缺陷 定位技术

熊英飞
北京大学

报告人介绍 – 熊英飞

- 2000~2004，电子科技大学本科
- 2004~2006，北京大学研究生
  - 导师：梅宏、杨芙清
- 2006~2009，日本东京大学博士
  - 导师：胡振江、武市正人
- 2009~2011，加拿大滑铁卢大学博士后
  - 导师：Krzysztof Czarnecki
- 2012~，北京大学“百人计划”研究员（Tenure-Track）
- 研究方向：软件分析、编程语言设计
- 主页：http://sei.pku.edu.cn/~xiongyf04/

北京大学软件工程研究所

- 国内最早开展软件工程研究、规模最大、最有影响力的软件工程研究团队
- 院士三名（含双聘一名），IEEE Fellow 2名，千人计划1名，博士生导师14名，硕士生导师13名
- 在软件工程顶级会议发表论文数大陆第一
- 获得ACM SIGSOFT杰出论文奖三次，占大陆获奖数一半
- 多名电子科技大学的优秀同学就读/毕业于软件工程研究所
  - 熊英飞（00级）、古亮（01级）、闫宁（02级）、陈俊宇（13级）

编程语言与开发环境小组

- 指导教师：熊英飞
- 紧密合作：
  - 张路教授（长江学者、杰青）
  - 郝丹副教授（青年长江学者、优青）
- 让计算机学会写程序，将程序员从繁重的体力劳动中解放出来
  - =让程序员下岗？
- 研究路线：从修复缺陷开始，逐步教会计算机写越来越复杂的程序
  - Issue=Bug Report+Feature Request

缺陷定位技术

- 要修复缺陷就必须知道缺陷的位置
- 知道出错的位置有多难？
- 能否自动告诉我们出错的位置？
  - 基于测试覆盖的缺陷定位技术
  - 基于构造正确状态的缺陷定位技术
  - 算法式调试
  - 差异调试

基于测试覆盖的缺陷定位技术

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：
  - 可能出错的语句列表（或方法、文件等），根据出错概率排序
- 如何做到？

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=语句、方法、类、文件等

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

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

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

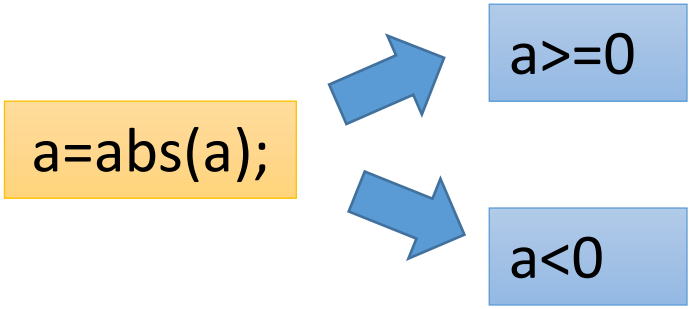
- 常见情况举例
  - 方法级别
  - 语句级别

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。

基于状态覆盖的错误定位

- 通过预定义常见条件把状态空间划分成不同的子集
- 在每个子集上计算测试覆盖

构造正确执行状态

动机

- 如果通过修改程序让测试通过，那么被修改的地方很可能就有错。
- 换句话说，该修改很可能是正确的补丁
- 问题：直接分析出这样的修改比较困难
- 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
  - 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

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);
```

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=4);
```

只有c++是可疑的表达式

算法式调试

算法式调试
Algorithmic Debugging

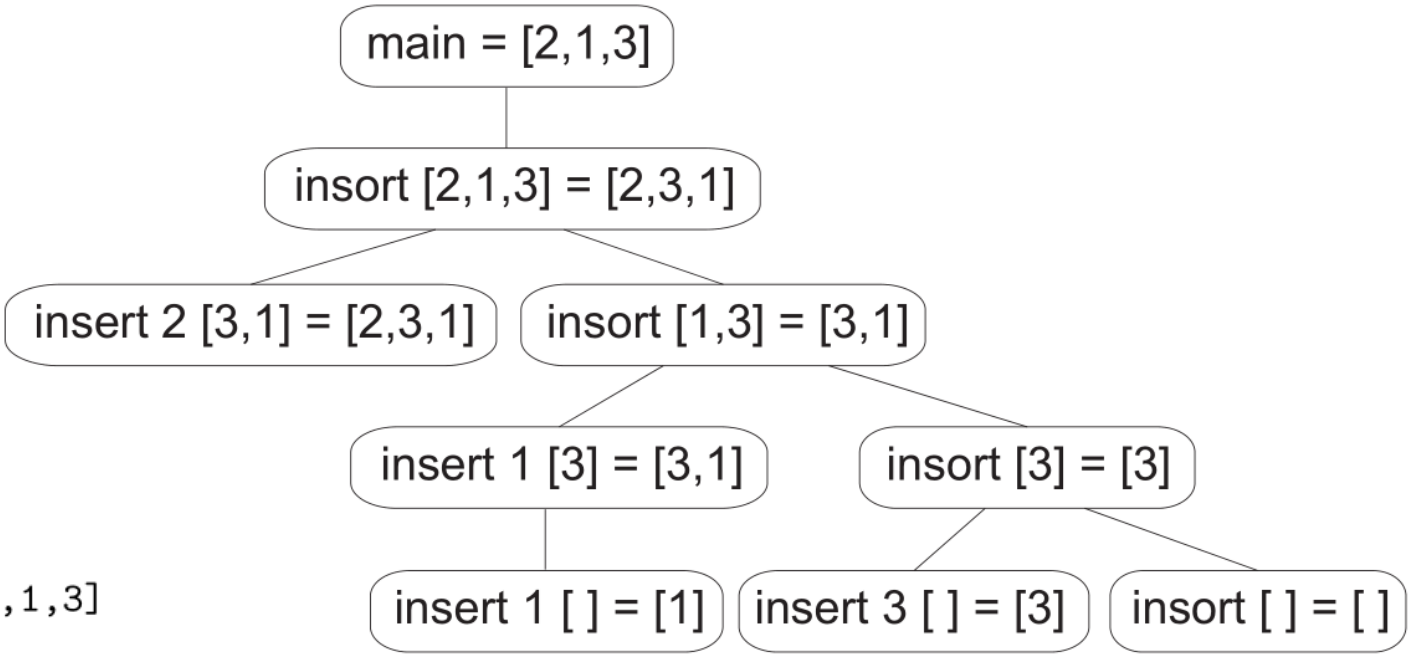
- 之前的所有方法都是试图直接找出错误位置
- 交互式调试：通过询问程序员来定位错误位置

- 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 = 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

算法调试的问题

- 不知道如何应用到命令式语言上
  - 不知道如何简洁有效地表达过程的输入输出
  - 除了过程不知道还能把其他什么元素提取为执行树上的结点
- 缺乏大规模的用户参与的实验

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$test: 2^C \to \{\times, \checkmark, ?\}$，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

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

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

where $c'_{\text{✘}} = c_1 \cup c_2 \cup \cdots \cup c_n$ 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> ✘
```

缺陷定位展望

- 工程领域最重要的问题之一
- 多个领域的研究人员投入研究
  - 软件工程
  - 程序语言
  - 人工智能
- 大量基础问题仍未解决
  - 缺陷定位精度还不够高
  - 缺陷定位技术仍缺乏统一的理论
- 期待感兴趣的同学和我们一起研究