要针对函数语言设计，在Haskell等函数语言上广泛实现
  - 主要通过询问“是”或者“否”的问题找到出错函数

算法调试示例

```
main = in sort [2,1,3]

in sort [] = []
in sort (x:xs) = insert x (in sort xs)

insert x [] = [x]
insert x (y:ys) = if x>=y then (x:y:ys)
                  else (y:(insert x ys))
```

Starting Debugging Session...

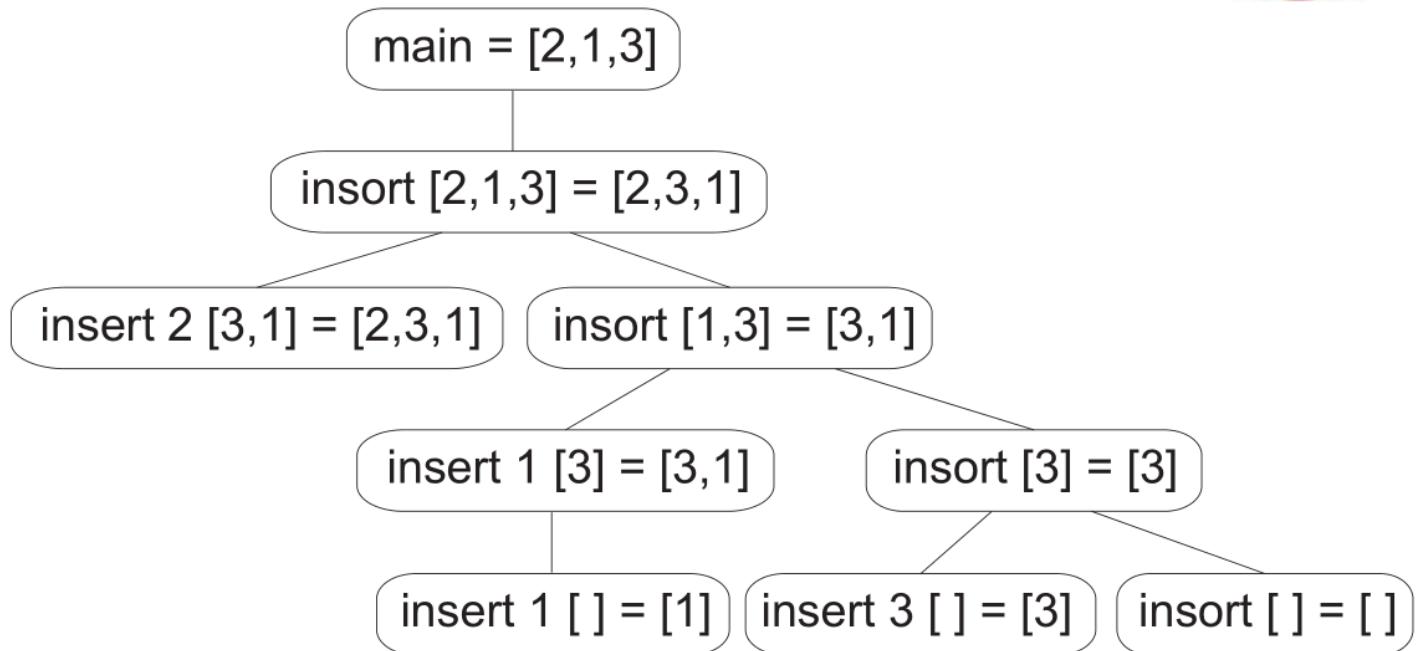
- (1) main = [2,3,1]? NO
- (2) in sort [2,1,3] = [2,3,1]? NO
- (3) in sort [1,3] = [3,1]? NO
- (4) in sort [3] = [3]? YES
- (5) insert 1 [3] = [3,1]? NO
- (6) insert 1 [] = [1]? YES

Bug found in rule:

```
insert x (y:ys) = if x>=y then (x:y:ys)
                  else (y:(insert x ys))
```



# 执行树 Execution Tree



```
main = in sort [2,1,3]
```

```
in sort [] = []
in sort (x:xs) = insert x (in sort xs)
```

```
insert x [] = [x]
insert x (y:ys) = if x>=y then (x:y:ys)
                  else (y:(insert x ys))
```



# 二分查找算法

- 当用户对某个结点回答“是”，该结点为根子树可以排除
- 当用户对某个结点回答“否”，该结点为根子树以外的结点可以排除
- 一个基本思路是问尽量少的问题
  - 即每次选择结点数最接近总结点一半的子树询问



# 算法式调试的其他改进

- 利用一次回答推出更多的信息
  - 利用重复的结点
  - 利用程序中的其他约束
- 减少人思维中的跳转，尽量同一时间针对一个函数问问题



# 差异化调试



# 差别化调试Delta Debugging

- 1999年由德国Saarland大学Andreas Zeller提出
- 场景1：
  - 昨天，测试还正常通过
  - 晚上，加班改了1000行代码
  - 今天，测试不通过了
  - 哪些修改是罪魁祸首？



# 更多场景

- 场景2
  - 写了一个编译器
  - 用户编译了一个1000万行代码的项目
  - 编译器崩溃了
  - 哪些输入代码导致编译器崩溃？
- 场景3
  - 输入a崩溃了，输入b没有崩溃
  - 在某个关键函数进入之前，系统中有1000个内存位置存有数据
  - 哪些内存位置存的数据导致输入a崩溃了？



# 基本思路

- 比较两个版本
  - 场景1：昨天的代码，今天的代码
  - 场景2：空白输入，失败输入
  - 场景3：测试b的状态，测试a的状态
    - 前者测试通过，后者测试不通过
- 找到最小修改集合C
  - 将C应用到前者上测试不通过
- 基本方法：集合上的二分查找



# ddmin 问题定义

- 输入：
  - 所有可能修改的集合  $C$
  - 测试函数  $test: 2^C \rightarrow \{x, \checkmark, ?\}$ , 满足  $test(\emptyset) = \checkmark$
  - 集合  $c_x \subseteq C$ , 满足  $test(c_x) = x$
- 输出：集合  $c'_x \subseteq c_x$ , 满足
  - $test(c'_x) = x$
  - $\forall c \in c'_x, test(c'_x - \{c\}) \neq x$ 
    - 并非完备的的最小定义, 但完备的做不出来

ddmin 算法 运行示例

```
1 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
2 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
3 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
4 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
5 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
6 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
7 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
8 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
9 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
10 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
11 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
12 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
13 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
14 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
15 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
16 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
17 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
18 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
19 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
20 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
21 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
22 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
23 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
24 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
25 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✓
26 <SELECT_NAME="priority" MULTIPLE_SIZE=7> ✗
```



# ddmin算法

The *ddmin* algorithm is defined as  $ddmin(c_{\mathbf{x}}) = ddmin'(c'_{\mathbf{x}}, 2)$  with

$ddmin'(c'_{\mathbf{x}}, n)$

$$= \begin{cases} c'_{\mathbf{x}} & \text{if } |c'_{\mathbf{x}}| = 1 \\ ddmin'(c'_{\mathbf{x}} \setminus c_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1..n\} \cdot \text{test}(c'_{\mathbf{x}} \setminus c_i) = \mathbf{x} \\ & \quad \text{("some removal fails")} \\ ddmin'(c'_{\mathbf{x}}, \min(2n, |c'_{\mathbf{x}}|)) & \text{else if } n < |c'_{\mathbf{x}}| \text{ ("increase granularity")} \\ c'_{\mathbf{x}} & \text{otherwise} \end{cases}$$

where  $c'_{\mathbf{x}} = c_1 \cup c_2 \cup \dots \cup c_n$  such that  $\forall c_i, c_j \cdot c_i \cap c_j = \emptyset \wedge |c_i| \approx |c_j|$  holds.

注意：99年Delta Debugging第一篇论文中的dd算法是错误的



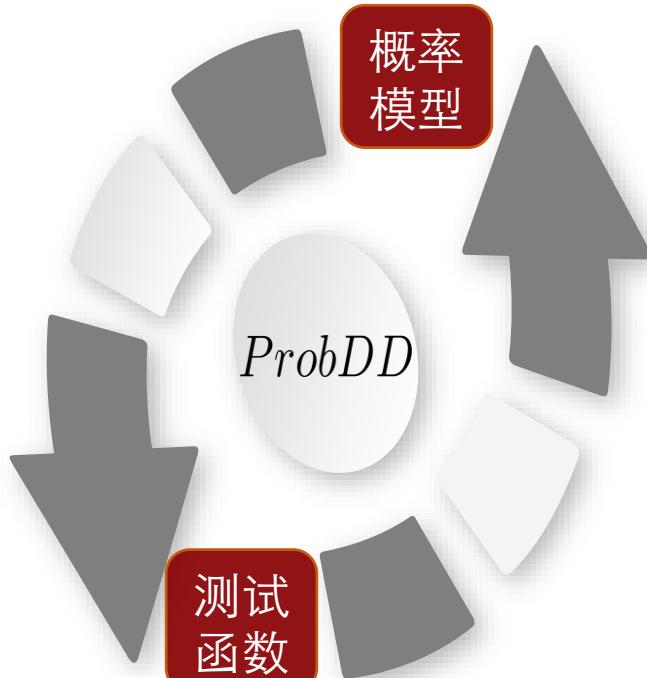
# ddmin算法的缺点

- 实际应用中效果不尽人意
  - 一次差异化调试常常需要数小时或数十小时
  - 结果可能数倍于最优结果
- 采用固定的尝试顺序
  - 无法充分利用测试结果包含的信息
- ProbDD算法
  - 根据测试结果调整下一次的策略
  - 实验中运行时间和输出结果都约是ddmin的1/2

# 概率差异化调试ProbDD



收益为期望删掉的元素数量  
计算最大化收益的测试方案



根据测试结果，计算后验概率，  
更新模型

	s1	s2	s3	s4	s5	s6	s7	s8	
1	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
2	s1	s2	s3	s4	s5	s6	s7	s8	F
3	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
4	s1	s2	s3	s4	s5	s6	s7	s8	T
5	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
6	s1	s2	s3	s4	s5	s6	s7	s8	F
7	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	T
9	0.6119	0	0.6119	0	0	0	0	0.6119	
10	s1	s2	s3	s4	s5	s6	s7	s8	F
11	0.6119	0	1	0	0	0	0	0.6119	
12	s1	s2	s3	s4	s5	s6	s7	s8	T
13	0	0	1	0	0	0	0	0.6119	
14	s1	s2	s3	s4	s5	s6	s7	s8	F
15	0	0	1	0	0	0	0	1	

每个元素有一个概率值决定其是否必要，不同元素必要的概率彼此独立，当且仅当所有必要元素都在的时候测试通过



# 模型更新方法

- 精确计算：根据测试结果计算后验概率

$$\Pr(\theta_i = 1 | T_1, T_2, \dots, T_n) = \frac{\Pr(\theta_i = 1, T_1, T_2, \dots, T_n)}{\Pr(T_1, T_2, \dots, T_n)}$$



计算方法：

- (1) 枚举与所有测试历史结果相容的子序列
- (2) 将每一个子序列是最优子序列的概率求和并带入公式

→ Weighted Model Counting  
难解！

- 近似计算：只使用当前测试的结果计算后验概率并更新模型，计算可得更新规则：

$$p_i = \begin{cases} p_i & \text{第 } i \text{ 个元素 } \in X \\ 0 & \text{第 } i \text{ 个元素 } \notin X \wedge \phi(X) = T \\ \frac{p_i}{1 - \prod_i (1 - p_i)^{1-x_i}} & \text{第 } i \text{ 个元素 } \notin X \wedge \phi(X) = F \end{cases}$$

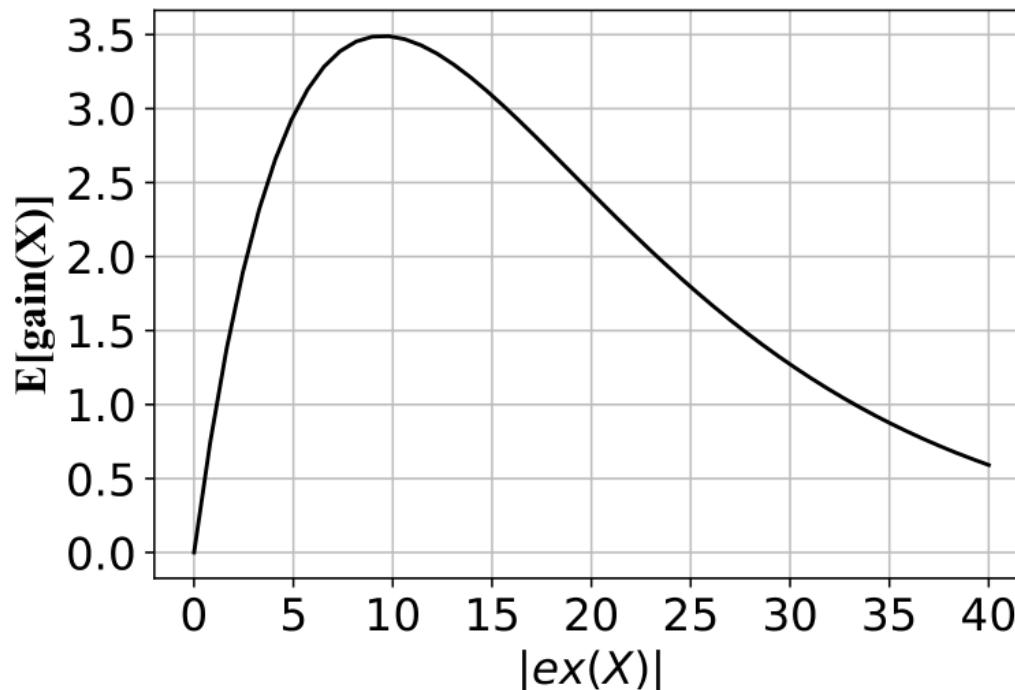
# 计算收益最大的测试方案



- 定义在子序列  $X$  上进行测试的增益  $gain$  为这个测试通过时新排除的元素数量:

$$gain(X) = \begin{cases} |X^{ex}| & \phi(X) = T \\ 0 & \phi(X) = F \end{cases}$$

- 一个测试的期望增益为:  $\mathbb{E}[gain(X)] = |X^{ex}| Pr(\phi(X) = T) = |X^{ex}| \prod_i (1 - p_i)^{1-x_i}$
- 将测试子序列选择目标定为选择子序列  $X$  使得  $\mathbb{E}[gain(X)]$  最大化。





# ProbDD的理论性质

## 高效性

- 时间复杂度  $O(n)$
- $ddmin$ 的复杂度  $O(n^2)$

## 正确性

- 返回结果保留所性质

## 最优性

- 如原问题单调，则ProbDD的结果是极小的
- 如原问题还具有无二义性，则ProbDD的结果是最小的