一种基于最弱前置条件的软件错误定位算法

李 雅[1,2],黄少滨[2],李艳梅[2],迟荣华[2],郎大鹏[2]
(1. 黑龙江工程学院计算机科学与技术学院,黑龙江哈尔滨 150050;
2. 哈尔滨工程大学计算机科学与技术学院,黑龙江哈尔滨 150001)

摘 要: 错误定位是软件调试中非常耗时费力的活动之一,自动错误定位技术可以有效地提高调试效率,降低调试成本. 基于最弱前置条件的错误定位技术,首先计算出程序需要满足的最弱前置条件,并为其构造错误分析图;然后在错误分析图上依照失败测试用例进行初始化标记;最后限定分析图的输入和输出,自顶向下再次对其进行标记,找到冲突的结点,从而进行错误定位. 实验结果表明,相对于其它方法,文中提出的方法能有效地提高程序错误定位的效率,使得调试人员只需检查更少的语句即可找到出错的位置.

关键词: 调试; 错误定位; 最弱前置条件; 反例理解
中图分类号: TP316 **文献标识码**: A **文章编号**: 0372-2112 (2019)01-0025-08
电子学报 URL: http://www. ejournal. org. cn **DOI**: 10. 3969/j. issn. 0372-2112. 2019. 01. 004

Technique of Software Fault Localization Based on Weakest Pre-condition

LI Ya[1,2], HUANG Shao-bin[2], LI Yan-mei[2], CHI Rong-hua[2], LANG Da-peng[2]
(1. *College of Computer Science and Technology, Heilongjiang Institute of Technology, Harbin, Heilongjiang* 150050, *China*;
2. *College of Computer Science and Technology, Harbin Engineering University, Harbin, Heilongjiang* 150001, *China*)

Abstract: Fault localization is one of the most time-consuming activities in software debugging, and automatic fault localization technique can effectively improve the efficiency and reduce the cost. In this paper, a fault localization technique based on weakest pre-condition is proposed. Firstly, we compute the weakest pre-condition, and construct the fault analysis graph. Secondly, label the nodes according to the failing test case. Finally, label the graph top-down again while the output is limited to true, and the input cannot be changed. The conflicting nodes were found as the possible errors. The experimental results suggest that the proposed method in this paper has better performance compared to other methods. So that the debugger can only check fewer statements to find out the location of the error.

Key words: debugging; fault localization; weakest pre-condition; counterexample explanation

1 引言

软件测试阶段最耗时、代价最昂贵的任务之一就是“调试”[1]. 软件调试是定位并排除软件错误的常用手段,错误定位是指在已知程序失效的情况下定位错误的过程. 自动化、半自动化的软件错误定位技术目前仍是一个研究热点[2],同时随着软件或者系统的复杂度越来越高,使得它也是一个研究难点. 利用模型检测的方法是一种自动的、基于模型的、性质验证处理方法. 当验证的模型不满足给定性质时给出反例是模型检测方法的一个显著特点[3]. 然而,反例仅反映了模型存在缺陷,并没有明确指出模型错误的原因,需要调试者根据自己的经验定位错误.

近年来,软件错误定位技术已经引起研究人员的广泛关注. 错误定位的研究按照其方法大致分为:基于程序切片的错误定位方法[4~7],基于程序谱的错误定位技术[8~10],基于模型检测的方法以及一些其他的方法[11,12]. 比较经典的利用模型检测进行错误定位的方法包括:利用模型检测器的定位方法,基于路径的语法级最弱前置条件的辅助算法,利用克雷格插值的证明

收稿日期:2016-09-09;修回日期:2018-06-03;责任编辑:孙瑶
基金项目:国家自然科学基金(No. 61772177, No. 61806075);黑龙江省自然科学基金(No. F2018029);黑龙江工程学院博士科研启动基金(No. 2018BJ03, No. 2017BJ13)

方法,以及基于可满足性的方法等[13~17]. 本文提出了一种利用最弱前置条件定位程序错误的方法. 该方法从目标断言出发,计算程序的最弱前置条件,构造程序的错误分析图,以触发程序错误的初始条件作为程序的入口点,在分析图中进行两次标记操作,找到两次标记的冲突部分,排除因为最弱前置条件计算过程中的结构产生的结点,找到程序错误的可能位置.

2 相关知识

本文准备研究的程序使用的语言是大多数命令型编程语言的核心语言. 忽略了平凡的语法变化,核心语言有三类语法论域:整数表达式、布尔表达式和命令也就是程序[18]. 可知整数表达式 E 和布尔表达式 B 用 Backus Naur 范式表示为

$$E::=n|x|(-E)|(E+E)|(E-E)|(E*E) \quad (1)$$

$$B::=\text{true}|\text{false}|(!\ B)|(B\&B)|(B\parallel B)|(E<E) \quad (2)$$

其中 n 是整数,x 是任意变量. ! 表示否定,& 表示合取,‖ 表示析取. 因此,程序的语法论域可定义为

$$S::=x=E|S;S|\text{if } B\ \{S\}\ \text{else}\ \{S\}|\text{while } B\ \{S\}$$

有了程序的描述和对程序需要满足的断言 assert,利用霍尔三元组表示程序的前置条件和后置条件. $(|\varphi|)P(|\psi|)$ 的形式称为霍尔三元组,公式 φ 称为 P 的前置条件,ψ 称为 P 的后置条件,也就是断言 assert. 程序的最弱前置条件(Weakest Precondition,WP)的定义如下.

定义 1 假设 P 表示一段程序,ψ 表示 P 的断言,则程序 P 对应于断言 ψ 的最弱前置条件表示为 $WP(P,\psi)$,其中程序基本语句的最弱前置条件的推演规则为

$$\overline{(|\psi[E/x]|)x=E(|\psi|)} \quad (3)$$

$$\frac{(|\varphi|)s_1(|\eta|)\ (|\eta|)s_2(|\psi|)}{(|\varphi|)s_1;s_2(|\psi|)} \quad (4)$$

$$\frac{(|\varphi_1|)s_1(|\psi|)\ (|\varphi_2|)s_2(|\psi|)}{(|(B\rightarrow\varphi_1)\wedge(\neg B\rightarrow\varphi_2)|)\text{if } B\{s_1\}\text{else}\{s_2\}(|\psi|)} \quad (5)$$

式(3)是赋值语句的推演规则,$\psi[E/x]$ 表示将 ψ 中的所有自由出现的 x 都用 E 替换而得到的公式. 式(4)是 s_1 和 s_2 复合语句的推演规则,其中 η 称为中间条件,它是 s_1 语句的后置条件,也是 s_2 的前置条件. 式(5)是 if 语句的推演规则. 显然,程序的最弱前置条件公式用 Backus Naur 范式可以表示为式(6).

$$\varphi::=p|(\neg\varphi)|(\varphi\wedge\varphi)|(\varphi\vee\varphi)|(\varphi\rightarrow\varphi) \quad (6)$$

接下来,本文将利用一种标记算法来定位程序的错误,使用共享语法分析树共同子式的有向无环图(Directed Acyclic Graph)表示程序的最弱前置条件,称之为错误分析图.

定义 2 式 φ 的错误分析图 $D(\varphi)=(S,\rightarrow,Q,M)$ 是一个四元组,S 是图中所有的顶点,$\rightarrow\subseteq S\times S$ 表示边,具有唯一初始结点 $Q\in S$,其所有终止结点为 $M\subseteq S$,并且标记为整数表达式,或者布尔值 true 和 false. 所有非终结结点用逻辑符号¬ ,∧,∨标记.

由式(6)可知,对于 φ 的错误分析图 $D(\varphi)$ 可以递归的定义为

$$\begin{aligned}&D(p)=p;D(\neg\varphi)=\neg D(\varphi);\\&D(\varphi_1\wedge\varphi_2)=D(\varphi_1)\wedge D(\varphi_2);\\&D(\varphi_1\vee\varphi_2)=D(\varphi_1)\vee D(\varphi_2);\\&D(\varphi_1\rightarrow\varphi_2)=\neg D(\varphi_1)\vee D(\varphi_2)\end{aligned} \quad (7)$$

利用定义 1 及它的推演规则,很容易获得程序 P 的最弱前置条件 $WP(P,\psi)$ 并构造出语法分析图. 假设导致程序断言违反的输入为 I,可知 $I\wedge WP(P,\psi)$ 是不可满足的. 按照输入 I,对语法分析图中的各个终结点和非终结点进行标记,显然 $D(WP(P,\psi))$ 的根结点必定标记为 FALSE.

3 基于最弱前置条件的软件错误定位

示例程序 1 包含了程序语言的核心语句赋值语句和 if 语句,并且返回数组 Array 的位置 i 的值. 很明显 i 的值必须是 $i\geqslant 0$ 并且 $i<3$,即示例程序 1 必须要满足断言 assert($i\geqslant 0$ && $i<3$). 但是示例程序 1 包含一个错误,如果 index 等于 1 时,if 语句的 else 分支使得 index =3,将不满足断言 assert($i\geqslant 0$ && $i<3$).

```
int   Array[3];
………
if(index!=1)   then
    index=2;
else
    index=index+2;
end   if
………
i=index;
return   Array[i]; //assert(i≥0&&i<3);
```

图1 示例程序

3.1 计算最弱前置条件

给定一段程序 P 和断言 ψ,其中程序 P 由语句 s_1, s_2,…,s_n 组成,令 $WP(P,\psi)$ 表示程序 P 在断言 ψ 下的最弱前置条件. $WP(s,\psi)$ 表示赋值语句 $s:x=E$ 关于 ψ 的最弱前置条件,由规则 3 可知 $WP(s,\psi)=\psi[E/x]$. 若 s 为 if 语句,由规则 5 可知 $WP(s,\psi)=(B\rightarrow\varphi_1)\wedge(\neg B\rightarrow\varphi_2)$. 由规则 4 可知:$WP(s_1;s_2,\psi)=WP(s_1,WP(s_2,\psi))$. 因此程序 P 关于断言 ψ 的最弱前置条件为:$WP(P,\psi)=WP(s_1,WP(s_2,WP(s_3,\cdots,WP(s_{n-1},WP$

$(s_n,\psi)))))$.

示例程序 1 关于断言 $i<3$ 的最弱前置条件为 $WP(P,i<3)=[(\text{index}!=1)\rightarrow(2<3)]\wedge[\neg(\text{index}!=1)\rightarrow(\text{index}+2<3)]$. Wang[14] 和黄洪涛[15] 计算的是基于反例的关于断言的最弱前置条件,而本文计算的是程序关于断言的最弱前置条件. 它们之间最大的不同在于 if 语句,例如示例程序 1 中的 if 语句 s,令 s_1 表示 if(index! =1)、s_2 表示 index =2、s_3 表示 index = index +2,后置条件是 φ,反例包含语句 s_1 和 s_2,则 $WP(s,\varphi)=WP(s_1,WP(s_2,\varphi))=WP(s_2,\varphi)\wedge(\text{index}!=1)$. 本文中根据 if 语句的推演规则 $WP(s,\varphi)=((\text{index}!=1)\rightarrow WP(s_2,\varphi))\wedge(\neg(\text{index}!=1)\rightarrow WP(s_3,\varphi))$. 由于本文研究的程序指的是核心的程序语言,仅包含 if 语句和赋值语句,因此最弱前置条件计算过程是机械的,可以使用程序完全自动的完成. 得到程序 P 起始位置的最弱前置条件 $WP(P,\psi)$之后,如果 P 的初始状态满足它,则可以保证程序运行将终止于一个满足 ψ 的状态,如果不满足,则程序的运行将终止于一个不满足 ψ 的状态. 因此对于模型检测给出的反例中的初始状态将不满足 $WP(P,\psi)$.

3.2 构造错误分析图

根据定义 2 和错误分析图 $D(\varphi)$的递归定义,用方框表示终结结点,圆圈表示非终结结点,可画出式(7)的错误分析图,如图 2 所示.

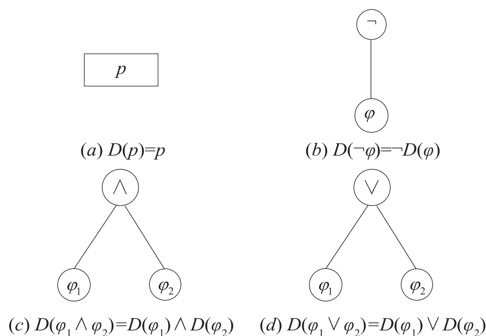


图2 $D(\varphi)$的递归表示

实际的程序包含了赋值语句,以及 if 语句,因此程序的最弱前置条件包含了整数,算术运算符,以及布尔表达式和逻辑运算符. 为了简化起见,本文将整数表达式 1 看作终结结点,即用方框表示如图 3(a)所示. 对于布尔表达式 2 中的 true 和 false 也视为终结结点用方框表示,如图 3(b)和(c)所示,其他为非终结结点用圆圈表示如图 3(d)所示.

为了方便的区分 if 结构和程序语句里的与,或和非,if 结构相关的非终结节点用 ∧、∨、¬ 表示,程序语句中用 &、‖,! 表示. 特别的是,赋值语句和复合语句不改变程序最弱前置条件的整体结构,仅新增深层终

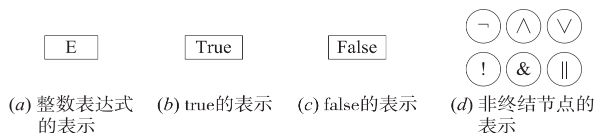


图3 节点的表示

结结点或者非终结结点. 而计算 if 语句的最弱前置条件时,将引入 4 个标记逻辑运算符的非终止结点,甚至改变最弱前置条件对应的错误分析图的上层图形结构,其中 if 语句的最弱前置条件为 $WP(s,\psi)=(B\rightarrow\varphi_1)\wedge(\neg B\rightarrow\varphi_2)=(\neg B\vee\varphi_1)\wedge(B\vee\varphi_2)$,则构造出的 DAG 具有图 4 的形式.

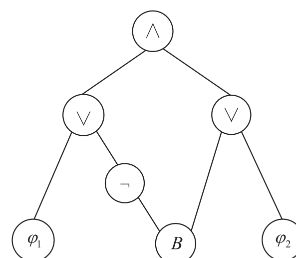


图4 if语句的错误分析图

如示例程序 1 所示,它关于断言 $i<3$ 的最弱前置条件为 $WP(P,i<3)=[(\text{index}!=1)\rightarrow(2<3)]\wedge[\neg(\text{index}!=1)\rightarrow(\text{index}+2<3)]$,则它的错误分析图为 $D=(S,\rightarrow,Q,M)$,其中顶点 $S=\{\neg,\wedge,\vee,2<3,\text{index}!=1,\text{index}+2<3\}$,起始点标记为 ∧,终止结点为 $M=\{2<3,\text{index}!=1,\text{index}+2<3\}$,边→如图 5 所示.

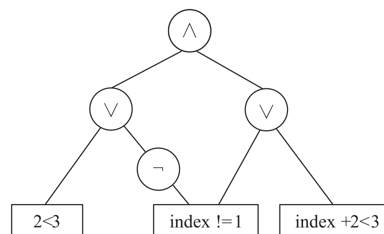


图5 示例程序1的错误分析图

3.2.1 自顶向下的错误标记

绘出程序的错误分析图后,本文在此图上执行一个自顶向下的标记算法,以期找到程序错误的位置,并试图修正程序的错误. 算法 1 给出了自顶向下的错误标记算法的整体框架,该算法接受一组失败的输入和程序的错误分析图,输出程序存在错误的可能位置.

算法 1 自顶向下的错误标记算法

//输入:失败的测试用例,I;程序 P 的最弱前置条件 $WP(P,\varphi)$的语法分析图,$D(P)=(S,\rightarrow,Q,M)$;

//输出:疑似错误语句集,Susp;

```
1. 根据测试用例 I,用真假值自底向上的标记 D(P)的所有节点;
2. 标记 if 语句结构的非终结节点 IFNode;
3. 将图 D(P)的起始节点标记为 TRUE;
4. while 存在需要二次标记的节点 do
5.      if (节点 Node 是∨,并且二次被标记为 TRUE) then
6.           将节点 Node 的子节点 Cnode 标记为 TRUE,并查看当前标
             记是否会引起 Cnode 的其他父节点二次标记冲突;
7.           if (存在冲突)then
8.                修改 Cnode 的标记为第一次标记的值;
9.           end if
10.     else
11.          依照相应的标记规则对 Node 的子节点 Cnode 进行
             标记;
12.     end if
13.     if (Cnode. firstflag≠Cnode. Secondflag) then
14.          Susp = Susp∪Cnode;
15.     end if
16. end while
17. Susp = Susp/Cnode;
18. return Susp;
```

在整个算法中,最重要的是对程序错误分析图的各个顶点进行真假值的标记,首先需要依据反例自底向上的标记,和自顶向下的再次标记. 对于非终结结点的标记需要依据一定的约束规则,如图 6 ~8 所示. 其中小圆圈表示任意结点(¬ ,∧,∨或原子).

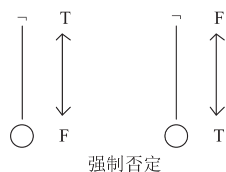


图6 否定约束规则

图 6 中的否定约束规则表示在¬ 结点上真的约束强加它的对偶值到它的子结点,反之亦然. 图 7 表示合取约束规则,其中图 7(a)表示将一个∧结点的 T 约束到它的两个子结点;对偶地,如果一个∧结点的所有子结点都标记 T,则该结点被约束为 T. 图 7(b)表示如果一个∧结点的任一子结点被标记为 F,则该结点约束为 F. 图 7(c)相对复杂一些,如果一个∧结点有 F 约束,并且它的一个子结点有 T 约束,那么它的另外一个结点将获得一个 F 约束. 图 8 表示析取约束规则,它的约束形式与合取约束相对偶,这里将不再赘述.

按照约束规则对程序的错误分析图中的所有结点进行初始化标记之后,需要标记出所有的 if 语句生成的非终结结点. 这些非终结结点不是程序的错误,if 语句结构的错误也不在我们考虑的范围之内,如若标记出 if 语句生成的非终结结点将会提高下一步算法的执行效率. 接下来算法 1 将自顶向下的按照约束规则对错误分析图进行二次标记,那些第一次标记和第二次标记冲突的结点将作为候选错误结点被记录下来. 二次

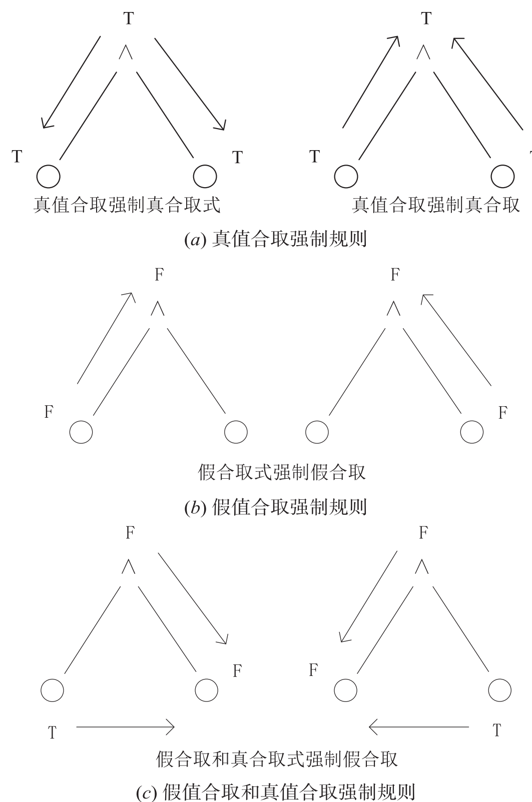
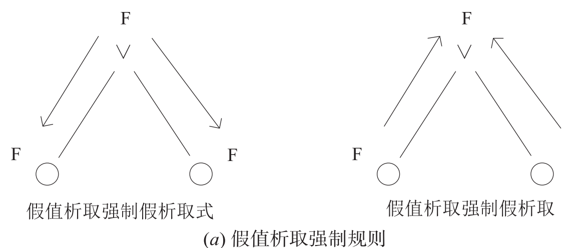
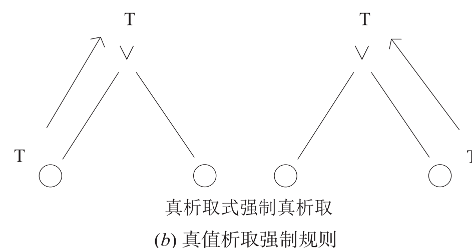


图7 合取约束规则





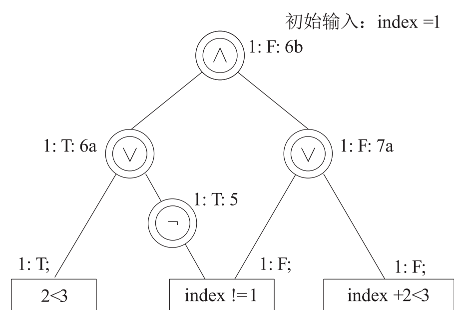
真析取和假析取式强制真析取

(c) 真值析取和假值析取强制规则

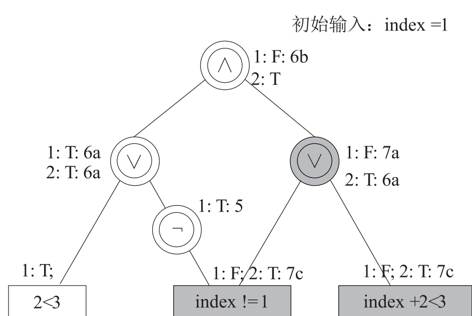
图8 析取约束规则

标记过程中值得注意的是图 7(*c*)和图 8(*c*)的情况，这两种情况的处理类似，这里仅详细的描述图 8(*c*)情况的处理方法.

以示例程序 1 为例，进行初始化标记和 if 结构标记之后的错误分析图如图 9(*a*)所示，其中双圆圈表示 if 结构结点，起始顶点标记为字符串 1: F: 6b，意味着第一次将当前结点按照规则 6b 标记为 F，其他节点类似. 图 9(*b*)表示对示例程序 1 的语法分析图进行二次标记之后，找到的几个疑似错误的结点(图中带阴影的结点). 值得注意的是在标记图 9(*b*)右侧的 ∨ 结点时发现冲突，之后需要用到约束规则图 8(*c*)"真析取和假析取式强制真析取"对它的两个子结点进行标记. 首先采用贪婪的思想，将当前结点的两个子结点都标记为"T"，则两个子结点都出现标记冲突，于是检测两个子结点的其他父节点，是否由于当前的二次标记跟顶点结点的标记值相冲突，如果冲突则放弃当前标记，否则当前结点为疑点. 最终排除 if 结构结点后，疑点展示给程序调试人员. 图 9(*b*)中的结点"index！ = 1"和"index + 2 < 3"即为两个疑点，分别对应示例程序 1 的第 3 行和第 6 行.



(*a*) 初始化标记后的错误分析图



(*b*) 二次标记后的错误分析图

图9　示例程序1的错误分析过程图

3.2.2　自顶向下的错误标记

在真实的程序中，可能存在两个或者多个错误语句. 而本文的方法也很容易扩展到包含两个或多个错误的程序错误定位中. 如果一个程序包含两个错误 A 和 B，显然可以将所有的违反断言的测试用例分成仅包含错误 A 导致断言违反的测试用例集 S_1，仅包含错误 B 导致断言违反的测试用例集 S_2 以及同时包含错误 A 和错误 B 的测试用例集 S_3 三个集合. 如果用 S_1 中的测试用例做为输入，获得的程序可疑点 $Susp_1$ 必然包含错误 A，同样 S_2 集合中的测试用例做为输入，必然能在定位结果 $Susp_2$ 中能找到错误 B. 因此要找到所有的错误 A 和 B，只要求得 $Susp_1 \cup Susp_2$ 即可. 在实际应用中可以简单的将所有的错误测试用例做为算法 1 的输入，再求结果的并集，即可获得程序的所有可疑语句.

4　实验分析

为了评估上述方法的有效性，本文使用 Siemens 基准测试组[19]作为实验对象，对其效果进行验证. 程序运行环境为双核 Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 处理器，内存为 16GB，操作系统为 64 位的 Windows 7.

4.1　实验对象

Siemens 基准测试组包含了 7 组实现不同功能的 C 程序，本文的实验主要是基于 TCAS 程序. TCAS 是一个飞行器冲突检测程序，包含 173 行代码，有 1 个正确版本和 41 个不同的错误版本，每个错误版本中包含 1 ~ 2 个缺陷，共 8 个错误类型，如表 1 所示[20]. 随 TCAS 程序发布的还有上千个测试用例，本文排除了任何不能正确执行和不能展示程序错误的那些测试用例(鉴于此，版本 V33 和 V38 所注入的数组越界错误不在本实验的范围之内)，剩下 1541 个测试用例. 在实际的实验中，为了简便和避免重复性实验，忽略了 if 条"enabled"相关的检测，用剩下的 27 个版本 TCAS 程序进行测试. 用这些测试用例在正确版本上执行，相应的程序输出作为正确输出.

表 1　TCAS 中的错误类型

错误类型	类型描述
op	运算符使用错误
code	逻辑编码错误
assign	赋值表达式错误
addcode	新增代码片段错误
const	常量错误
init	变量初始化错误
index	数组越界错误
branch	分支条件引起的分支错误

同时在所有的错误版本上运行了这些测试用例，并与正确输出作对比，得到失败的测试用例. 然而这些错误程序中并不包含待验证的断言，因此这些失败的测试用例作为反例，正确的输出值作为待验证的断言，具体信息见表 2. 表 2 中反例数指的是在当前测试用例下正确版本的运行结果与当前版本运行结果不同的测试用例总数. 第 3 列为程序待验证的断言，第 4 列为错误定位算法的评价指标值. 第 5 列为当前版本的程序包含的错误类型.

表 2 实验对象及实验结果

程序版本	反例数	待验证断言	score	错误类型
V1	140	alt_sep_test( ) =0	97.11	op
V2	48	alt_sep_test( ) =2	98.84	const
V3	51	alt_sep_test( ) =0	95.37	op
V4	55	alt_sep_test( ) =0	97.67	op
V5	10	alt_sep_test( ) =0	95.37	assign
V6	14	alt_sep_test( ) =0	97.69	op
V7	5	alt_sep_test( ) =2	97.69	const
V8	2	alt_sep_test( ) =1	97.69	const
V9	10	alt_sep_test( ) =2	98.84	op
V10	17	alt_sep_test( ) =0	95.95	op
V11	17	alt_sep_test( ) =0	95.95	op
V12	70	alt_sep_test( ) =0	95.37	op
V16	4	alt_sep_test( ) =2	97.11	init
V17	2	alt_sep_test( ) =2	97.11	init
V18	18	alt_sep_test( ) =2	97.11	init
V19	1	alt_sep_test( ) =2	97.11	init
V21	25	alt_sep_test( ) =0	96.531	op
V22	23	alt_sep_test( ) =1	98.841	code
V24	25	alt_sep_test( ) =0	95.951	op
V25	9	alt_sep_test( ) =2	98.27	code
V26	11	alt_sep_test( ) =0	95.37	addcode
V28	63	alt_sep_test( ) =2	98.27	Branch
V29	23	alt_sep_test( ) =1	98.27	code
V30	63	alt_sep_test( ) =2	98.27	code
V31	17	alt_sep_test( ) =0	95.95	addcode
V32	2	alt_sep_test( ) =0	97.11	addcode
V34	77	alt_sep_test( ) =0	95.95	op
V35	63	alt_sep_test( ) =2	98.27	code
V37	120	alt_sep_test( ) =1	98.27	index
V39	9	alt_sep_test( ) =2	98.27	op
V40	151	alt_sep_test( ) =2	97.6	assign
V41	51	alt_sep_test( ) =0	97.11	assign

4.2 实验结果与分析

为了与已有的错误定位方法进行对比，本文使用了 Renieris 和 Reiss 定义的评价指标[21]，即调试人员找到缺陷语句需要查看程序代码的工作量的大小，来衡量错误定位方法的精度，如式(8)所示. 得分越高，表明错误定位技术越有效，对于用户精确定位越有帮助. 表 2 第 4 列为本文错误定位算法的 T-score，可以看出本文的算法的定位精度较高，平均分值为 97.57.

$$\text{T-score} = \left(1 - \frac{\text{疑点数}}{\text{程序可执行代码行数}}\right) \times 100\% \quad (8)$$

本文算法与其他 7 种错误定位方法[13,20,22,23]，在版本 V1，V11，V31，V40 和 V41 上的实验结果如图 10 所示. 横坐标表示 5 个版本，纵坐标表示检查时可以忽略的代码行数占总代码行数的百分比. 当横坐标相同时，纵坐标值越大表明定位效果越好. 由图 10 可以看出，除了在版本 V11 上为 0.97，略低于 GSB 方法的 0.97 外，始终领先于其他 6 种方法，本文方法具有良好的定位效果，能够提供有效的错误定位信息.

图 11 采用箱式图统计了本文方法与其他两种错误定位方法 GSB 和 BugAssist 的 T-score 分布情况. 同样纵轴表示 T-score. 从图 11 可以看出在平均和最差的情况下，相比于 GSB 方法和 BugAssist，本文 WTREE 方法找到错误时需要检查的语句大为减少. 在最好的情况下，WTREE 和 GSB 方法具有最高的分值，BugAssist 方法分值稍低. 同时也可以看出，在稳定性方面 GSB 方法稍弱. 因此可以看出 WTREE 方法的实验效果更好，能保持较高的稳定性能.

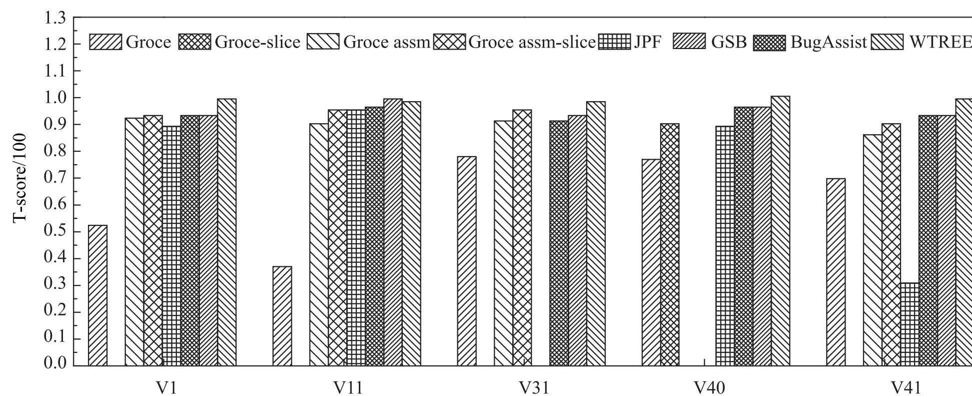


图10 多种方法的错误定位效果对比图

4.3 两个错误实例分析

TCAS 程序的 V10 包含两个错误，分别在源代码中的第 105 和 111 行，都属于 op 类错误. 程序 V10 需要满足断言 alt_sep_test( ) =0，包含有 17 个失败的测试用例，14 个输出结果为 1，3 个为 2. 因此，求得程序 V10 关于断言 alt_sep_test( ) =0 的最弱前置条件 $WP(P_{V10},$ alt_sep_test( ) =0)后可以画出它的错误分析图(如图 12 所示).

顺序的以 17 个失败的测试用例和它的错误分析图执行自顶向下的错误标记算法. 以 14 个输出结果为

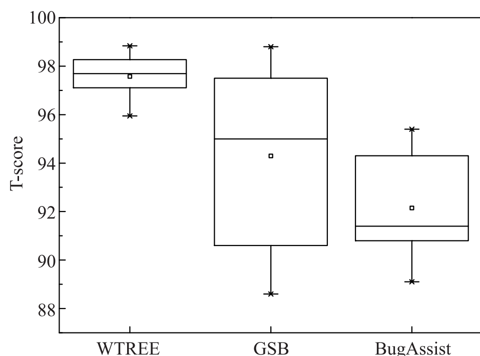
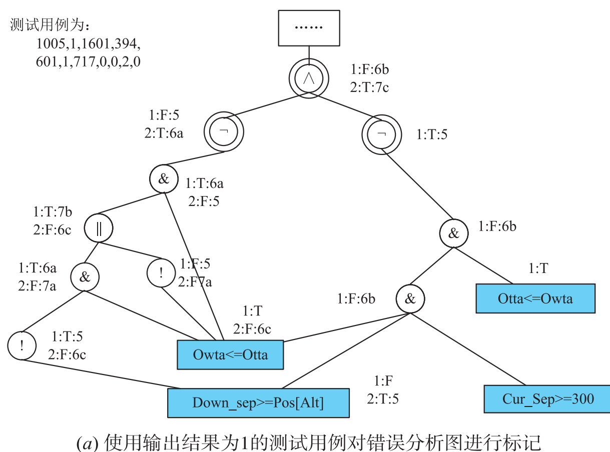
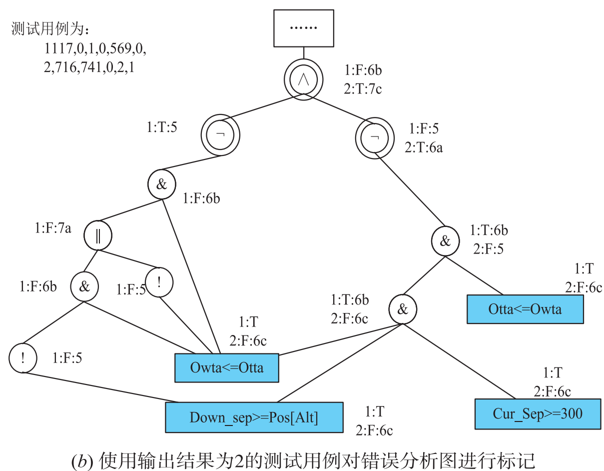


图11　三种错误定位方法的T–score箱式图

1 的测试用例为输入得到的标记结果如图 12(*a*)所示,以其他 3 个失败的测试用例作为输入的标记结果如图 12(*b*)所示. 其中图 12(*a*)和(*b*)中的灰色格标记的即为当前标记过程得到的疑点集,对应到源程序分别是 128,105,58,75 和 111,93,105,58,12 行代码,它们的并集为{128,105,58,75,111,93,12}共 7 行代码,因此本文算法在 TCASV10 版本上的错误定位精度为 T-score $= (1-7/173)\times 100 = 95.95$.



(*a*) 使用输出结果为1的测试用例对错误分析图进行标记



(*b*) 使用输出结果为2的测试用例对错误分析图进行标记

图12　TCAS V10的错误分析过程图

5　结论

本文提出了一种利用程序最弱前置条件和错误分析图的错误定位技术. 首先根据程序需要满足的断言,计算程序的最弱前置条件;其次构造最弱前置条件的错误分析图;最后,在错误分析图上进行自底向上初始化标记和自顶向下错误分析标记,找到两次标记的冲突结点,最终定位节点到相应的语句上. 通过实验研究与分析,本文算法能够有效的提高错误定位的精度,方便调试人员高效的定位到程序中的错误. 我们未来的研究工作是对本文提出的算法进行改进,在程序存在多个错误的时,找到一种规则或者策略选择失败的测试用例来提高二次标记的效率,更快的发现错误疑点.

参考文献

[1] JOSE M,MAJUMDAR R. Cause clue clauses:error localization using maximum satisfiability[J]. Acm Sigplan Notices,2010,46(6):437–446.

[2] WEN W,LI B,SUN X,et al. Program slicing spectrum-based software fault localization[A]. Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineerin [C]. Miami: DBLP, 2011. 213–218.

[3] HOLZMANN G J,SMITH M H. Software model checking [A]. Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification,Testing and Verification (PSTV XIX)[C]. Netherlands:ACM,1999. 1729–1739.

[4] 文万志,李必信,孙小兵,等. 一种基于层次切片谱的软件错误定位技术[J]. 软件学报,2013(5):977–992.
WEN W-Z,LI B-X,SUN X-B,et al. Technique of software fault localization based on hierarchical slicing spectrum [J]. Journal of Sotfware,2013(5):977–992. (in Chinese)

[5] LEI Y,MAO X,DAI Z,et al. Effective statistical fault localization using program slices[A]. Proceedings of the IEEE 36th Annual Computer Software and Applications Conference[C]. Washington DC:IEEE,2012. 1–10.

[6] 曹鹤玲,姜淑娟,鞠小林,等. 基于动态切片和关联分析的错误定位方法[J]. 计算机学报,2015(11):2188–2202.
CAO H-L,JIANG S-J,JU X-L,et al. Fault localization based on dynamic slicing and association analysis[J]. Chinese Journal of Computers,2015(11):2188–2202. (in Chinese)

[7] ZHANG X,HE H,GUPTA N,et al. Experimental evalua-

tion of using dynamic slices for fault location[A]. Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging[C]. California: ACM, 2005.33 – 42.
[8] 赵磊,王丽娜,高东明,等.基于关联挖掘的软件错误定位方法[J].计算机学报,2012,35(12):2528 – 2540.
ZHAO L, WANG L-N, GAO D-M, et al. Mining associations to improve the effectiveness of fault localization[J]. Chinese Journal of Computers, 2012, 35(12):2528 – 2540. (in Chinese)
[9] ABREU R, ZOETEWEIJ P, VAN GEMUND A J C. On the accuracy of spectrum-based fault localization[A]. Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques[C]. Washington DC: IEEE, 2007.89 – 98.
[10] JONES J A. Empirical evaluation of the tarantula automatic fault-localization technique[A]. Proceedings of the 20th IEEE/ACM International Conference on Automated software Engineering[C]. New York: ACM, 2005.273 – 282.
[11] 王建峰,魏长安,盛云龙,等.基于错误交互集的组合测试软件故障定位方法[J].电子学报,2014,42(6):1173 – 1178.
WANG Jian-feng, WEI Cang-an, SHENG Yun-long, et al. Locating errors in combinatorial testing using set of possible faulty interactions[J]. Acta Electronica Sinica, 2014, 42(6):1173 – 1178. (in Chinese)
[12] 曹鹤玲,姜淑娟.基于 Chameleon 聚类分析的多错误定位方法[J].电子学报,2017,45(2):394 – 400.
Cao He-ling, JIANG Shu-juan. Multiple-fault localization based on chameleon clustering[J]. Acta Electronica Sinica, 2017, 45(2):394 – 400. (in Chinese)
[13] GRIESMAYER A, STABER S, BLOEM R. Fault localization using a model checker[J]. Software Testing Verification & Reliability, 2010, 20(2):149 – 173.
[14] WANG C, YANG Z, IVANČIĆ F, , et al. Whodunit? Causal analysis for counterexamples[J]. Lecture Notes in Computer Science, 2006, 4218:82 – 95.
[15] 黄宏涛,黄少滨,陈志远,等.基于克雷格插值的反例理解方法[J].吉林大学学报:理学版,2013,51(1):94 – 100.
HUANG H-T, HUANG S-B, CHEN Z-Y, et al. Understanding counterexamples based on craig interpolation[J]. Journal of Jilin University (Science Edition), 2013, 51(1):94 – 100. (in Chinese)
[16] JOSE M, MAJUMDAR R. Cause clue clauses: error localization using maximum satisfiability[J]. Acm Sigplan Notices, 2010, 46(6):437 – 446.
[17] 徐勇,毋国庆,袁梦霆.结合 Craig 插值分析的软件错误诊断方法[J].电子学报,2016,44(10):2514 – 2521.
XU Yong, WU Guo-qing, YUAN Meng-ting. Software fault localization based on model-Based diagnosis combined craig interpolant analysis[J]. Acta Electronica Sinica, 2016, 44(10):2514 – 2521. (in Chinese)
[18] HUTH M, RYAN M. Logic in Computer Science: Modeling and Reasoning About Systems[M]. London: Cambridge University Press, 2004.201 – 205.
[19] HUTCHINS M, FOSTER H, GORADIA T, et al. Experiments on the effectiveness of dataflow and control-flow-based test adequacy criteria[A]. Proceedings of 16th International Conference on Software Engineering[C]. USA: IEEE, 1994.191 – 200.
[20] JOSE M, MAJUMDAR R. Cause clue clauses: error localization using maximum satisfiability[J]. Acm Sigplan Notices, 2010, 46(6):437 – 446.
[21] RENIERIS M, REISS SE. Fault localization with nearest neighbor queries[A]. Proceedings of the 18th International Conference Automated Software Engineering (ASE 2003)[C]. Quebec: IEEE, 2003.30 – 39.
[22] GROCE A, CHAKI S, KROENING D, et al. Error explanation with distance metrics[J]. International Journal on Software Tools for Technology Transfer, 2006, 8(3):229 – 247.
[23] GROCE A, VISSER W. What went wrong: explaining counterexamples[A]. Proceedings of the 10th International Conference on Model Checking Software[C]. Portland: Springer-Verlag, 2002.121 – 136.

作者简介

李 雅 女,1985 年 9 月出生,河南郑州人.2007 年、2017 年分别在哈尔滨工业大学、哈尔滨工程大学获工学学士、工学博士学位.现为黑龙江工程学院计算机科学技术学院讲师,主要从事模型检测及软件错误定位方面的研究工作.
E-mail: liya_heu@163.com

黄少滨 男,1965 年出生,黑龙江哈尔滨人.教授、博士生导师.现为哈尔滨工程大学计算机科学与技术学院教授,主要从事分布式计算与仿真以及模型检测方面的研究工作.
E-mail: huangshaobin@hrbeu.edu.cn