一种基于单条程序执行路径的错误定位方法①

周 艺[1,2], 易秋萍[1,2], 刘 剑[1], 淮晓永[1]

[1](中国科学院软件研究所 总体部, 北京 100190)
[2](中国科学院大学 北京 100049)

摘 要: 当程序在测试中发生错误时, 将形成一条错误的程序执行路径, 程序员将会花费很多精力去检测程序代码和定位最终的程序错误. 提出一种基于单条程序执行路径的错误定位方法, 该方法通过对程序进行反向执行, 计算出多个最弱前置条件及其相对应的疑似错误语句集, 并生成错误定位树, 来辅助程序员进行快速错误定位. 对西门子测试数据集进行的实验表明了该方法具有良好的效果.

关键词: 错误定位; 最弱前置条件; 可满足性理论; 动态分析; 自动化测试

Fault Localization Method Based on Program Execution Trace

ZHOU Yi[1,2], YI Qiu-Ping[1,2], LIU Jian[1], HUAI Xiao-Yong[1]

[1](General Department, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: After a failed test is encountered and the error trace is generated, a significant amount of effort is often required for programmers to manually examine the program code and localize the failure's root cause. In this paper, we propose a fault localization method based on program execution trace. This method computes a set of possible error statements based on a combination of weakest pre-condition computation through program's reverse execution. All possible error statements are organized in a fault localization tree to help developer identify the root cause. Our experiments on SIR datasets demonstrate that our method can not only efficiently compute the possible causes, but also provide sufficient information to help programmers quickly locate the root cause.

Key words: fault localization; weakest pre-condition computation; satisfiability modulo theories; dynamic analysis; automated testing

1 引言

软件错误定位(Fault Localization)技术, 即当程序出现了错误, 程序的预期输出与实际输出不一致时, 研究如何找到引发软件错误的程序段或语句. 如图 1 给出了一个含有逻辑错误的计算三个正整数的最大值的示例程序, 当输入为<3,2,1>时, 预期输出与实际输出不一致, 将导致断言语句 17 出现错误. 传统的错误定位方法, 主要是程序员通过手工设置断点, 重新运行失败的测试用例, 观察并修改程序运行至断点时的各种程序状态等方法, 推断程序中的错误可能存在的位置. 由于测试用例中仅仅定义了该次测试的最终预期输出结果, 并没有给出运行过程中每个状态的预期结果, 因而程序员往往只能根据经验来猜测需要观察和修改哪些状态. 而随着软件规模的逐渐增大, 软件结构日渐复杂, 程序的状态随着程序的运行呈指数级增长, 通过手工定位程序错误将耗费程序员大量的时间和精力. 有研究表明, 测试和调试是软件开发过程中最花费时间的阶段, 通常会花费超过 50%以上的时间[1]. 因此单纯地依靠人工的方法来进行软件的错误定位变得越来越困难, 有必要提供辅助错误定位的自

① 基金项目:中国科学院知识创新工程重要方向性项目(KGCX2-YW-12);国家"核高基"重大科技专项(2010ZX01036-001-002-2);国家自然科学基金青年基金(61305054)

收稿时间:2014-02-24;收到修改稿时间:2014-03-18

© 中国科学院软件研究所 http://www.c-s-a.org.cn

动化方法.

现有的程序自动化错误定位方法主要有: 程序切片技术[2-4]、基于测试用例的特征统计方法[5-9]、基于程序状态的形式化方法[10-13]. 程序切片技术是最早被提出的错误自动化定位技术, 但该方法在切片规模较大时, 算法调度不停询问缩减可疑区域, 有效性降低明显, 代价较大. 基于测试用例的特征统计方法的算法复杂度较低, 易于实现, 可以处理规模中等以上的程序, 但该方法需要大量的测试用例且其在定位过程中不考虑程序的语义信息, 定位出的错误往往不能很好地进行解释, 且该种方法的定位准确性依赖于统计的指标, 如使用该方法的工具 Tarantula[5,14]对于一些程序初始化错误(赋值错误等)难以进行定位. 基于程序状态的形式化方法通常可以很好地解释定位出来的程序错误, 但当程序规模较大及程序结构很复杂时, 需要搜索的的状态数量会变的非常巨大, 因此该方法不适用于大规模程序. 使用一定的搜索规则可以有效的减少搜索状态, 如 Delta Debugging[15], 但同时也会减少定位到的错误或者导致运行异常.

基于以上分析, 本文提出一种准确高效的基于单条程序执行路径的错误定位方法来对软件进行自动化错误定位, 这种方法只需要一条错误的程序执行路径和程序输入, 通过一次程序反向执行, 就可以定位出程序源代码中的错误语句.

```
1   int find_max(int t1,int t2,int t3){
2       int max=0;
3       a=t1,b=t2,c=t3;
4       if(a>=0&&b>=0&&c>=0){
5           if(a<b){                //error should be a>b
6               if(a>c)
7                   max=a;
8               else
9                   max=c;
10          }
11          else{
12              if(b>c)
13                  max=b;
14              else
15                  max=c;
16          }
17          assert(max==MAX);
18      }
19      return max;
20  }
```

图 1 求三个正整数最大值示例程序

2 基础知识

定义 1. 程序错误执行路径 假设程序因为出错而结束执行, 我们可以得到一条程序错误执行路径 $<I,\pi^{1,n}>$, 其中 I 是程序的输入向量, 路径 $\pi^{1,n}$ 是程序语句的执行序列, 表示为 $\pi^{1,n}=<s_e^1,\ldots,s_j^i,\ldots,s_m^n>$, 其中 s_j^i 表示路径中第 i 条执行的语句为 s_j, j 为源代码中的语句的行号. 程序中的每条语句有如下两种类型:

– 赋值语句 $s:v:=e$, 其中 v 是程序变量, e 是程序表达式.
– 分支语句 $s:assume(c)$, 其中 c 是逻辑命题变量, 该变量的形成可能来自于判断语句 $if(c)\ldots else\ldots$ 或者断言语句 $assert(c)$.

如在计算整数最大值示例程序中, $I=<3,2,1>$ 时, 程序的错误执行路径为 $\pi^{1,n}=<s_2^1-s_5^4,s_{11}^5-s_{13}^7,s_{17}^8>$.

定义 2. 最弱前置条件 当存在一条程序执行路径 $\pi^{1,n}=s_e^1,\ldots s_m^n$ 和初始命题变量 ϕ, 那么初始命题变量 ϕ 关于路径 $\pi^{1,n}$ 的最弱前置条件可以表示为 $WP(\pi^{1,n},\phi)$, 具体计算方法如下:

– 赋值语句 $s:v:=e$, $WP(s,\phi)=\phi(e/v)$, 其中 e/v 表示将最弱前置条件中的变量 v 用表达式 e 来进行替换;
– 分支语句 $s:assume(c)$, $WP(s,\phi)=\phi\wedge c$;
– 序列 $s1;s2$, $WP(s1;s2,\phi)=WP(s1,WP(s2,\phi))$.

如在计算整数最大值示例程序中, 初始条件 ϕ 为 $max==MAX$, 那么当反向执行语句 13 后, $WP(s_{13}^7,\phi)$ 将变为 $b==MAX$.

定义 3. 最小不可满足核心 当我们计算得到一条程序错误执行路径的 $WP(\pi^{1,n},\phi)$ 或 $I\wedge WP(\pi^{1,n},\phi)$ 不可满足时, 我们将该不满足的合取式, 表示为 $WP_{unsat}(\phi)$. 该式本质上是一个形如 $p_x^{'}\wedge\ldots\wedge p_y^{'}$ 或 $I\wedge p_x^{'}\wedge\ldots\wedge p_y^{'}$ 的合取式. 该合取式中存在一个最小化的命题变量集合, 移除该集合中任意一个命题变量都将改变合取式的可满足性, 我们将由这个集合中的命题变量形成的合取式称为最小不可满足核心, 表示为 $WP_{unsat}^{min}(\phi)$. 如对于合取式 $a\geq 1\&\&b>0\&\&a<0\&\&c<0$, 该合取式的最小不可满足核心为 $a\geq 1\&\&a<0$.

定义 4. 传递语句及疑似错误语句集 假设 $p_l^{'}$ 是 $WP_{unsat}^{min}(\phi)$ 中的一个命题变量. 我们需要将该命题变量与程序源代码进行关联, 作为最后的结果返回给程序员. 假设命题变量 $p_l^{'}$ 是从命题变量 p_l 传递转换而来, 那么我们认为所有造成该转换的赋值语句都是命题变

© 中国科学院软件研究所 http://www.c-s-a.org.cn

量 $p_l^{'}$ 的传递语句, 我们用 $TP_{p_l^{'}}$ 来表示与 $p_l^{'}$ 相关联的传递语句的集合. 例如, 赋值语句 $s:x=y+1$, 是命题变量 $p_l^{'}:(y+1>0)$ 的传递语句, 因为对于 $p_l^{'}$ 的原始命题 $p_l:(x>0)$ 计算 $WP(s,p_l)$ 可以得到 $p_l^{'}:(y+1>0)$. 这个定义是可传递的, 即 $p^{'}=p(e/v)$ 和 $p^{''}=p^{'}(e_2/v_2)$ 中的赋值语句 $s:v=e$ 和 $s:v_2=e_2$ 都是命题变量 $p^{''}$ 的传递语句. 而对于引入原始命题变量 p_l 的分支语句, 我们同样认为该分支语句属于 $TS_{p_l^{'}}$, 如上面例子中的 $TS_{p_l^{'}}$ 也包含分支语句 $s:(x>0)$. 那么对于一条执行路径 $<I,\pi^{1,n}>$ 以及初始命题变量 ϕ 其对应的 TS_ϕ 即所求的疑似错误语句集, TS_ϕ 定义如下:

$$TS_\phi=\cup\{TS_p\mid p\in WP_{unsat}^{min}(\phi)\}$$

定义 5. 关键语句 我们将程序执行路径中可以影响某个命题变量p的TS中任意语句执行的分支语句 s_i^j 称为 TS_p 的关键语句, 即条件语句 s_i^j 可以对语句 $s_k^l\in TS_p$ 造成影响, 表示为 $s_i^j\to s_k^l$, 其中分支语句 s_i^j 直接决定 s_k^l 是否执行. 需要注意的是, 由于我们需要根据这些关键语句对程序做反向执行, 因此只有出现在程序错误执行路径中的语句, 才能成为关键语句.

最后, 我们用 CC_{TS_p} 来表示这些关键语句的集合, 其定义如下:

$$CC_{TS_p}=\cup_{s_k^l\in TS_p}\{s_i^j\mid s_i^j\to s_k^l\}$$

定义 6. 错误定位树 我们的算法最终会将结果以一棵树的形式返回给用户, 我们将这棵树称为错误定位树, 表示为 $T=<V,E>$, 其中 V 代表树节点的集合, E 代表树边的集合. 对于每个节点 $v=<TS,cc>$, 即节点 v 是疑似错误语句集和用于计算该疑似错误语句集的关键语句的一个二元组. 易知用于计算错误定位树根节点的关键语句即程序执行路径中最后一句出错的断言语句. 对于边 $v_1\to v_2\in E$, 表示 $v_1.TS$ 中的某条语句的关键语句为 $v_2.cc$. 疑似错误语句集的层级即含有该疑似错误语句集的节点到根节点的距离.

3 算法

在这部分我们将结合图 1 所示的示例程序来详细阐述我们的算法. 在该程序的第 5 行中有一个语句错误, 该行的作用是让第一次判断出的较大的数进入下一轮判断, 因此正确的语句应为 a>b, 我们将像语句 5 这样的语句称为最终错误语句, 即真正引发程序错误的语句. 当输入向量为<3,2,1>时, 我们将得到一条程序执行路径<2-5,11-13,17>, 这条路径在程序的 17 行将发生错误. 在第 17 行语句中, *MAX* 表示变量 *max* 在当前输入条件下的正确值.

3.1 计算错误定位树

图 2 给出了我们错误定位算法的整体框架, 该算法接受一条错误的程序执行路径作为输入, 输出指导程序员进行错误定位的错误定位树. 在整个过程中将会使用到计算疑似错误语句集的算法 2 和计算关键语句的算法 3, 这两个子算法将在后面两小节进行介绍. 我们首先将关键语句集 S_{cc} 初始化为 $(s_m^n,0)$, 其中 s_m^n 代表错误路径中的最后一条语句, 0 代表语句的层级, 即由该语句计算得到的疑似错误语句集出现在错误定位树的第 0 层. 在该算法的每一次迭代中, 一条位于层级 i 的关键语句将会从 S_{cc} 集合中移除. 该关键语句将作为参数传递给算法2, 算法2将计算出该语句的最弱前置条件, 同时计算出该条件的最小不满足核心. 计算出的最小不满足核心中的每个命题变量都可以映射到相关的源程序代码上, 这些语句组成的集合 *TS* 就是层级为 i 的疑似错误语句集 *TS*. 然后我们可以使用算法 3 计算得到 *TS* 中任意语句的关键语句集合 *C*, 该集合中的关键语句层级为 i+1, 我们将集合 *C* 加入集合 S_{cc}, 然后进行下一轮迭代计算, 直到集合 S_{cc} 为空时, 计算结束. 注意, 虽然算法的主体是一个循环, 但程序路径的反向执行只需要做一次即可.

算法1 计算错误定位树
输入: 错误执行路径 $<I,\pi^{1,n}\leftarrow\{s_e^1,\ldots,s_j^i,\ldots s_m^n\}>$
输出: 错误定位树
1 初始化关键语句集 $S_{cc}\leftarrow\{s_m^n,0\}$;
2 while $S_{cc}\neq\varnothing$ do
3 从关键语句集 S_{cc} 中选择关键语句 (s,i);
4 使用算法2计算与关键语句 (s,i) 对应的疑似错误语句集 TS;
5 错误定位树=错误定位树 $\cup\{(TS,i)\}$;
6 使用算法3计算疑似错误语句集TS的关键语句集 C;
7 更新关键语句集 $S_{cc}=S_{cc}\cup\{(c,i+1)\mid c\in C\}$;
8 end while

图 2 错误定位算法整体框架

3.2 计算疑似错误语句集

在实际程序中并不是所有程序执行路径中的语句都会导致程序断言语句的失败, 因此在这一小节中, 我们将给出算法用来计算导致程序断言语句失败的最小相关语句的集合, 即疑似错误语句集.

假设程序错误执行路径中最后的断言语句 s_m^n 表

© 中国科学院软件研究所 http://www.c-s-a.org.cn

示的命题变量为 ϕ，如在图 1 的示例程序中该命题变量为 $max == MAX$，则程序出错的原因就是程序路径执行到语句 s_m^n 时命题变量 ϕ 的值为假. 我们以 ϕ 作为初始命题变量，从 s^{n-1} 开始计算最弱前置条件，我们可以得到:

$$WP(\pi^{i,n-1},\phi)=\phi'\wedge(p_v'\wedge\ldots\wedge p_w')$$

其中 $1\le i\le n-1$，ϕ' 从初始命题变量 ϕ 传递转换而来，每个命题变量 p_v' 从命题变量 p_v 传递转换而来，p_v 来自于路径中的分支语句 $s^v: assume(c_v)(1\le i\le n-1)$. 假设我们在程序执行的每一步都检测最弱前置条件的可满足性，那么会有两种可能:

- 在第 i 步 $(1\le i\le n-1)$，$WP(\pi^{i,n-1},\phi)$ 不可满足
- 在整条程序执行路径中计算出的 $WP(\pi^{i,n-1},\phi)$ 都是可满足的，但 $I\wedge WP(\pi^{i,n-1},\phi)$ 不可满足

第一种情况说明执行路径本身将导致最后的命题变量 ϕ 发生错误；第二种情况说明输入与路径中的某些条件发生了冲突，引起了最后错误，如在示例程序中，必须加上输入条件 $t1==3\&\&t2==2\&\&t3==1$ 才能使合取式不满足.

从这两种情况，我们都可以获得 $WP_{unsat}^{min}(\phi)$，同时根据 $WP_{unsat}^{min}(\phi)$ 计算出 TS_ϕ，TS_ϕ 即我们要求的疑似错误语句集.

图 3 给出计算 TS_ϕ 的算法. 第三个参数 s_x^y 代表由路径中最后一句语句 s_m^n 或者其他的关键语句. 我们将在下一小节中说明如何计算得到关键语句. 我们从初始命题变量 ϕ 开始反向执行程序，并计算最弱前置条件. 在算法的循环部分，在第 i 次迭代计算时，$WP_\phi=WP(\pi^{i,n-1},\phi)$，在循环中既可以是通过赋值语句对 WP_ϕ 中的变量进行替换，也可以是通过分支语句向 WP_ϕ 增加一个新的命题变量，每一次 WP_ϕ 发生变化，我们都将利用 SMT 求解器[16]检测 WP_ϕ 的可满足性. 如果算法执行完循环部分(算法 2-14 行) WP_ϕ 仍是可满足的，那么当加入输入向量 I 时，将导致 WP_ϕ 不可满足. 在算法的最后部分，将 $WP_{unsat}^{min}(\phi)$ 中的每一个命题变量的 TS_p 加入到疑似错误语句集中.

$max == MAX$ 作为初始命题变量 ϕ，并开始进行程序反向执行，在反向执行的过程中，程序的状态将用一系列的逻辑命题的合取式来表示. 在示例中，我们的算法反向执行到语句 3 的时候将暂停，这时候形成的不可满足的合取式为:

$$b==MAX\&\&b>c\&\&a\ge b\&\&a\ge 0\&\&b\ge 0\&\&c\ge 0$$

其中, a 的值为 3, b 的值是 2, c 的值为 1, MAX 的值为 3. 因此在我们的示例中，最小不可满足核心 $WP_{unsat}^{min}(\phi)$ 为：$b==MAX$，该命题的相关传递语句为语句 13 和语句 3，因此计算得到的疑似错误语句集 TS_ϕ 为<17,13,3>.

算法2 计算疑似错误语句集
输入: 错误执行路径 $<I,\pi^{1,n}\leftarrow\{s_e^1,\ldots,s_j^i,\ldots s_m^n\}>$ 及关键语句 s_x^y
输出: 疑似错误语句集

```
1   根据关键语句 s_x^y 初始化命题变量 φ;
2   for i=y-1;i>=1;i-- do
3       if s^i 是一条表示命题变量p的分支语句 then
4           WP_φ.add(p);
5           TS_p.add(s^i);
6       else if s^i 是关于命题 p∈WP_φ 的传递语句 then
7           TS_p.add(s^i);
8           根据语句 s^i 的语义更新命题变量p
9       end if
10      if WP_φ 发生更新并且 WP_φ 不可满足时 then
11          求出 WP_φ 的最小不可满足核心 WP_unsat^min(φ);
12          break;
13      end if
14  end for
15  if WP_unsat^min(φ) 可满足 then
16      WP_φ.add(I);
17      求出 WP_φ 的最小不可满足核心 WP_unsat^min(φ);
18  end if
19  for 任意 p∈WP_unsat^min(φ) do
20          TS_φ.add(TS_p);
21  end for
```

图 3 计算疑似错误语句集

3.3 计算关键语句

算法 2 计算出关于命题变量 ϕ 的疑似错误语句集，有两种方法可以修正发现的错误：一种方法是直接修改 TS_ϕ 中的语句，即修改赋值语句，改变 WP_ϕ 中某些变量的值，有可能使最后的命题变量 ϕ 得到满足；另一种方法是间接影响 TS_ϕ 中的语句，具体的做法是改变这些赋值语句的可达性，当这些赋值语句不可达时，就有可能避开最后的命题变量 ϕ 或者改变它的值. 因此当程序的最终错误语句没有出现在 TS_ϕ 中，我们可以根据后一种方法计算出对应的关键语句从而计算出新的疑似错误语句集.

在这一小节中，我们将说明如何根据之前计算出的疑似错误语句集计算出关键语句. 考虑图 1 中的计算三个正整数的最大值的示例程序，当第 17 行的程序断言发生错误时，我们从 17 行开始对程序做反向执

© 中国科学院软件研究所 http://www.c-s-a.org.cn

行, 并利用上一小节中的算法可得 $TS_{\phi} =<17,13,3>$, 但最终错误语句 5 并没有出现在这个疑似错误语句集中, 因此我们需要根据该疑似错误语句集计算出其他的疑似错误语句集. 现在我们分别考虑语句 17 和语句 13. 对于语句 13 来说, 语句 12 直接控制该语句是否被执行, 那么语句 12 将被认为是用来计算下一层级疑似错误语句集的关键语句. 同理对于语句 17, 语句 4 即为关键语句. 在这里我们认为初始关键语句 17 是语句 12 和语句 4 的父亲语句, 这种父子关系有利于程序员对程序的分析, 同时这也是将这些疑似错误语句集组织成最终的错误定位树的依据. 然后, 我们将语句 12 和语句 4 的命题变量取反, 取反后的命题变量为 $b \le c$ 和 $!(a \ge 0 \&\& b \ge 0 \&\& c \ge 0)$, 将这两个条件作为算法 2 的第 3 个参数, 我们就可以计算出层级为 1 的疑似错误语句集 $TS_{cc_1} =<12,3>$ 和 $TS_{cc_1} =<4,3>$. 这两个疑似错误语句集可以解释 $b \le c$ 和 $!(a \ge 0 \&\& b \ge 0 \&\& c \ge 0)$ 这两种程序执行情况没有发生的原因.

```
算法3 计算关键语句集
输入: 疑似错误语句集TS
输出: 关键语句集 TS_cc
1   for 任意属于疑似错误集TS的语句 s_k^l do
2       if 条件语句 s_i^j 距离语句 s_k^l 最近且具有控制关系;
3           CC_TS.add(s_i^j);
4       end if
5   end for
```

图 4 计算关键语句集 TS_{cc}

图 4 给出了计算关键语句集的伪代码, 对于任何传递语句 $s_k^l \in TS$, 我们选择对该语句具有控制作用且距离最近的分支语句作为关键语句. 选择所有有控制关系的分支语句作为关键语句, 将会导致分析成本大大增加, 并产生大量的疑似错误语句集, 不利于指导程序员进行分析, 同时我们随后的实验表明, 这种简化算法定位出了所有的错误. 因此在上文中我们只考虑了语句 12 和语句 4.

表 1 求三个正整数最大值示例程序的错误定位结果

层数	关键语句	关键语句的父亲语句	疑似错误语句集	是否包括最终错误语句
0	17	-	{17,13,3}	否
1	12	17	{12,3}	否
1	4	17	{4,3}	否
2	5	12	{5,3}	是

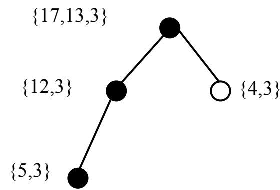

图 5 表 1 中疑似错误集所组成的错误定位树

考虑 $TS_{cc_1} =<4,3>$, 我们可以得到新的关键语句: 语句 5. 当以语句 5 作为算法起点时, 我们可以得到最小不可满足核心 $a<b$, 因此可以得到疑似错误语句集 <5,3>. 其中语句 5 正是我们所要求的最终错误语句.

通过对整条程序错误执行路径的迭代分析, 我们将获得的这些层级不同的疑似错误语句集组成错误定位树返回给程序员, 用以指导他们发现最终错误. 如表 1 列出了示例程序中这些层级不同的疑似错误语句集, 图 5 展示了由这些疑似错误语句集所组成的错误定位树, 黑色节点表示这些节点中的语句的执行导致了最终错误的发生.

4 实现

为了对我们的方法进行评估, 我们实现了错误定位工具—BugLocator. 图 6 为该工具的实现框架. 它由前端、计算分析、错误报告三部分组成. 前端部分首先使用 LLVM-GCC 将被测程序编译成字节码, 然后将编译后的程序交给符号执行虚拟机进行符号执行. 这个过程中虚拟机将记录下程序的错误执行路径. 在计算分析部分, 我们开发的求解器模块将根据错误执行路径计算最弱前置条件, 并根据计算结果得到疑似错误语句集和关键语句. 在错误报告部分, 我们将获得计算所得到的错误定位树, 程序员可以根据这棵树的指导进行调试, 发现最终的错误.

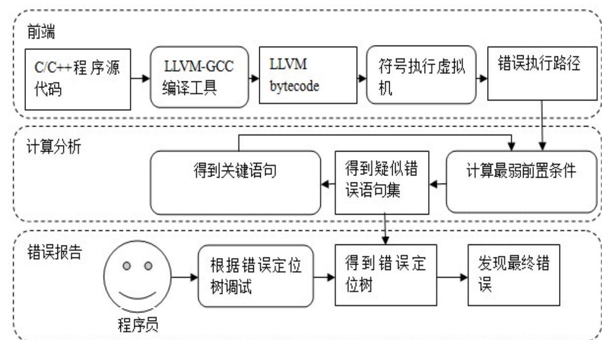

图 6 BugLocator 工具实现框架

5 实验

为了测试我们方法的有效性, 我们对程序错误定

© 中国科学院软件研究所 http://www.c-s-a.org.cn

位研究[6,11]中常用的西门子测试集[17]进行了实验一和实验二. 其中实验一用于检测本文的方法是否能够有效定位到程序错误; 实验二是一组对比实验, 将本文的方法和国际上现有工作在准确性、时空消耗等方面进行了比较. 该测试集包括 7 个程序系列: TCAS 程序是一个飞行器冲突检测程序, schedule 和 schedule2 程序是两个优先级调度程序, totinfo 是一个对给定数据集统计相关信息的程序, printtokens 和 printtokens2 是两个词法分析程序, replace 是一个模式匹配替换程序. 这些被测程序中包含了指针、函数调用、递归、循环、动态内存分配等复杂程序结构, 具有较好的通用性. 实验环境为: IntelCore-i7-2600CPU3.40GHz, 内存 4G, ubuntu10.04 系统.

在实验一中我们使用 BugLocator 对测试集中的六个程序系列进行错误定位实验. 表 2 给出了实验一的结果. 第三列给出了每个程序的错误类型, 其中 op 代表操作符误用, const 代表常量错误, code 代表程序代码逻辑错误, assign 代表赋值错误, addcode 代表代码冗余, 表 3 中该列的 init 代表初始化错误; 第五列, 疑似错误语句集总数表示在发现最终错误语句时, 所计算出的疑似错误语句集的总个数; 第六列有两个子列, 第一个子列表示最终错误语句所在的疑似错误语句集在树中的层数, 第二个子列表示该疑似错误语句集的大小, 即对应程序源代码的行数.

表 2 西门子测试集实验一结果

程序	行数	错误类型	执行路径长度	错误集总数	错误所在错误集		耗时
					层级	大小	
schedule2	564	code	8428	24	8	4	6.65s
		op	2945	6	3	2	0.08s
schedule	374	const	12390	11	4	4	1.78s
		op	13689	35	5	4	7.25m
totinfo	565	assign	12770	4	3	1	3m
printtokens2	523	assign	1719	12	7	6	0.06s
		addcode	10771	20	1	3	0.27s
printtokens	726	assign	3069	27	3	6	17.6s
replace	512	assign	3488	31	3	6	8.27s
		op	16737	38	4	7	15m

实验一结果表明我们的工具定位出了所有程序中各种类型不同的错误. 同时程序的错误执行路径较长, 但我们的工具计算出的错误定位树层数较少, 具有良好的错误定位效果.

在实验二中, 我们分别使用 BugLocator 和 BugAssist[18]工具对 TCAS 系列程序的 20 个不同版本进行了错误定位对比实验. BugAssist 是一款对整个程序代码进行形式化编码, 并根据编码的可满足性进行错误定位的工具. 表 3 给出了实验二的结果. 第四列为工具能正常定位出错误的实验用例数, 其中子列 L 代表工具 BugLocator, 子列 A 代表 BugAssist; 第五列为工具所标记出的疑似错误语句在程序源代码中的所占比例.

实验二的结果表明我们的工具在所有的测试用例中都准确定位出了错误代码, 而 BugAssist 在版本 3 和版本 12 中只在部分测试用例中定位出了错误代码, 同时我们的工具所定位出疑似错误语句所占源码的比例更小, 定位时间更短. 因此使用我们的工具, 程序员可以检查更少的程序代码, 在更短的时间内发现程序错误. 该实验表明我们的工具比 BugAssist 更为准确高效.

6 结语

在本文中, 我们提出了一种基于单条程序执行路径的错误定位方法, 这种方法仅仅需要一条错误的程序执行路径就可以计算出多个疑似错误语句集, 并生成树形图形化的错误定位树, 可辅助开发人员高效地理解定位错误, 本文方法相比与传统方法只需单一的错误测试用例即可精准地定位出程序中的错误.

表 3 西门子测试集实验二结果

版本	测试用例数	程序错误数	成功检测用例数		疑似错误语句比例(%)		耗时(s)		错误类型
			L	A	L	A	L	A	
1	131	1	131	131	4.6	8.6	0.058	0.016	op
2	69	1	69	69	6.9	4.6	0.055	0.068	const
3	23	1	23	13	6.7	9.8	0.057	0.096	op
4	20	1	20	20	5.1	9.2	0.055	0.104	op
5	10	1	10	10	7	8.6	0.054	0.120	assign
6	12	1	12	12	7.7	8.6	0.053	0.108	op
7	36	1	36	36	6.4	9.2	0.061	0.072	const
8	1	1	1	1	6.8	8.6	0.062	0.112	const
9	7	1	7	7	6.5	5.2	0.057	0.092	op
10	14	2	14	14	7.2	9.2	0.060	0.136	op

© 中国科学院软件研究所 http://www.c-s-a.org.cn

11	14	2	14	14	5.8	6.3	0.055	0.080	op
12	70	1	70	48	6.7	9.2	0.054	0.164	op
13	4	1	4	4	9.1	9.2	0.055	0.080	const
14	50	1	50	50	1.2	8.1	0.031	0.028	const
15	10	1	10	10	6.0	7.5	0.051	0.104	const
16	70	1	70	70	6.5	9.2	0.054	0.104	init
17	35	1	35	35	6.0	9.2	0.060	0.096	init
18	29	1	29	29	5.7	6.9	0.060	0.124	init
19	19	1	19	19	6.0	9.2	0.061	0.112	init
20	18	1	18	18	7.2	9.2	0.051	0.120	op

参考文献

1 Beizer B. Software Testing Techniques. London, UK: International Thomson Computer press, 1990.

2 Weiser M. Program Slicing. ICSE '81 Proceedings of the 5th international conference on Software engineering. Piscataway, NJ, USA: IEEE Press, 1981.

3 Korel B, Laski J. Dynamic program slicing. Information Processing Letters, 1988, 29(3): 155–163.

4 DeMillo RA, Pan H, Spafford EH. Critical slicing for software fault localization. ISSTA’96 Proc. of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA. New York, NY, USA. ACM Press. 1996.

5 Jones J, Harrold MJ, Stasko J. Visualization of test information to assist fault localizatgion. ICSE’02 Proc. of the 24th International Conference on Software Engineering. NY, USA. ACM Press. 2002.

6 Renieris M, Reiss SP. Fault localization with nearest neighbor queries. ASE’03 Proc. of the 18th IEEE/ACM International Conference on Automated Software Engineering. New York, NY, USA. ACM Press. 2003.

7 Groce A, Kroening D, Lerda F. Understanding counterexamples with explain. Proc. of 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004.

8 Zhang ZY, Chan WK, Tse TH. Fault localization based only on failed runs. Computer, 2012,45(6): 64–71.

9 Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. PLDI’05 Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA. ACM Press. 2005.

10 Balakrishnan G, Ganai M. PED: Proof-guided error diagnosis by triangulation of program error causes. Software Engineering and Formal Methods, 2008. SEFM’08. Sixth IEEE International Conference. Los Alamitos, CA, USA. IEEE Computer Society Press. 2008.

11 Griesmayer A, Staber S, Bloem R. Automated fault localizatin for C program. Electronic Notes in Theoretical Computer Science, 2007,174(4): 95–111.

12 Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. ICSE’06 Proc. of the 28th International Conference on Software Engineering. New York, NY, USA. ACM Press. 2006.

13 Groce A, Chaki S, Kroening D, Strichman O. Error explanation with distance metrics. International Journal on Software Tools for Technology Transfer, 2006, 8(3): 229–247.

14 Jones J, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. ASE’05 Proc. of the 20th IEEE/ACM int. Conf. on Automated software engineering. NY, USA. ACM Press. 2005.

15 Cleve H, Zeller A. Locating causes of program failures. ICSE’05 Proc. of the 27th International Conference on oftware Engineering. NY, USA. ACM Press. 2005.

16 http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories

17 Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005, 10(4): 405–435.

18 http://bugassist.mpi-sws.org/

© 中国科学院软件研究所 http://www.c-s-a.org.cn