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

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

```
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)
…
```

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

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

```
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)
…
```

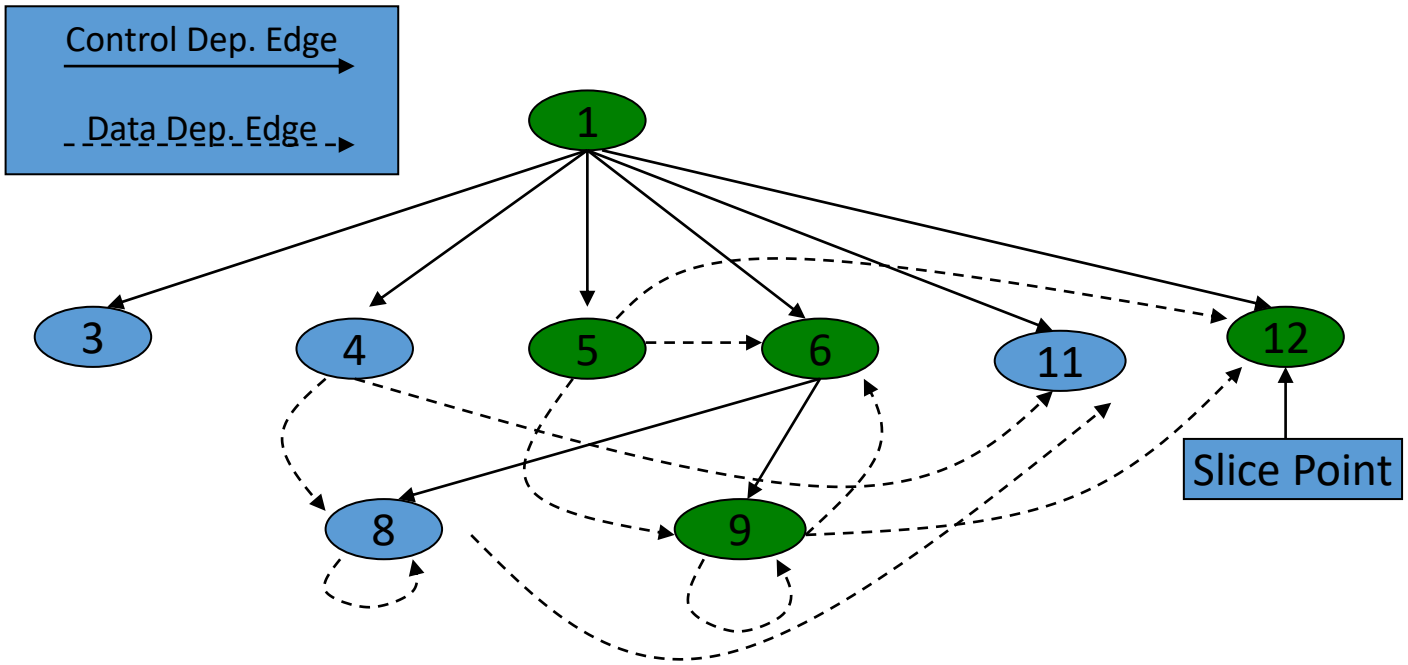
假设x>0

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

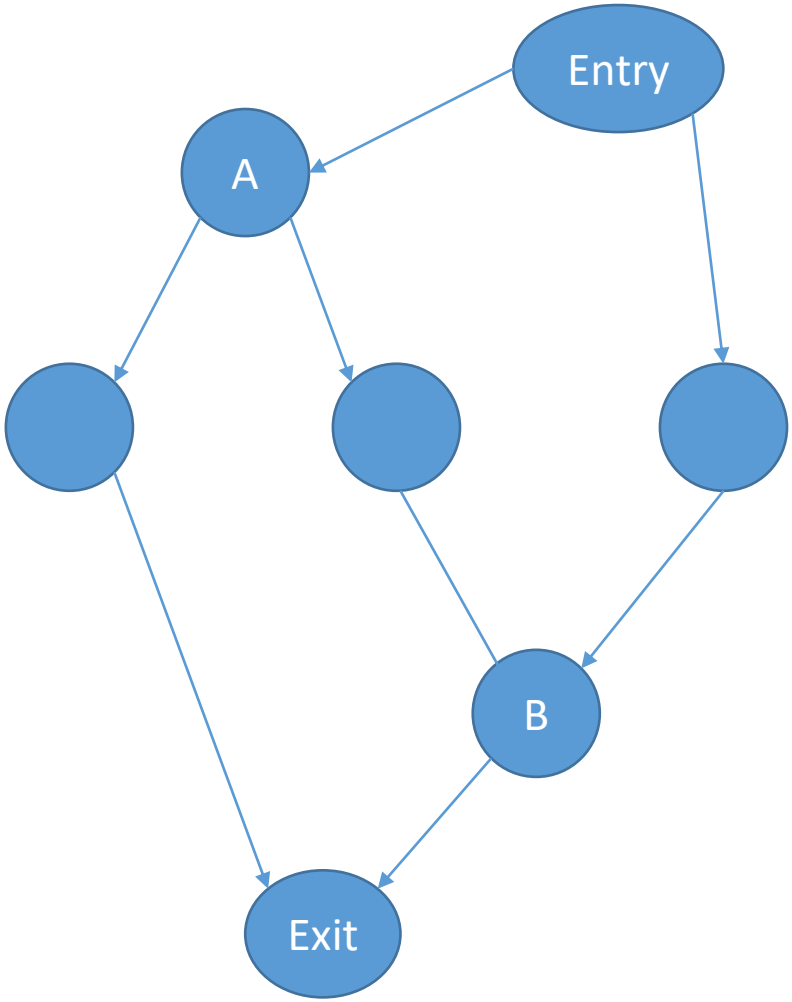
- 结点为语句，边为依赖关系
- 切片：从出发点开始求图可达性



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 ⇔ 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

加州大学埃文分校
James Jone教授
原佐治亚理工大学博士生

佐治亚理工大学
Mary Jean Harrold教授

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;
	if (count > 1) /* Bug*//* supposed : count>0*/ {
			●	●	●

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*: $\frac{a_{ef}^{*}}{a_{nf}+a_{ep}}$，*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前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
- 定位到某个特定覆盖单位，说明影响该单位的语句可能有错
  - 分支覆盖：对应分支可能有错，或者分支语句的条件可能有错
  - 数据流覆盖：对应的定义-使用对可能有错

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
- 数据流覆盖可以定位到该def-use对，供程序员查看

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

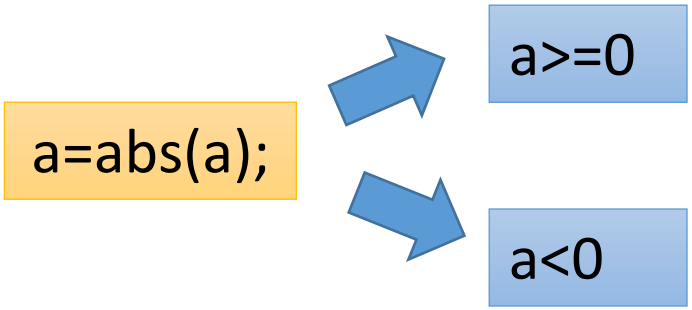
- 常见情况举例
  - 方法级别
  - 基本块级别

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: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

历史

- 基于状态覆盖和基于频谱的方法是同时提出
  - 基于状态覆盖的方法是由程序设计语言社区提出，叫做统计性调试
- 统计性调试方法提出了自己的打分公式
  - 假设令predicate为真的状态为a，为假的状态为b
  - $$\frac{2}{\frac{1}{\frac{a_{ef}}{a_{ep}+a_{ef}}-\frac{a_{ef}+b_{ef}}{a_{ep}+a_{ef}+b_{ep}+b_{ef}}}+\frac{\log F}{\log a_{ef}}}$$
- 后续研究证明该公式远远不如基于频谱的公式
  - Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, Lu Zhang. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. ASE'19: 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, California, United States , November 2019

斯坦福大学
Alex Aiken教授
统计性调试提出者

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

卢森堡大学
Yves Le Traon教授

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

韩国科学技术院
Moonzoo Kim教授

英国UCL大学
Shin Yoo教授

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

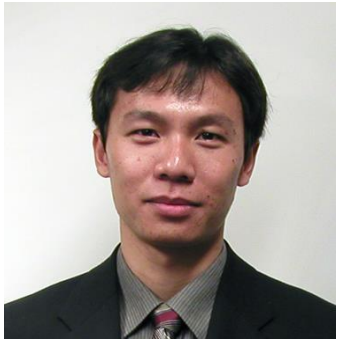
构造正确执行状态

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年由普渡大学张翔宇教授提出
- 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
  - true -> false
  - false -> 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



普渡大学
张翔宇教授

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?

华盛顿大学
Emina Torlak教授

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

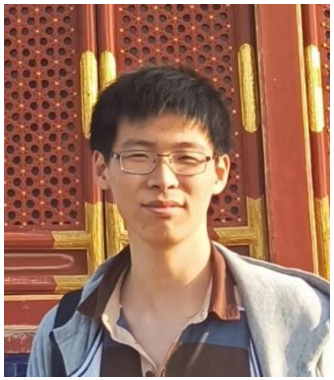
基于测试的错误定位

错误概率建模

动机

- 基于变异的错误定位试图捕获测试和语句出错的关系
  - 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
- SmartFL方法：2022年由北京大学熊英飞团队提出
  - 直接根据程序语义建模这种关系
    - 即建模每个语句出错时特定测试失败的概率
  - 根据测试执行结果，计算每个语句出错的后验概率
  - 效果上超过基于变异的缺陷定位，效率上接近基于频谱的错误定位

北京大学吴宜谦
SmartFL的主要提出者



北京大学曾沐烙
SmartFL的主要提出者

示例

```java
public class CondTest {
    public static int foo(int a) {
        if (a <= 2) { // buggy, should be a < 2
            a = a + 1;
        }
        return a;
    }

    @Test
    void pass() {
        assertEquals(2, foo(1));
    }

    @Test
    void fail() {
        assertEquals(2, foo(2));
    }
}
```

$$P(S_3) = P(S_4) = 0.5$$

$$P(V_2^p) = 1$$

$$P(V_3^p \mid S_3 \wedge V_2^p) = 1$$

$$P(V_3^p \mid \neg S_3 \vee \neg V_2^p) = 0.5$$

$$P(V_4^p \mid S_4 \wedge V_2^p \wedge V_3^p) = 1$$

$$P(V_4^p \mid \neg S_4 \vee \neg V_2^p \vee \neg V_3^p) = 0.01$$

$$P(V_2^f) = 1$$

……

$$P(S_4 \mid V_4^p, \neg V_4^f) = ?$$

$$P(S_3 \mid V_4^p, \neg V_4^f) = ?$$

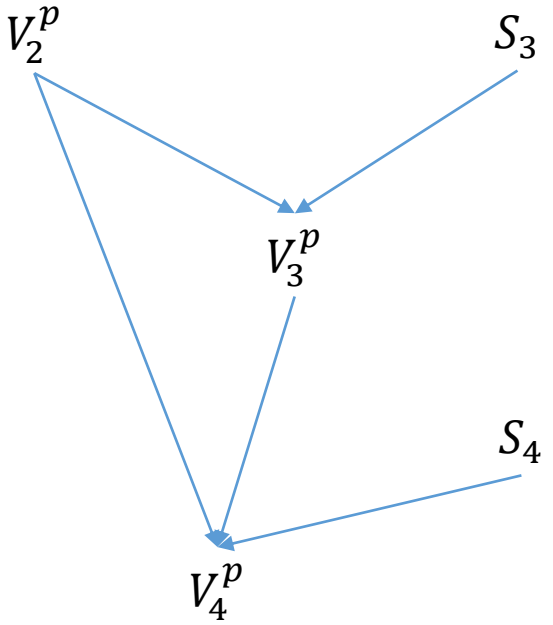
二元随机变量：

V_i^t：测试t中第i行的表达式计算结果正确

S_i：第i行的语句正确

贝叶斯网

- 基础有向图概率模型
- 两个结点连边->后继节点和前驱结点有因果关系
- 两个结点不连边->后继节点和前驱结点没有因果关系
- 网络联合分布可以表示为每个随机变量相对前驱结点的条件概率的连乘
  - $P(V_2^p, V_3^p, V_4^p, S_3, S_4) = P(V_2^p)P(S_3)P(S_4)P(V_3^p \mid V_2^p, S_3)P(V_4^p \mid S_4, V_3^p, V_2^p)$
- 理论上可以依据联合分布计算边缘分布
  - 存在多种快速计算算法，比如信念传播



未执行修正

```
if (z > 0)
  x++;
else
  y++;
```

- 如果z>0的条件执行结果有错，那么不仅仅x被设置成错误的值，y也被设置成错误的值
- 静态分析这种没有潜在被执行的路径的影响，添加对应条件概率

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
  - 1982年在Ehud Shapiro博士论文提出，获得ACM杰出博士论文奖，之后出版为《算法式调试》一书
  - 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

以色列魏茨曼科学研究所
Ehud Shapiro教授
博士毕业于耶鲁大学

算法调试示例

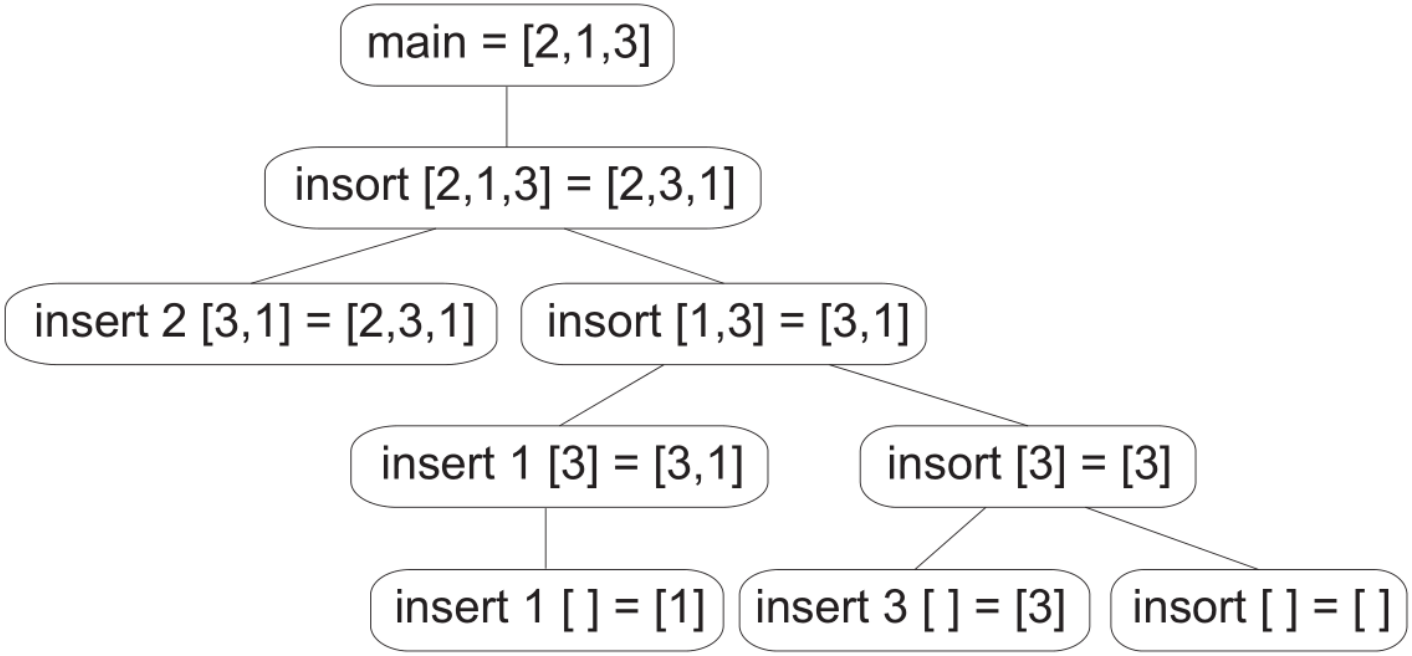
```
main = insort [2,1,3]

insort [] = []
insort (x:xs) = insert x (insort 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)  insort [2,1,3] = [2,3,1]? NO
(3)  insort [1,3] = [3,1]? NO
(4)  insort [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 = insort [2,1,3]

insort [] = []
insort (x:xs) = insert x (insort 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?

Saarland大学
Andreas Zeller教授

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$test: 2^C \to \{\times, \checkmark\}$，满足$test(\emptyset) = \checkmark$
  - 集合$c_\times \subseteq C$，满足$test(c_\times) = \times$
- 输出：集合$c'_\times \subseteq c_\times$，满足
  - $test(c'_\times) = \times$
  - $\forall c \in c'_\times, test(c'_\times - \{c\}) \neq \times$
    - 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 $ddmin(c_{\times}) = ddmin'(c'_{\times}, 2)$ with

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

where $c'_{\times} = c_1 \cup c_2 \cup \cdots \cup c_n$ 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

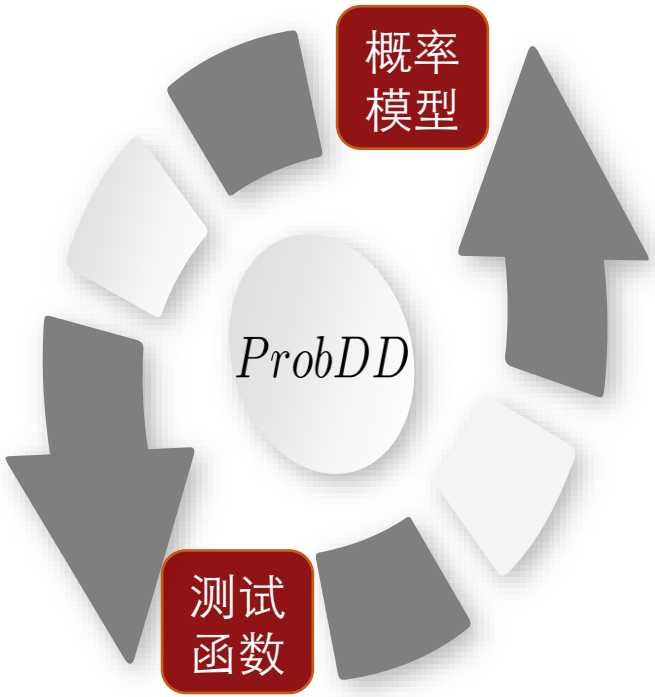
- ProbDD算法
  - 2021年由北京大学熊英飞团队提出
  - 根据测试结果调整下一次的策略
  - 实验中运行时间和输出结果都约是ddmin的1/2

北京大学王冠成
ProbDD的主要提出者

北京大学沈若冰
ProbDD的主要提出者

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



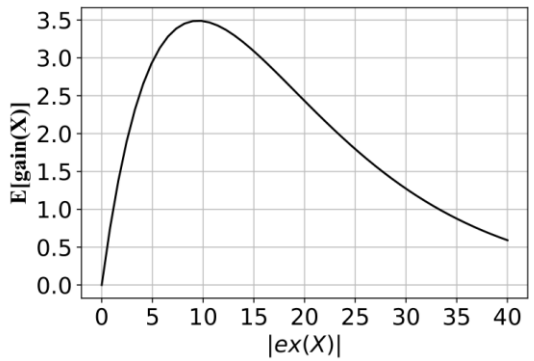
根据测试结果，计算后验概率，
更新模型

	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	×
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	✓
	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失败

选择删除方案

- 期望删除元素个数
  - $|D|\prod_{d\in D}(1-p_d)$
  - D为待删除集合
  - p_d为d元素必要的概率
- D集合大小相同时，所选元素概率越小越好
- 可以证明集合大小存在一个极值点



	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	×
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0	0	1	0	0	0	0	1	

更新模型概率

- 根据测试结果更新概率模型
- 测试通过
  - $p_d := P(p_d \mid \times) = 0$
- 测试未通过
  - $p_d := P(p_d \mid \checkmark) = \frac{p_d}{1 - \prod_{d \in D}(1 - p_d)}$

	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	×
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	×
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	✓
	0	0	1	0	0	0	0	1	

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

- 返回结果保留所需性质

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