软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(7):1455−1468 [doi: 10.3724/SP.J.1001.2013.04310]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于最小调试边界的断点自动生成技术*

李 丰, 霍 玮, 陈聪明, 李 龙, 衷璐洁, 冯晓兵

(计算机体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

通讯作者: 李丰, E-mail: lifeng2005@ict.ac.cn

摘 要: 时至今日,调试仍然占据软件开发过程中近 70%的时间;以断点的设置和检查为基础的传统交互式调试依旧是实际工作中最常用的错误定位手段.日常调试过程中,断点的选择和调试的效率主要依赖于调试人员自身的经验以及对所调试程序的理解程度.提出一种基于最小调试边界的断点自动生成方法.最小调试边界描述了一个由程序执行轨迹上一组轨迹点构成的结合.该集合具有对错误传播的阻隔性,以及所对应的程序状态规模最小化的特征.受益于最小调试边界(minimum debugging frontier set,简称 MDFS)的错误阻隔性,一旦查明其上的程序状态是否符合设计预期,即可确定错误触发位置与该 MDFS 在程序执行轨迹上的先后顺序,将错误触发的范围限定在 MDFS 的一侧.而状态规模的最小化也减轻了用户检查断点处语句实例的开销.为评价断点质量,还制定了一组断点评价标准,用于考量断点与程序失效之间的关联性、断点本身的易判性以及对调试收敛的帮助.实验结果表明,采用该方法生成的断点具有检查开销低、加速调试收敛等优势;采用所提供的断点的调试流程,与基于经典错误定位方法的流程相比,能以更低的人工开销定位更多的错误.

关键词: 调试;断点;最小调试边界;错误定位;依赖分析

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 李丰,霍玮,陈聪明,李龙,衷璐洁,冯晓兵.一种基于最小调试边界的断点自动生成技术.软件学报,2013,24(7):1455−1468. http://www.jos.org.cn/1000-9825/4310.htm

英文引用格式: Li F, Huo W, Chen CM, Li L, Zhong LJ, Feng XB. Automatic breakpoint generating approach based on minimum debugging frontier set. Ruan Jian Xue Bao/Journal of Software, 2013,24(7):1455−1468 (in Chinese). http://www.jos.org.cn/1000-9825/4310.htm

Automatic Breakpoint Generating Approach Based on Minimum Debugging Frontier Set

LI Feng, HUO Wei, CHEN Cong-Ming, LI Long, ZHONG Lu-Jie, FENG Xiao-Bing

(State Key Laboratory of Computer Architecture (Institute of Computing Technology, The Chinese Academy of Sciences), Beijing 100190, China)

Corresponding author: LI Feng, E-mail: lifeng2005@ict.ac.cn

Abstract: Until recently, debugging still takes almost 70% of the time in software engineering. The conventional debugging process, based on setting breakpoints and inspecting the states on them, remains the most common and useful way to detect faults. The efficiency of debugging differs a lot as both the selection and inspection of breakpoints are up to programmers. This paper presents a novel breakpoint generating approach based on a new concept named minimum debugging frontier sets (abbr. MDFS). A debugging frontier set describes a set of trace points, which have the ability of bug isolation, and a MDFS is the one with minimum size. Benefiting from the ability of bug isolation, the error suspicious domain will always be narrowed down to one side of the MDFS no matter the state of MDFS is proven correct or not. Breakpoints generated on the basis of MDFS also make the statement instances to be inspected at each breakpoint at the minimum. The paper also establishes a set of criterions to judge the quality of breakpoints. Empirical result indicates that breakpoints generated through this approach not only require low inspecting cost, but also have the ability to accelerate the efficiency of

* 基金项目: 国家自然科学基金(61100011, 60921002); 国家重点基础研究发展计划(973)(2011CB302504); 国家高技术研究发展计划(863)(2012AA010901); 国家核高基重大专项(2011ZX01028-001-002)

收稿时间: 2012-04-23; 修改时间: 2012-07-23; 定稿时间: 2012-08-20

debugging. It also shows that this MDFS-based debugging prototype performs better than the state-of-art fault-localization techniques on the Siemens Suite.

Key words: debugging; breakpoint; minimum debugging frontier set; fault localization; dependence analysis

长期以来,软件从业者为软件质量的提高付出了诸多努力.然而,开发完全无错的程序依然任重道远.尽管各类自动错误检测与错误定位工具[1−10]层出不穷,以人为主导的交互式调试仍然是最通用的追踪错误根源的手段.常用的源码级调试工具,如 GNU 调试器(GDB)、微软的 Visual Studio 调试器等,均提供丰富的辅助设施,其中最常用的基础设施是对断点设置与程序状态打印的支持.

传统的交互式调试流程包括 3 步:

1) 复现错误执行轨迹,并将该轨迹作为初始的错误怀疑范围;
2) 在怀疑范围内设置一个检查位置,也称断点,由调试人员判断该位置上的程序状态是否正确;
3) 根据第 2)步的判断结果精化错误怀疑范围,如果精化后的怀疑范围规模仍过于庞大,无法使调试人员直接发现错误源,则返回步骤 2);否则,报告错误源,调试收敛.

上述过程是一个将错误怀疑范围从全程序执行轨迹向错误源被触发的位置逐渐收敛的过程.其间包含若干次迭代,每次迭代对应一次断点选择与检查.对断点处的程序状态的检查可以等价为对程序执行到断点所在位置时所有活跃变量的值的检查.由于断点位置通常采用源程序行号或函数名称的形式标识,每个断点可能对应动态执行轨迹上的多个位置.为避免混淆,下文用“程序点”指代静态程序代码中的位置,用“轨迹点”指代动态执行轨迹上的位置.

优质的断点是促进调试迅速收敛的关键.然而,在实际调试过程中,断点的选择很大程度上依赖于调试人员自身的经验以及对所调试程序的理解程度.统计表明,当程序行为出现异常时,编程人员对责任代码的初步猜测几乎总是错误的(90%)[11].研究人员为提高断点质量与调试效率,提出了一些自动推荐断点的方法.该类技术以公认的精度较高的自动错误定位方法为依托,综合考虑由错误定位方法获得的错误候选集合以及调试历史生成候选断点,供调试人员选择.代表性的断点推荐工具如 BPGen[12],VIDA[13]等.目前,对上述领域的研究主要存在 3 个问题:1) 缺乏评价断点质量的标准;2) 断点的选择仍由人工完成,没有从根本上减轻调试人员的负担;3) 断点的生成效率依赖于底层错误定位方法的效率.

本文提出一组评价断点质量的标准,并以该标准为衡量准则,提出了一种基于最小调试边界的断点自动生成方法.

本文认为,断点除包含位置信息外,还应包含该位置上需要检查的内容(语句实例).在此基础上,一个优质的断点应满足如下 3 个性质:关联性、易判性和收敛性.本文第 3 节将具体介绍上述 3 个性质.

为获得满足上述标准的断点,本文提出一种基于最小调试边界的断点自动生成方法.最小调试边界描述了一个由程序执行轨迹上一组轨迹点构成的集合.该集合具有对错误传播的阻隔性以及所对应的程序状态规模最小化的特征.受益于最小调试边界的错误阻隔性,一旦查明其上的程序状态是否符合设计预期,即可确定错误触发位置与该调试边界在程序执行轨迹上的先后顺序,将错误触发的范围限定在该调试边界的一侧.而状态规模的最小化又减轻了用户检查断点的开销.实验结果表明,采用本文方法生成的断点具有检查开销低、加速调试收敛等优势;采用本文提供的断点的调试流程与以经典错误定位方法为指导的调试流程相比,能以更低的人工检查开销定位更多的错误.

本文的主要贡献如下所示:

- 提出最小调试边界的概念.该概念描述了程序执行轨迹上一组具备错误阻隔性和状态信息最小化的轨迹点;
- 提出一组评价断点质量的标准:关联性、易判性和收敛性;
- 提出一种基于最小调试边界的断点自动生成方法.藉由该方法生成的断点,具备关联性、易判性和收敛性.实验结果表明,其对调试的辅助效果优于经典错误定位方法.

本文第 1 节介绍背景技术.第 2 节通过一段程序示例说明断点选择对调试收敛的影响,之后介绍最小调试

边界的概念以及基于最小调试边界的断点生成方法,并阐述如何根据本文生成的断点进行程序调试.第 3 节通过实验分析,说明由本文技术提供的断点具有判断开销低、促进调试收敛的作用.第 4 节介绍断点推荐以及错误定位领域的相关工作.第 5 节总结全文.

1 技术背景

1.1 动态依赖图与动态切片

程序语句间的依赖关系是由语句间数据流或控制流形成的一种约束,分别称为数据依赖和控制依赖.语句 S_2 数据依赖于 S_1,当且仅当 S_1 中定义了变量 x,S_2 中使用变量 x,并且存在一条从 S_1 到 S_2 的非空路径,该路径上没有对 x 的定值.如果存在一条从 S_1 到 S_2 的非空路径,S_2 后控制该路径上除 S_1 之外的所有节点,但不后控制 S_1,则称语句 S_2 控制依赖于 S_1.

程序 P 在输入为 I 时的动态依赖图,记作 $dPDG(P,I)$,记录了程序当次执行轨迹上各语句实例之间的动态依赖关系.其中,结点代表被执行的语句实例,边代表语句实例间在动态执行时的依赖关系.由于程序代码中的每条语句均可能对应多个执行实例,在不采取任何优化策略的前提下,每条语句的每个执行实例将对应动态依赖图中的一个结点.动态依赖图中有两个特殊结点,*ENTRY* 和 *EXIT*,分别代表图的入口结点和出口结点.

动态后向切片技术[14,15]是一项程序轨迹提取技术.以语句 S 的某个执行实例为切片标准,动态切片能够从程序执行轨迹中提取出可能影响语句 S 当次执行结果的所有语句实例集合.提取出的语句实例的集合称为动态切片.动态切片可以表示成动态依赖图子图的形式.

1.2 最小割

图论中,割[16]是图的一个划分:将图中结点分成两组结点集合,属于不同集合的结点之间互不连通.有向图中,割的大小为跨越它的所有有向边的数量,这些有向边的集合称为该图的割集;其中,包含有向边数量最少的割集称为该图的最小割集.根据图论中的最大流-最小割定理,计算有向无权图 $G=(V,E)$割的问题可以转化为求解最大流的问题.目前,存在多种计算最大流的算法.经典的 Edmonds-Karp 算法的时间复杂度为 $O(|V|\times|E|^2)$;虽然其复杂度高于压入重标记类算法等后续出现的算法[17],但在处理稀疏图时颇具优势.

2 基于最小调试边界的自动断点生成方法

第 2.1 节讨论了对断点质量的需求,并定义了优质断点需要满足的 3 个性质:关联性、易判性和收敛性;第 2.2 节提出最小调试边界的概念,并证明基于最小调试边界设置的断点满足上述 3 个性质;第 2.3 节和第 2.4 节分别介绍基于最小调试边界的自动断点生成方法以及基于最小调试边界的程序调试方法.

2.1 断点评价方法

2.1.1 优质断点需求

本节通过一段缺陷代码说明调试过程中对断点的要求.

图 1 所示是取自文献[5]的一段串行程序,该程序的功能是采用希尔排序算法将输入的整数按照升序排列,但误将 *main* 函数输入参数 *argc* 的值作为待排序的整数的数目(正确的数目应为 *argc*–1).当用户为输入整数序列 14 11 时,第 24 行输出的序列为 0 11,而预期输出应为 11 14.

一种极限情况是,假设调试人员在每条语句执行之后设置一个断点.失效执行轨迹上从错误源的触发位置(第 22 行函数调用语句的第 1 个执行实例)到失效表现位置(第 24 行的 *printf* 语句)之间的语句实例数量为 26,其中涉及错误传播的语句实例间的动态依赖构成的错误传播链的长度为 10.如果调试过程中能够排除与失效无关的程序行为的干扰,需经过 10 步逆向追踪,方可确定错误源;如果无法排除无关干扰,则需要 26 步.

另一种极限情况是,调试人员可以自由地设置断点,并且假设他们能够准确知晓程序执行轨迹上各变量在不同轨迹点上预期的值.但上述假设也不能保证对所设断点的检查能够有效地缩小错误怀疑范围.例如,调试人

员在第 8 行 for 循环的循环体第 2 次执行之前设置断点,检查当时数组 *a* 中的内容.检查结果表明,此时数组 *a* 中的内容正确,故其推断引发失效的原因出现在断点之后.最坏情况下,调试人员在检查完位于断点之后的全部执行轨迹后,才发现其对错误所在范围的推断是错误的.产生上述错误推断的原因是调试人员没有完整地检查断点处所有影响可能到达失效表现位置的语句的计算结果,比如循环条件"*i*<*size*"的真假.

```
    static void shell_sort(int a[],int size)
    {
1       int i, j;
2       int h=1;
3       do {
4           h=h×3+1;
5       } while (h<=size);
6       do {
7           h/=3;
8           for (i=h; i≤size; i++)
9           {
10              int v=a[i];
11              for (j=i; j>=h && a[j−h]>v; j−=h)
12                  a[j]=a[j−h];
13              if (i!=j)
14                  a[j]=v;
15          }
16      } while (h!=1);
    }
    int main(int argc,char*argv[])                          //输入:./a.out 14 11
    {
17      int i=0;
18      int*a=NULL;

19      a=(int*)malloc((argc−1)*sizeof(int));
20      for (i=0; i<argc−1; i++)
21          a[i]=atoi(argv[i+1]);

22      shell_sort(a,argc);                                 //第2个参数应为argc−1

23      for (i=0; i<argc−1; i++)
24          printf("%d",a[i]);
25      printf("\n");                                       //输出:0 11, 正确输出:11 14

26      free(a);
27      return 0;
    }
```

Fig.1 Sample program

图 1 示例程序

只有当所有在断点前定义并在后续执行轨迹上被使用,且影响可传播到失效表现位置的所有语句实例均被认定为正确,才能确定错误源在断点之后被触发.满足上述条件的断点构成了程序执行轨迹(或动态依赖图)上的一道"屏障",它将程序执行轨迹划分成两个子部分;"屏障"对应的程序状态的正确与否,决定了位于"屏障"哪一侧的子轨迹将作为新一轮调试迭代的错误怀疑范围.由上述分析可知,设置具备"屏障"属性的断点可以提高调试效率、加快调试收敛.同时,对断点处程序状态的检查开销也不容忽视:即便所选断点具备"屏障"属性,如果其包含的语句实例的数量过多,调试人员的检查负担过重,依旧不利于调试效率.故,所选"屏障"类型的断点还应具有相对低的检查开销.本节将介绍的基于最小调试边界的断点自动生成方法就是一种旨在寻找满足上述要求的断点的方法.

2.1.2 断点评价标准

本文认为,断点除包含位置信息外,还应包含该位置上需要检查的语句实例的集合.一个优质的断点至少可以根据以下 3 个性质进行评价:

1) 关联性.并非每条语句的执行都与程序的失效表现之间存在直接或间接的关联.为减轻调试人员分析和理解程序的负担,应尽量避免对与失效表现无关的程序语句及其变量的检查.因此,判断断点质

量的依据之一是,其中包含的需要检查的语句实例是否是程序失效所直接或间接依赖的语句实例;

2) 易判性.程序执行轨迹上某个位置对应的程序状态表现为一组在该轨迹点上活跃的变量的值.为了判断断点处的程序状态是否符合设计意图,调试人员必须明确程序执行到该位置时,所有活跃变量的预期计算结果.优质的断点应尽可能地控制对其上程序状态的检查开销,即:包含尽可能少的与程序失效相关的活跃变量的数量;

3) 收敛性.断点设置与检查的目的是缩小对错误的怀疑范围,并最终促使调试收敛.因此,判断断点质量的另一个依据是其对错误怀疑范围的收敛作用,并且收敛后的错误怀疑范围内包含错误源语句.

2.2 最小调试边界(minimum debugging frontier set,简称MDFS)的定义及性质

本文采用 Agrawal 和 Horgan 的第 3 种动态依赖图构造算法[14],所构造出的动态依赖图是无环的.动态依赖图的子图记作 $dPDG(P,I,V_{begin},V_{end})$.其中,$V_{begin}$ 和 V_{end} 分别指代子图的根结点集合与漏结点集合.子图仅包含位于根结点与漏结点间路径上的所有结点和边.如果有 $V_{begin}=\{ENTRY\},V_{end}=\{EXIT\}$,则当前子图即为完整的动态依赖图 $dPDG(P,I)$.图上每个结点对应执行轨迹上的一个轨迹点,以及该点处的程序状态(下文简称状态).给定程序 P 在输入为 I 时的执行轨迹,该轨迹对应的动态依赖图记作 $dPDG(P,I,V_{begin},V_{end})$,一组结点集合(记作 C)称为该图的割点集,当且仅当删除 C 中所有的结点后,图中不存在从 V_{begin} 中任一结点到达 V_{end} 中任一结点的路径.

定义 2.1. 给定动态依赖图 $dPDG(P,I,V_{begin},V_{end})$和动态执行轨迹上一组轨迹点的集合 *trace_pt_set*,*trace_pt_set* 对应的动态依赖图上的结点集合记作 *v_set*,*v_set* 的最小调试边界是动态依赖子图 $dPDG(P,I,V_{begin},v\_set)$ 的所有割点集中包含结点数量最少的,记作 $mdfs(P,I,V_{begin},v\_set)$.

以图 1 所示的缺陷程序为例,其在图 1 中所指定的输入下生成的动态依赖图片段以及失效节点的最小调试边界如后文的图 3(b)所示.由图 3(b)可知,该结点的最小调试边界包含两个语句实例;删除上述语句实例对应的结点后,动态依赖图将被划分成两个互不连通的片段:DDG^0 和 DDG^1.

由定义 2.1 可知,最小调试边界是基于动态依赖图定义的,故图中某个结点的最小调试边界中包含的语句实例必与结点间存在直接或间接的动态依赖关系.因此,依据最小调试边界设置断点,可以确保断点处的程序状态与失效表现之间的关联性.

假定程序 P 在输入为 I 的前提下,在第 i 次执行到语句 S 时观察到程序失效.为方便描述,这里假设失效表现仅对应动态依赖图上的一个结点 v.令 *mdfs* 为 v 的最小调试边界,*mdfs* 包含的每个结点对应 P 的执行轨迹上的一个语句实例.下文用动态依赖图上结点泛指结点所代表的语句实例.

由定义 2.1 可以得到下述两个推论:

推论 2.1. 如果 *mdfs* 中包含的所有语句实例的计算结果均符合设计预期,则本次执行过程中,错误源的触发位置必然位于子图 $dPDG(P,I,mdfs,\{v\})$所对应的执行轨迹上.

证明:根据定义 2.1,删除 $dPDG(P,I)$中所有属于 *mdfs* 的结点后,从图的入口结点 *ENTRY* 到 v 是不可达的.假设导致本次失效执行的错误源的触发位置位于 *mdfs* 中任何一个结点对应的轨迹点之前,则错误源必须经由 *mdfs* 中至少一个结点作为传播的中间媒介方可到达结点 v.由此可知,*mdfs* 中至少有一个结点对应的语句实例的计算结果是受到错误源污染的,即其计算结果是有误的.这就与"*mdfs* 中包含的所有语句实例的计算结果均符合设计预期"的前提产生矛盾.假设不成立.因此,错误源的触发位置只能位于 *mdfs* 之后,即 $dPDG(P,I,mdfs,\{v\})$ 所代表的执行轨迹上. □

推论 2.2. 如果 *mdfs* 包含的所有语句实例中至少有一个的计算结果被确认是不符合设计预期的,则本次执行过程中,错误源的触发位置必然位于子图 $dPDG(P,I,\{ENTRY\},mdfs)$所代表的执行轨迹上.

证明:在程序执行轨迹上,所有在错误源触发位置之前执行的语句实例的计算结果都是正确的.因此,如果错误源的触发位置位于 *mdfs* 中所有结点对应的轨迹点之后,则有以下结论:*mdfs* 中包含的所有语句实例的计算结果均正确.该结论与"*mdfs* 包含的所有语句实例中至少有一个的计算结果被确认是不符合设计预期"的前提相矛盾.故假设不成立,错误源的触发位置只能位于 $dPDG(P,I,\{ENTRY\},mdfs)$代表的执行轨迹上. □

推论 2.1 和推论 2.2 描述了最小调试边界对错误传播的阻隔性,即一旦查明最小调试边界中各语句实例的

计算结果是否与预期结果相符,即可将对错误源触发位置的怀疑范围缩小到当前分析范围内位于最小调试边界某一侧的部分.这一特征确保了依据最小调试边界设置的断点对缩小故障怀疑范围的促进效果,即断点的收敛性.同时,该概念还确保了最小调试边界中包含的语句实例的数量少于所分析轨迹上的其他所有位置,从而降低了人工检查断点的开销,提高了断点的易判性.推论 2.1、推论 2.2 同样适用于一组结点集合的最小调试边界.

综上所述,依据最小调试边界设置断点,既为断点设置的位置、断点处需要检查的内容提供了参考,又能确保:(1) 断点的关联性:断点处的程序状态与程序失效表现间存在直接或间接的动态依赖关系;(2) 断点的易判性:断点处需要检查的与失效表现相关的语句实例的数量相对于当前分析的轨迹范围内的其他位置而言是最少的;(3) 断点的收敛性:一旦能够确定需要检查的语句实例的计算结果与预期结果间是否符合,即可确定将对错误源触发位置的怀疑范围缩小到当前分析范围内位于断点一侧的部分.

2.3 自动断点生成算法

本节将介绍最小调试边界的计算方法,以及如何基于最小调试边界设置断点.算法流程如图 2 所示.

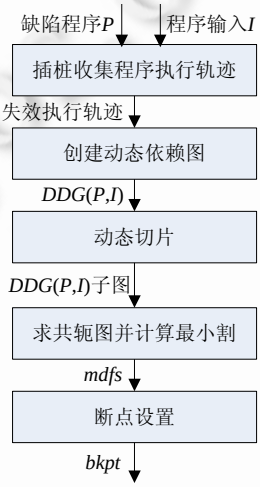


Fig.2 Automatic breakpoint generating algorithm

图 2 自动断点生成算法流程图

该方法以动态依赖图为基础中间表示形式.动态依赖图的构造采用文献[14]中介绍的第 3 种方法:每条语句的每个执行实例对应动态依赖图中一个唯一的结点.动态依赖关系以及程序执行过程中各语句实例的计算结果通过程序动态插桩技术记录.集合 $Var=\{var_1,var_2,\ldots,var_n\}$代表程序 P 中的所有变量的集合.其中,每个变量关联一个初始值为 0 的实例号.程序执行过程中,变量每更新 1 次,关联的实例号加 1:代表创建了该变量的一个新的实例.

假定程序 P 在输入为 I 的前提下,执行到结点 v 对应的语句实例时观察到程序失效或出现程序执行行为不符合预期的情况,则以 v 为切片标准,对动态依赖图进行后向切片.所获得的动态切片可以映射成(完整)动态依赖图的一个子图.子图中的每个结点对应程序执行轨迹上一个唯一的轨迹点.每个轨迹点处的程序状态表现为在该点处活跃的并且与观察到的失效行为间存在(直接或间接的)依赖关系的所有变量实例的值.这些变量实例的值已通过动态插桩记录.

上述子图的最小调试边界求解问题可以转化为共轭图上求解最小割集的问题.将动态依赖子图上的结点与边互换,转换后的动态依赖子图即为共轭图.本文根据图论中的最大流最小割定理,将共轭图上最小割的求解转化成最大流问题,并采用经典的 Edmonds-Karp 算法求解最大流.共轭图的最小割集中的每条割边对应转换之前图上的一个结点.这些结点就构成了动态依赖子图上的最小调试边界.

在根据最小调试边界设置断点时,断点位置为最小调试边界中最后执行的语句实例对应的轨迹点,断点处

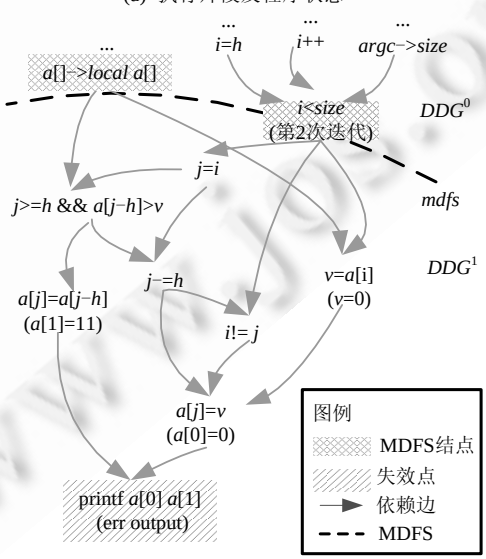
需要检查的内容即最小调试边界中包含的所有结点对应的语句实例.如果当前分析范围内存在多个最小调试边界,本文默认选择失效执行轨迹上最靠近失效表现位置的最小调试边界作为断点设置的依据.

断点的设置可以采用中断响应的形式来实现.由于动态依赖图的每个结点对应一个语句实例,基于 MDFS 生成的断点可以通过中断的形式准确地插入到指定的语句实例所对应的轨迹点上.断点处需要检查的语句实例的集合将通过中断处理函数输出给用户.

以图 1 所示的程序为例,图 3(a)所示为该程序在输入序列为 14 11 时的部分执行轨迹,自动断点生成算法将首先根据插桩记录的执行轨迹创建动态依赖图.图 3(b)中带斜纹的结点为错误输出对应的结点,自动断点生成算法以该结点为切片标准计算动态切片,切片结果对应的动态依赖子图片段如图 3(b)所示.而后,算法将图 3(b)中的边和点互换,生成动态依赖图的共轭图,并求解共轭图上的最小割,并将最小割中所包含的边逆向映射到图 3(b)中的结点上,即图 3(b)中用 X 纹标识的结点.黑色虚线代表由上述结点构成的最小调试边界 *mdfs* 所在的位置;由图 3(b)可知,代表 *mdfs* 的黑色虚线将动态依赖图划分成两个子图:DDG^0 和 DDG^1.基于 *mdfs* 生成的第 1 个断点将位于源程序第 8 行 for 循环的循环条件(图 1 中带下划线的语句)的第 2 个执行实例执行之后;其上需要检查的内容包括循环条件 *i*<*size* 和数组 *a* 的值.

… //*main*函数	for (…; …; *i*++)
a[0]=*atoi*(...); //*a*[0]=14;	for (…; *i* < *size*; …) //第2次迭代
…	*v*=*a*[*i*]; //*v*=*a*[2];
a[1]=*atoi*(...); //*a*[1]=11;	for (*j*=*i*; …; …)
… //调用*shell_sort*函数	for (…; *j*>=*h* && *a*[*j*−*h*]>*v*; …)
for (*i*=*h*; …; …)	*a*[*j*]=*a*[*j*−*h*]; //*a*[2]=*a*[1];
for (…; *i*<*size*; …) //第1次迭代	for (…; …; *j*−=*h*)
v=*a*[*i*]; //*v*=*a*[1];	for (…; *j*>=*h* && *a*[*j*−*h*]>*v*; …)
for (*j*=*i*; …; …)	if (*i*!=*j*)
for (…; *j*>=*h* && *a*[*j*−*h*]>*v*; …)	*a*[*j*]=*v*; //*a*[1]=*v*;
a[*j*]=*a*[*j*−*h*]; //*a*[1]=*a*[0];	for (…; …; *i*++)
for (…; …; *j*−=*h*)	for (…; *i*<*size*; …) //False
for (…; *j*>=*h* && *a*[*j*−*h*]>*v*; …)	while (*h*!=1) //False
if (*i*!=*j*)	…
a[*j*]=*v*; //*a*[0]=*v*;	print *a*[0] *a*[1]

(a) 执行片段及程序状态



(b) 动态依赖图片段及最小调试边界示例

Fig.3 An example of MDFS

图 3 最小调试边界实施示例

2.4 基于MDFS断点的程序调试方法

本节将介绍如何使用第 2.3 节生成的断点(下文简称 MDFS 断点)调试程序.图 4 所示为调试流程,包括 4 个关键阶段:

1) 精化错误怀疑范围:初始的错误怀疑范围是以程序失效为切片标准得到的后向切片;之后的每轮迭代以上一轮迭代新证实的传播媒介语句为切片标准,从当前错误怀疑范围对应的动态依赖子图中提取新的错误怀疑范围;
2) 自动生成断点:按照第 2.3 节所述方法生成 MDFS 断点;
3) 检查断点状态:检查MDFS断点包含的各语句实例的计算结果是否符合程序的设计预期,如果发现计算结果不符合设计预期的语句实例,则将其记录为错误传播的一个中间媒介语句,供后续迭代使用;
4) 判断调试收敛:如果当前错误怀疑范围内不包含未经检查的语句实例,则终止调试并输出候选错误集合;否则,返回步骤 1).

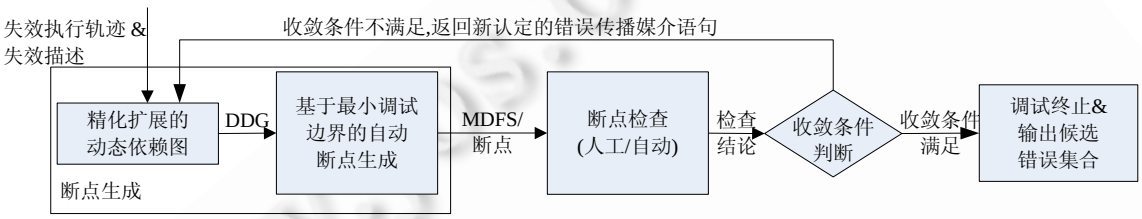


Fig.4 Framework of the MDFS-based debugging process

图 4 基于 MDFS 的程序调试流程

下面我们仍将以图 1 所示的程序为例,说明如何使用 MDFS 断点进行调试.

调试开始之前,先通过动态插桩记录错误执行轨迹.以输出语句为切片标准计算动态切片,并将其映射成动态依赖图的子图,作为计算断点的输入.

如第 2.3 节所述,首个自动生成的断点位于源程序第 8 行 for 循环的循环条件(图 1 中带下划线的语句)的第 2 个执行实例执行之后;断点处需要判断的内容包括:1) 循环条件 *i*<*size* 的计算结果是否应当为 True;2) 数组 *a* 的内容是否正确.

希尔排序算法的关键是对增量序列的选择.由于本例中待排序的整数只有两个,故增量为 1.由此可以推断出第 8 行的 for 循环只能执行 1 次;换言之,循环条件 *i*<*size* 的第 2 次执行结果应为 False.即便对于不了解希尔排序算法的程序员也可以发现:将循环条件 *i*<*size* 的第 2 次执行结果改为 False 后,即可获得预期输出.由此推断错误源位于断点之前.故,以 *i*<*size* 在失效执行轨迹上的第 2 个执行实例为切片标准精化错误怀疑范围,所得到的动态切片作为新的错误怀疑范围.

新一轮迭代自动生成的断点位于 *main* 函数中调用 *shell_sort*(第 22 行)的轨迹点处,包含的待检查的内容为传给 *shell_sort* 的第 2 个实际参数的值.该实际参数的预期值应与待排序的整数的数量相等,即为 2.经检查,该实参的值为 3,与预期值不等.由于该实参的值来源于程序执行时输入参数的个数(即 *argc* 的值),而外部输入总是正确,故该参数是导致输出结果失效的根源.

通过上述调试过程,用户也了解到该错误源的传播路径:*shell_sort* 的第 2 个实参通过翻转第 8 行的循环条件的第 2 次判断结果间接地修改了程序的输出.

3 实验分析

本节通过实验分析说明本文方法所生成的断点的质量与效率.第 3.1 节介绍实验程序选择以及实验平台的搭建方法.第 3.2 节分析 MDFS 断点的易判性和收敛性.第 3.3 节统计了断点自动生成的效率.

3.1 实验设计

本文选择由10个程序通过植入错误(其中,部分为已经被修正的真实错误)产生的179个缺陷程序版本作为实验用例.实验用例包括错误诊断领域常用 Siemens 测试包的 7 个程序:print-tokens,print-tokens2,replace,schedule,schedule-2,tcas 和 tot-info;以及 3 个实际程序:sed,grep 和 space.上述程序版本均摘自 Software-Artifact Infrastructure Repository(简称 SIR,http://sir.unl.edu/portal/index.php).程序 Sed 和 Grep 中植入的错误取自两个程序的 buglist(http://git.savannah.gnu.org/cgit);程序 space 中的错误来自该程序开发过程中记录的真实错误;其余均为人为植入错误.表 1 列出了这 10 个程序的名称、规模以及本文选用的缺陷版本(植入了错误的版本)数量与 SIR 提供的缺陷版本总量的比值、所选缺陷版本中包含的植入错误的数量、由上述错误引发的失效执行轨迹的平均规模以及对各程序功能的描述.其中,判断程序执行是否失效的方法是对比各缺陷版本的执行结果与正确版本在相同输入下的结果是否相同,如果不同即为失效;失效执行轨迹的平均规模即每个执行轨迹对应的动态依赖图上的结点的平均数量.

本文在选择实验用例的过程中剔除了 11 个缺陷版本,其中 10 个版本的执行结果与其所对应的正确版本在相同输入下的执行结果完全相同;另一个版本受限于本文实现的实验原型的健壮性,不能被正确分析.综上,本文使用的实验用例的缺陷版本数量共计 179 个,其中共包含 198 个已知错误.针对每个缺陷版本中的每个已知错误,本文随机选择 10 个触发该错误的测试输入(若触发该错误的测试输入少于 10 个,则无需选择),然后逐一将缺陷程序版本 P、输入 I 以及由该输入触发的程序失效 F 交由本文方法计算断点,并按照第 4 节所述的方法,根据断点选择情况进行人工调试.其中,F 代表失效轨迹上产生第 1 个错误输出的语句实例.

Table 1 Benchmark programs

表 1 实验用例

程序名	规模	选用缺陷版本/可用缺陷版本	错误数量	失效轨迹的平均规模	功能描述
print_token	472	6/7	4	3 773.00	词法分析器
print_token2	399	10/10	10	3 870.60	词法分析器
replace	512	31/32	31	12 619.50	模式替换
schedule	292	9/9	10	6 047.20	优先调度器
schedule2	301	9/10	10	8 046.75	优先调度器
tcas	141	41/41	48	221.78	高度分离
tot_info	440	23/23	23	6 265.49	信息测量
sed	14K	10/10	10	791.33	流编辑应用
grep	10K	3/3	3	9 822.00	文本搜索应用
space	6K	30/38	41	24 679.00	ADL 解释器

本文以 Valgrind[18]为基础搭建实验平台.Valgrind 是一套工具组件,以动态二进制插桩技术为基础,可服务于程序调试和剖析.本文借助 Valgrind 收集程序执行轨迹,在此基础上创建动态依赖图、计算最小调试边界.实验的硬件配置为主频 3.19GHz,内存 4GB 的 Intel(R) Xeon(TM)处理器.

3.2 断点质量分析

本节从断点检查开销和对调试收敛的作用两方面分析本文自动生成的断点的质量.第 3.2.1 节分析断点的易判性:通过对比本文方法与其他 3 种自动断点选择方法所提供的断点在数量和规模(即断点处需要检查的语句实例的数量)上的差异,说明本文方法生成的断点具有易判性.第 3.2.2 节从两个方面说明本文的 MDFS 断点对调试收敛的促进作用:1) 平均检查一个 MDFS 断点对调试范围的缩减比例;2) 对比使用本文方法与经典错误定位方法指导调试时的人工检查开销.

基于上述目标,本文额外实现了 3 种不基于最小调试边界的断点自动选择原型,分别用于模拟单步断点选择、二分断点选择和随机断点选择.上述 3 种原型均以动态依赖图为中间表示形式.首先,以产生失效表现的语句实例为切片标准对动态依赖图切片;然后,计算动态切片对应的动态依赖子图上各个结点到失效语句所对应的结点间路径的长度,下文称其为各结点的依赖步长.3 个原型分别维护一个初始值各异的步长索引变量(记作

I);设置断点时,选择依赖步长等于步长索引变量的所有结点对应的语句实例,作为断点处需要检查的对象的集合.上述语句实例的集合构成当前怀疑范围内一个与失效结点之间依赖步长为 *I* 的调试边界(但不一定是失效结点的最小调试边界).单步原型将步长索引变量的初始值设为 1;每完成一次断点选择,该索引变量的值增加 1.二分原型将步长索引变量的初始值设为动态依赖子图中依赖步长的最大值(记作 Max_*len*);每轮断点选择后,如果检查发现断点处的程序状态有误,即错误源在当前检查的调试边界之前触发,则将 *I* 的值置为⌈(Max_*len*–*I*)/2⌉;否则,将当前 *I* 的值设为新的 Max_*len*,并将新的 *I* 值置为⌈*I*/2⌉.随机原型在每轮断点选择之前为步长索引标量随机选择一个新值,选择范围为区间(0,Max_*len*).

实验过程中,对断点处程序状态的检查默认由人工完成.为降低人工检查的开销,本文采用一种辅助检查策略:利用正确的调试边界(记作 cMDFS)替代人工检查.正确的调试边界分为两种:保守的正确调试边界和激进的正确调试边界.所谓保守的正确调试边界即被以往迭代认定为正确的最小调试边界.获得激进的调试边界的方法是,从失效执行轨迹中抽取一个正确的程序状态(比如正确的输出、为真的断言等),计算其所对应的结点在当前错误怀疑范围内的最小调试边界.实验过程中,我们选取失效执行轨迹上产生最后一个正确输出的语句实例,将其最小调试边界作为 cMDFS.如果某个待检查的断点对应的 *mdfs* 与 cMDFS 重合,则认为该断点处待检查的语句实例都是符合设计预期的,即:下一轮迭代将仅检查失效轨迹上位于断点之后的部分.

表 2 统计了使用 MDFS 断点调试本文所选的 179 个缺陷程序的失效执行轨迹的总体效果,包括:需要设置的断点数量(#Bkpt 列);调试过程中需要检查的语句实例数量(#Inst 列);以及为定位到错误源,用户需要检查的语句数量占源程序中语句数量的比例(Reduce%列).其中,Range 列代表断点数量的波动范围,Aver 列代表平均值.Reduce%列中比例为 1 的情况,代表未定位到错误源.出现该情况的原因分为两种:1) 注入错误为变量类型声明错误.如,将双精度浮点变量误声明为单精度浮点类型.由于本文调试流程的初始错误怀疑范围中不包括变量声明语句,故对此类错误,无法收敛到错误源;2) 失效描述不完整.由于实验过程中仅选择产生第 1 个错误输出的语句实例对应的动态依赖图结点作为失效描述,以该结点为切片标准生成的初始错误怀疑范围可能未包含完整的错误源.

Table 2 Overall results of the MDFS-based debugging process

表 2 MDFS 断点的总体效果

程序名	#Bkpt		#Inst		Reduce (%)	
	Range	Average	Range	Average	Range	Average
print_token	1~3	2.25	1~3	2.25	0.21~1	33.48
print_token2	1~3	2.40	3~6	4.10	0.25~0.51	0.33
replace	1~5	3.40	3~7	4.90	0.20~0.78	0.31
schedule	1~5	2.78	4~6	5.20	0.34~1.03	0.65
schedule2	2~4	2.80	4~6	4.91	0.33~1.00	0.80
tcas	1~4	1.27	3~5	3.68	0.71~1	2.95
tot_info	1~3	2.12	3~8	3.63	0.23~1	4.51
sed	2~6	3.10	3~8	4.30	0.01~0.02	0.01
grep	3~6	4.00	7~11	8.33	0.03~0.06	0.04
space	2~5	3.96	3~7	6.01	0.01~1	7.01

3.2.1 易判性分析

本节分析 MDFS 断点的易判性.

表 3 分析了本文提出的断点生成方法(下文简称 MDFS 方法)与额外实现的 3 种断点选择方法:单步、二分以及随机方法,为定位到错误根源,需要检查的断点数量(#Bkpt 列)以及断点的规模(#Inst 列).考虑到基于随机方法生成的断点具有不确定性,难以确保调试收敛,故将基于随机断点生成方法进行调试时的断点上限设为 10 个.选择该值作为上界的原因是根据表 2 的统计,基于 MDFS 断点的调试过程所需要的断点数量均未超过 10 个.

由表 3 可知:采用基于单步断点的调试方法,需要设置的断点数量以及断点处需要检查的语句实例的数量均显著高于基于 MDFS 断点的调试;基于二分断点选择的调试方法,需要设置的断点数量少于单步调试,但仍高于基于 MDFS 的调试;随机方法由于限定了断点数量上限,故不能确保调试收敛.表 3 的最后一列(#located 列)

统计了采用随机断点的调试过程可以定位到的错误的数量:共计 42 个,仅占错误总数的 21.21%.综上所述,采用 MDFS 断点的调试在定位相同数量的错误的前提下,无论在断点数量还是断点规模上均优于采用单步断点和二分断点的调试;而与随机断点相比,在断点数量相当的前提下,采用 MDFS 断点的调试不仅需要检查的语句实例数量更少(即断点规模更小),而且可以定位到更多的错误.

Table 3 Comparison with step-by-step, binary and random breakpoint generating approaches

表 3 与单步、二分、随机断点选择方法的对比

程序名	MDFS 方法		单步方法		二分方法		随机方法	
	#Bkpt	#Inst	#Bkpt	#Inst	#Bkpt	#Inst	#Inst	#located
print_token	1~3	1~3	21~43	26~268	11~15	33~182	15~318	1
print_token2	1~3	3~6	33~36	41~317	4~14	6~22	6~194	4
replace	1~5	3~7	9~52	19~177	7~16	43~81	16~84	7
schedule	1~5	4~6	11~56	2~163	8~23	23~57	6~20	0
schedule2	2~4	4~6	11~68	2~150	8~23	31~57	6~22	0
tcas	1~4	3~5	4~23	5~23	3~9	8~13	10~15	15
tot_info	1~3	3~8	6~42	10~42	5~8	10~14	14~25	7
sed	2~6	3~8	8~10	20~47	6~10	22~34	1-19	2
grep	3~6	7~11	5-35	5~42	8~11	8~47	7~31	0
space	3~5	3~7	2~92	2~97	6~11	14~27	8~21	6

3.2.2 收敛性分析

本节分析基于 MDFS 生成的断点对调试收敛的帮助.

表 4 统计了平均检查一个 MDFS 断点,错误怀疑范围的缩减比例.由此可知,基于 MDFS 生成的断点平均可以使错误怀疑范围缩减 37.98%~58.91%.

Table 4 Ratio of trace shortening by using MDFS-based breakpoints

表 4 断点选择对错误怀疑范围收敛的影响

	print_token	print_token2	replace	schedule	schedule2	tcas	tot_info	sed	grep	space
Reduce (%)	45.38	47.28	47.84	35.75	37.98	54.82	47.11	58.91	49.23	50.29

为进一步说明基于 MDFS 的断点选择方法对调试收敛的帮助,本节下面分析了用户在使用基于 MDFS 生成的断点对第 3.1 节所选程序的失效执行轨迹进行调试时,需要检查的程序比例以及能够最终定位到的错误源数量;并与采用经典错误定位技术指导的调试过程中需由人工检查的程序比例进行对比.

经典错误定位方法通常为程序中的每个单元(如源程序语句、语句执行实例、程序状态等)计算一个怀疑分值(suspicious score),用于代表该单元可能为错误源的概率.方法的输出结果通常是一组按照怀疑分值降序排列的程序单元的序列(下文简称怀疑序列),其中,怀疑分值相同的单元共享同一个序号.程序员将按照怀疑序列逐一检查程序中的各个单元,直到确定错误根源所在的单元.

T-score[9]是错误定位领域的一种常用的评价标准.该标准通过统计错误定位方法在要求用户检查不同比例的源程序代码时可以定位到的错误占所有错误的比例,并分析所有可定位到的错误在不同代码检查比例区间内的分布情况评判方法的优劣.检查相同比例的代码可以定位到的错误的比例越高,说明方法对调试的辅助效果越显著.经典错误定位方法在统计需要由人工检查的语句比例时默认遵循“完美的错误理解(perfect bug understanding)”[19]假设,即默认程序员在按照怀疑分值的降序依次检查程序各个单元时,有能力识别出当前所走查的程序单元是否为错误源.根据上述假设,程序员为识别错误源需要检查的语句比例即怀疑序列中位于错误源语句之前的所有语句(包括错误源语句本身以及所有序号与之相同的语句)的数量与源程序中语句数量的比值.

表 5 分别统计了采用本文生成的断点与使用经典错误定位方法提供的怀疑序列调试 Siemens 测试包中注入的错误时,为定位到错误源语句,需要用户检查的程序代码规模的比例.统计采用 T-score 标准.代表性的错误定位方法,如 PPDG[1],CBI[6],SOBER[20]等,均以程序依赖图(简称 PDG)为基础评价错误定位的开销,度量公式

形如

$$S=1-\frac{|V_{examined}|}{|PDG|},$$

其中,$|V_{examined}|$代表从方法所提示的可疑语句对应的程序依赖图结点开始,按广度优先顺序遍历,直到到达错误源对应的结点时,所有访问过的结点的数量.该计算方法也称为基于 PDG 的 T-score 计算方法.Tarantula[9]则直接根据错误源语句在怀疑序列中的名次(ranking)计算 T-score,即统计从怀疑序列列首起,按名次由高到低遍历被执行的语句,直到到达错误源语句时,所有访问过的语句数量;然后计算未被访问的语句数量占所有被执行的语句数量的比例.该度量方式也称为基于名次的 T-score 度量标准.在使用 MDFS 断点进行调试的过程中,需要由人工检查的语句数量即各断点处提示应检查的语句实例的总数.为了兼顾与采用了不同 T-scpre 计算方法的多种经典错误定位技术间的比较,本文在为 MDFS 方法统计 T-score 时,如果遇到被检查的语句实例中不包含错误源的情况,则保守地认为用户需要检查完整的执行轨迹,即 S=1.表 5 用 MDFS 代表采用本文方法生成断点的调试流程,分别计算了当检查的代码比例少于 1%和 10%时,可以定位到的错误比例.表 5 还列出了采用 5 种经典错误定位方法:Tarantula[9],CT[5],PPDG[1],CBI[6]和 SOBER[20]生成的怀疑序列调试 Siemens 测试包时,检查怀疑序列中前 1%和 10%的语句或按广度优先序分别遍历程序依赖图中 1%和 10%的结点时,分别能够定位的错误比例.其中,PPDG(best)和 PPDG(worst)分别代表检查指定比例代码的前提下,基于 PPDG 的错误定位方法可以定位的错误的最佳比例和最差比例.由表 5 所列数据可知,采用本文提供的断点指导调试,可以在检查少于 1%和 10%的代码比例时分别定位到 42.96%和 96.50%的错误,均优于以经典错误定位方法为指导的调试所能定位到的错误比例.

Table 5 Comparison with the state-of-art fault-localization approaches (%)

表 5 与经典错误定位方法的对比(%)

T-score	MDFS	CT	Tarantula	PPDG (worst)	PPDG (best)	CBI	SOBER
<1	42.96	4.65	13.93	17.74	41.94	7.69	8.46
<10	96.50	26.56	55.19	44.16	73.39	40.63	53.13

3.3 断点生成效率

本节分析计算 MDFS 断点的开销.表 6 统计了本文使用的 10 个实验程序平均定位一个错误所需要的断点生成时间.对交互式调试而言,表 6 所示的时间开销是可接受的.

Table 6 Average time cost for breakpoint generation (s)

表 6 平均断点生成时间 (秒)

	print_token	print_token2	replace	schedule	schedule2	tcas	tot_info	sed	grep	space
时间	1.19	1.51	1.04	2.19	2.62	0.08	7.05	1.48	6.94	2.60

4 相关工作

时至今日,调试仍然占据软件开发过程中近 70%的时间,通过设置断点并检查其上程序状态的交互式调试方法依旧是实际工作中最常用的定位错误的手段.为了提高断点质量,研究人员提出了一系列断点推荐方法.Chern[21]认为,当前调试器所提供的静态断点(不带条件的断点)设置方式往往不能精确描述出程序员感兴趣的轨迹点,动态断点(带条件的断点)虽然可以描述断点的触发条件,但需要由人工书写.为此,他提供了一种描述断点触发条件的语句以及触发条件的自动生成技术,并能够随调试的进行持续精化断点的触发条件.Ko[11]从经验数据中发现:程序员对错误代码的猜测几乎总是错误的,并且几乎所有调试器都不支持诸如“为什么某些事件没有发生”的询问.为克服上述问题,他开发了调试辅助工具 Whyline.该工具以依赖分析为基础,预先准备一系列问题供调试人员选择,并根据选择做答.断点推荐工具 VIDA 以 Tarantula 生成的怀疑序列为依据,为用户推荐断点;并根据用户的选择和检查结论,更新怀疑序列.BPGen[12]则采用 NN[10]错误定位技术和 delta 调试从错误传播链中分离出少量关键位置,供用户从中选取断点.目前,该领域的研究存在 3 个问题:1) 断点的选择需要人工参

与,没有从根本上减轻调试人员的负担;2) 缺乏评价断点质量的标准;3) 断点的生成效率取决于底层错误定位方法的效率.本文提出的自动断点生成技术以最小调试边界为基础,使断点具有与失效相关、易断、促进收敛等优势.

与断点推荐技术相比,自动错误定位技术能够缩小调试人员需要分析的程序范围,并提示其中各语句可能为错误源的概率.代表性错误定位方法有:基于统计的错误定位方法、基于状态对比的错误定位方法、基于程序切片的方法等.基于统计的错误定位技术是迄今为止公认的错误定位精度最高的一类技术.该类技术以程序动态插桩和概率统计为基础,为每个程序单元(通常是程序语句或程序语句间的依赖关系),计算一个怀疑分值;怀疑分值越高的程序单元,成为错误源的概率也越大.代表性的工具,如 Tarantula[9].后续的研究人员在此基础上继续改进统计模型以及怀疑分值的计算公式.基于状态的错误定位技术通常需要两组测试输入,其中,一组是能够触发程序中的错误并导致程序失效的测试输入,另一组是使该程序能够正确执行的测试输入.通过对比两组输入产生的执行轨迹间的差异,发掘错误根源.典型的方法,如 CT[5],NN 等.上述两类方法目前所面临的共同障碍是错误定位精度对程序测试输入的敏感性[19].虽然目前出现了不少缓解输入敏感的策略[6,7,22,23],但并未根本上解决该问题.基于程序切片的错误定位方法[24−26]虽然是错误定位和错误理解领域的一类基础性方法,能够有效地缩小错误调试范围,但自动定位错误源的能力较弱.

5 结束语

本文提出了一种新的自动断点生成技术.该技术以本文提出的最小调试边界的概念为基础,识别程序失效执行轨迹上满足错误隔离性和状态最小化的轨迹点集合以及其所对应的语句实例作为断点设置的依据.实验结果表明:采用本文方法生成的断点具有检查开销低、加速调试收敛等优势;以其为基础的调试流程与采用经典错误定位方法指导的调试过程相比,能够以更低的检查开销获得更好的错误定位比例.

References:

[1] Baah KG, Podgurski A, Harrold JM. The probabilistic program dependence graph and its application to fault diagnosis. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2008. 189−200. [doi: 10.1145/1390630.1390654]

[2] Baah KG, Podgurski A, Harrold JM. Causal inference for statistical fault localization. In: Proc. of the 19th Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2010. 73−84. [doi: 10.1145/1831708.1831717]

[3] Baah KG, Podgurski A, Harrold JM. Mitigating the confounding effects of program dependences for effective fault localization. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. New York: ACM Press, 2011. 146−156. [doi: 10.1145/2025113.2025136]

[4] Chandra S, Torlak E, Barman S, Bodik R. Angelic debugging. In: Proc. of the 33rd Int'l Conf. on Software Engineering. New York: ACM Press, 2011. 121−130. [doi: 10.1145/1985793.1985811]

[5] Cleve H, Zeller A. Locating causes of program failures. In: Proc. of the 27th Int'l Conf. on Software Engineering. New York: ACM Press, 2005. 342−351. [doi: 10.1145/1062455.1062522]

[6] Liblit B, Naik M, Zheng XA, Aiken A, Jordan IM. Scalable statistical bug isolation. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2005. 15−26. [doi: 10.1145/1065010.1065014]

[7] Nainar AP, Liblit B. Adaptive bug isolation. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering, Vol.1. New York: ACM Press, 2010. 255−264. [doi: 10.1145/1806799.1806839]

[8] Jose M, Majumdar R. Cause clue clauses: Error localization using maximum satisfiability. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2011. 437−446. [doi: 10.1145/1993498.1993550]

[9] Jones AJ, Harrold JM. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2005. 273−282. [doi: 10.1145/1101908.1101949]

[10] Renieris M, Reiss PS. Fault localization with nearest neighbor queries. In: Proc. of the 18th IEEE Int'l Conf. on Automated Software Engineering (ASE 2003). Washington: IEEE Computer Society, 2003. 30−39. [doi: 10.1109/ASE.2003.1240292]

[11] Ko JA, Myers AB. Debugging reinvented: Asking and answering why and why not questions about program behavior. In: Proc. of the 30th Int'l Conf. on Software Engineering. New York: ACM Press, 2008. 301−310. [doi: 10.1145/1368088.1368130]

[12] Zhang C, Yan DC, Zhao JJ, Chen YT, Yang SQ. BPGen: An automated breakpoint generator for debugging. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering, Vol.2. New York: ACM Press, 2010. 271−274. [doi: 10.1145/1810295.1810351]

[13] Hao D, Zhang LM, Zhang L, Sun JS, Mei H. VIDA: Visual interactive debugging. In: Proc. of the 31st Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2009. 583−586. [doi: 10.1109/ICSE.2009.5070561]

[14] Agrawal H, Horgan RJ. Dynamic program slicing. In: Proc. of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1990. 246−256. [doi: 10.1145/93542.93576]

[15] Zhang XY, Gupta R, Zhang Y. Precise dynamic slicing algorithms. In: Proc. of the 25th ACM/IEEE Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2003. 319−329.

[16] Cormen HT, Stein C, Rivest LR, Leiserson EC. Introduction to Algorithms. 2nd ed., Cambridge: MIT Press, 2001.

[17] Chekuri SC, Goldberg VA, Karger RD, Levine SM, Stein C. Experimental study of minimum cut algorithms. In: Proc. of the 8th Annual ACM-SIAM Symp. on Discrete Algorithms. Philadelphia: Society for Industrial and Applied Mathematics, 1997. 324−333.

[18] Valgrind. http://valgrind.org/

[19] Wong EW, Debroy V. A survey of software fault localization. Technique Report, UTDCS-45-09, Department of Computer Science, The University of Texas at Dallas, 2009.

[20] Liu C, Yan XF, Fei L, Han JW, Midkiff PS. SOBER: Statistical model-based bug localization. In: Proc. of the 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC/FSE-13). New York: ACM Press, 2005. 286−295. [doi: 10.1145/1081706.1081753]

[21] Chern R, Volder DK. Debugging with control-flow breakpoints. In: Proc. of the 6th Int'l Conf. on Aspect-Oriented Software Development. New York: ACM Press, 2007. 96−106. [doi: 10.1145/1218563.1218575]

[22] Artzi S, Dolby J, Tip F, Pistoia M. Directed test generation for effective fault localization. In: Proc. of the 19th Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2010. 49−60. [doi: 10.1145/1831708.1831715]

[23] Sumner NW, Bao T, Zhang XY. Selecting peers for execution comparison. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 309−319. [doi: 10.1145/2001420.2001458]

[24] Weiser M. Program slicing. In: Proc. of the 5th Int'l Conf. on Software Engineering (ICSE'81). Piscataway: IEEE Press, 1981. 439−449.

[25] DeMillo RA, Pan H, Spafford EH. Critical slicing for software fault localization. In: Proc. of the 5th Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 1996. 121−134. [doi: 10.1145/229000.226310]

[26] Zhang XY, Gupta N, Gupta R. Pruning dynamic slices with confidence. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2006. 169−180. [doi: 10.1145/1133981.1134002]

李丰(1985－),女,福建福安人,博士生,CCF 学生会员,主要研究领域为程序分析,错误诊断.
E-mail: lifeng2005@ict.ac.cn

李龙(1988－),男,研究实习员,CCF 学生会员,主要研究领域为程序分析技术.
E-mail: lilong@ict.ac.cn

霍玮(1981－),男,博士,助理研究员,CCF 会员,主要研究领域为编译技术,程序错误检测.
E-mail: huowei@ict.ac.cn

袁璐洁(1979－),女,博士生,讲师,主要研究领域为编译技术,程序错误检测.
E-mail: zhonglujie@ict.ac.cn

陈聪明(1985－),男,博士生,主要研究领域为指针分析,并行程序错误检测.
E-mail: chencongming@ict.ac.cn

冯晓兵(1969－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为先进编译技术及相关工具环境.
E-mail: fxb@ict.ac.cn