基于路径分析的死循环检测

阮 辉[1),2)] 严 俊[1)] 张 健[1)]

[1)](中国科学院软件研究所计算机科学国家重点实验室 北京 100190)

[2)](中国科学院研究生院 北京 100190)

摘 要 提出了一种自动检测 C 语言程序中是否含有死循环的方法.该方法基于程序分析技术,包括循环展开和路径可行性分析技术.该方法首先通过遍历控制流图生成待查循环的检验路径;之后通过分析检验路径的可行性以及路径之间的联系,判断这些路径是否符合死循环模式.在此方法基础上实现了原型工具 LoopAnalyzer,并对一组基准程序进行测试.实验结果表明此工具能有效地检测出 C 语言程序中的死循环,并且准确率较高.

关键词 死循环;循环展开;路径可行性;测试数据

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2009.01750

Infinite Loop Detection Based on Path Analysis

RUAN Hui[1),2)] YAN Jun[1)] ZHANG Jian[1)]

[1)](*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190)

[2)](*Graduate University of Chinese Academy of Sciences, Beijing* 100190)

Abstract This paper proposes a method for detecting infinite loops in C programs. The method is based on some static analysis techniques, including loop unwinding and path feasibility checking. By traversing the program's flow graph, it firstly generates a set of diagnosis paths for a specified loop. Then it analyzes all the paths according to their test data and relationship, to decide whether there is a subset of the paths matching some non-termination pattern. A prototype tool called LoopAnalyzer is developed to illustrate the feasibility of this method. The experimental results on some benchmark programs show that the tool can detect infinite loops effectively and accurately.

Keywords infinite loop; loop unwinding; path feasibility; test data

1 引 言

在现代程序设计中,几乎所有的程序都会含有循环.然而,即使是简单的循环程序,也很容易出错.循环中的很多错误往往需要执行多次或者在某些特定的情况下才能被发现,检测这些错误的代价很高.死循环错误是软件的一种致命的错误,简单地通过有限的测试用例很难发现.自动地检查出这种错误能够有效地提高软件开发的效率和质量.

死循环的检测,或者程序的终止性判定,一直是软件研究中的重要问题.从可计算性的角度来讲,死

收稿日期:2009-04-17;最终修改稿收到日期:2009-08-06.本课题得到国家自然科学基金(60633010)和国家"八六三"高技术研究发展计划项目基金(2009AA01Z148)资助.阮 辉,男,1984 年生,硕士研究生,研究方向包括程序分析与软件测试自动化.严 俊,男,1980 年生,博士,助理研究员,研究方向包括约束求解、程序分析与软件测试.张 健,男,1969 年生,博士,研究员,博士生导师,主要研究领域包括自动推理、形式化方法、约束求解与软件测试等. E-mail: zj@ios.ac.cn; jian_zhang@acm.org.

循环的检测可归结为停机问题,因而是一个不可计算的问题.也就是说,没有一种通用的算法能够精确地判定任意一个程序是否含有死循环.但是,如果对程序进行一些限制,我们还是可以通过分析程序的代码,在一定的程度上自动地检测出程序中的死循环.

目前,大部分的关于死循环检测的研究工作集中在对于抽象程序进行理论分析,针对实际代码的实用的自动化技术研究比较少.在另一方面,程序分析技术是形式化方法和软件工程领域的一个热点,被广泛应用于程序测试和正确性验证[1].本文利用程序分析技术在死循环检测这一问题上做了一些探索,将死循环当成一种错误模式,并自动地寻找这种错误模式.直观来讲,当循环的循环条件不断被满足,循环体不断被执行时,就可以认为程序中出现了死循环.基于这样的思想,本文采用基于程序分析的办法,通过分析程序控制流图中具有相同测试数据的路径,来检测程序中是否包含死循环.即静态地分析程序的源代码,判断是否存在一组测试数据,使得程序的循环体一直执行下去.

本文第 2 节首先介绍一些基本概念,包括控制流图、循环以及检验路径等;第 3 节给出死循环检测方法并进行分析;第 4 节通过一个具体的例子展示如何使用我们的方法检测出程序中的死循环,并给出一组基准程序的实验数据;最后介绍一些相关工作并对本文进行总结和展望.

2　基于路径的程序分析技术

在介绍死循环检测方法之前,本节给出基本的术语并介绍分析过程中需要用到的基于路径的程序分析技术.

2.1　基本概念

基于代码的分析和测试都需要对代码进行抽象,建立一种中间模型.控制流图(Control Flow Graph,CFG)是一种常用的中间模型.

定义 1.　控制流图.控制流图是以基本块(basic block)为节点的有向图.所谓基本块,是程序中具有唯一入口和唯一出口的语句序列,其中不包含条件转移语句.基本块中的语句在程序执行过程中要么全都执行,要么全都不执行.形式化地说,控制流图是一个有向图 $G=\langle N,E,s,f\rangle$,其中 N 是节点的集合,表示程序中的基本块;$E\subseteq N\times N$ 表示边的集合,如果程序中有从 B_i 转向 B_j 的语句,则在 G 中有一条从 B_i 到 B_j 的有向边,记作 $e=(B_i,B_j)\in E$;s 和 f 分别表示控制流图的起点和终点,对应于程序中的入口基本块和出口基本块.

控制流图中的路径(path)描述了程序的一次具体执行过程.

定义 2.　路径.路径指控制流图中的一个执行序列 $p=\langle n_1,n_2,\cdots,n_m\rangle$,其中 m 是路径的长度,$(n_i,n_{i+1})\in E(1\leq i<m)$.称 $n_1=s$ 的路径为部分路径(partial path).如果 $n_1=s$ 并且 $n_m=f$,我们称 p 是程序的一条完整路径(complete path).在下文中如无特殊说明,我们所说的程序路径都指部分路径.(部分)路径 p 是可行的(feasible),当且仅当存在一组输入数据使得程序能沿这条路径执行.此时称路径 p 是可行路径(feasible path),否则称为不可行路径(infeasible path)[2].

几乎所有程序中都存在着循环,程序中的循环可以通过控制流图中的有向环来描述.

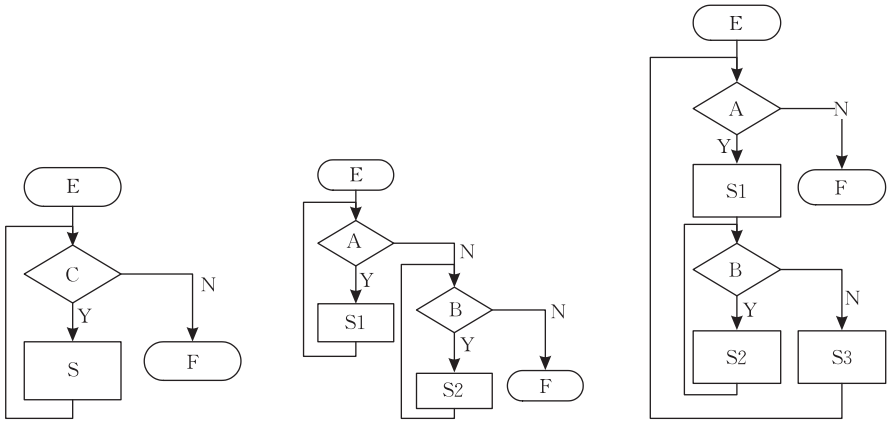
定义 3.　环.控制流图中的边 $e=(B_j,B_i)$ 称为回边(back edge),当且仅当从起点 s 到 B_j 的任何路径都经过 B_i,同时称 B_i 为 B_j 的必经点[3].控制流图中的回边 (B_j,B_i) 定义了一个自然环,该环由 B_i 和所有不经过 B_i 可到达 B_j 的节点集合以及连接这些节点的边所组成的子图[3].控制流图中的环,对应于程序中的一个循环.通常用回边 (B_j,B_i) 来标识这个环,或者在不出现二义性的情况下,也可以直接使用 B_i 来标识这个环.

结构化程序中的循环有 3 种基本形式:简单循环(simple loop)、串接循环(concatenated loops)和嵌套循环(nested loops)[4].其控制流图的组成形式如图 1(a)、(b)和(c)所示.注意图 1 中的 S、$S1$、$S2$ 和 $S3$ 都是一个抽象的整体,其中可能含有较为复杂的结构,并不仅仅是定义 1 中的基本块.

定义 4.　检验路径.在由回边 (B_j,B_i) 确定的环中,其检验路径(diagnosis path)指从 s 经过 B_j 到达 B_i 的一条路径.简单来说,检验路径是一条部分路径,其结束节点是程序中的一个循环条件判断.

为了对上述概念有一个清晰的认识,下面给出一个具体的例子.

例 1.　图 2 为一段 C 语言代码,函数计算参数 m 和 n 的最大公约数并返回.图 3 为函数 gcd 对应的控制流图,其中 s 和 f 表示开始和结束节点,边(6,1)表示回边,虚线表示条件为假.



(a) 简单循环 (b) 串接循环 (c) 嵌套循环

图 1 3 种循环的形式

```
int gcd(int m, int n)
{
    while (m !=n){      /*1*/
      if (m>n){         /*2*/
          m=m-n;        /*3*/
      } else {
          n=n-m;        /*4*/
      }                 /*6*/
    }
    return m;           /*5*/
}
```

图 2 一段示例代码

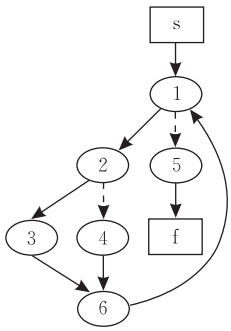


图 3 函数 gcd 的控制流图

由图 3 可知,函数 gcd 中有一个循环,由节点 1、2、3、4 和 6 构成. 路径 $p_1=\langle s,1,2,3,6,1\rangle$ 和 $p_2=\langle s,1,2,4,6,1\rangle$ 是该循环的两条检验路径. 它们的语句形式如图 4(a)和(b)所示,图中"@"表示后面是一个条件. 这两条路径都是可行的,而图 4(c)给出了一条不可行路径的例子. 可以采用符号执行的方法从后往前替换[2]得到路径条件. 初始路径条件 $P=\{\text{true}\}$;根据 s4,得到 $P=\{j<4\}$;根据 s3,替换 P 中的 j 得到 $P=\{i+1<4\}$;根据 s2,再替换 P 中的 i 得到 $P=\{i+2+1<4\}$,化简得到 $P=\{i<1\}$;根据 s1,添加条件到 P 得到 $P=\{i<1\wedge i>3\}$. 显然路径条件 P 无法满足,因此路径 p_3 是不可行的.

```
int m, n;
{
    @(m !=n);
    @(m>n);
    m=m-n;
    @(m!=n);
}
```

(a) 可行路径 p_1

```
int m, n;
{
    @ (m!=n);
    @ !(m>n);
    n=n-m;
    @ (m!=n);
}
```

(b) 可行路径 p_2

```
int i, j;
{
    @ (i>3);    /*s1*/
    i=i+2;      /*s2*/
    j=i+1;      /*s3*/
    @ (j<4);    /*s4*/
}
```

(c) 不可行路径 p_3

图 4 路径示例

定义 5. 死循环模式. 环 loop 出现死循环时,其检验路径的组成形式.

对于简单循环、串接循环和嵌套循环,它们具有如表 1 所示的死循环模式(其中,符号"+"表示 0 次或有限次重复,"*"表示无限次重复). 比如,图 1(a)所示的简单循环如果出现无限循环,那么其循环条件 C 应该不断被满足,循环体 S 也不断被执行,

表 1 程序中的死循环模式

循环形式	死循环模式
简单循环(如图 1(a)所示)	$\langle E,C,(S,C)^*\rangle$
串接循环(如图 1(b)所示)	$\langle E,A,(S1,A)^*\rangle$ $\langle E,(A,S1)+,\neg A,B,(S2,B)^*\rangle$
嵌套循环(如图 1(c)所示)	$\langle E,(A,S1,(B,S2)+,\neg B,S3)+,$ $A,S1,B,(S2,B)^*\rangle$ $\langle E,A,(S1,(B,S2)+,\neg B,S3,A)^*\rangle$

从而其检验路径应该符合模式〈E,C,(S,C)*〉.注意表 1 中的 S 是一个抽象的整体,而不是特指某一个特定的基本块或路径.如果图 1(a)的循环体中含有分支,那么任何一个分支都可以用 S 表示.

例 2.　对于图 1 中的函数 gcd,如果出现死循环,只可能是基本块 1 表示的循环条件一直满足,且基本块 1 之后的循环体不断执行下去,实际的循环模式可以记为〈s,1,(1B,1)*〉(其中 1B 表示基本块 1 之后的循环体).

2.2　路径可行性判断

在基于路径的程序分析中,通常会遇到很多不可行路径,不可行路径的存在极大地降低了程序分析的效率.如何判断程序中的路径是否可行,即路径的可行性判断问题,是基于路径的程序分析的一个重要的问题.在一般情况下,路径可行性是不可判定的.但是我们可以通过限制程序中的变量和条件表达式来部分地求解这个问题.工具 PAT[2] 是一个针对 C 语言子集的路径可行性判定工具,使用了符号执行和约束求解技术,是一种较为精确的判断方法.它能处理的数据类型包括整型、布尔型、枚举类型和数组类型.EPAT 是对该工具的一个扩展,使其能处理指针和结构类型[5].目前,EPAT 能处理的表达式形式包括布尔逻辑以及线性数值约束(如图 4 所示的 3 条路径).如果路径是可行的,EPAT 还能给出一组输入变量的初始值,使得程序在该初始值下能沿着这条路径执行.关于 EPAT 的详细讨论可以参考文献[2,5].

3　死循环检测方法

直观来讲,死循环是指程序运行过程中,循环的条件不断被满足,循环体不断被执行.基于这样的观察,本文采用静态分析的办法来检测程序中的死循环,从查找错误的角度出发,将程序中出现死循环当成是一种错误模式(bug pattern),静态地寻找包含这种错误模式的部分路径,从而从大规模的路径集合中挖掘出程序是否有死循环的信息.

基于静态分析的死循环检测方法可以简单描述如下:先采用循环展开策略,针对这个循环产生出一系列检验路径;进一步地,通过分析检验路径的可行性以及路径之间的联系,检查这些路径是否符合死循环模式,从而给出目标程序中是否含有死循环的结论.

该方法的流程如图 5 所示,大体上可以分为 3 步,前端处理、路径生成和死循环识别:

1. 前端处理.解析源程序,生成控制流图并找出程序中的循环;

2. 路径生成.根据一定的策略,生成一系列可行的检验路径;

3. 死循环识别.分析检验路径,判断它们是否满足死循环模式或者循环执行次数是否超过某一个给定的阈值等.

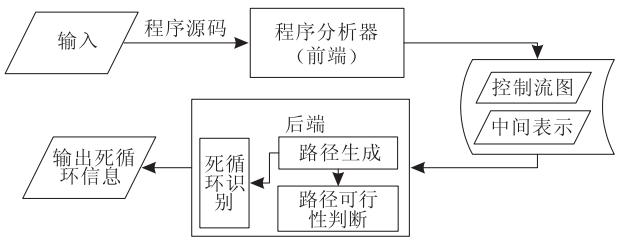


图 5　基于静态分析的循环检测

接下来的 3 个小节将具体说明这 3 步采用的方法.

3.1　前端处理

静态分析器前端工作与编译器前端类似,主要完成对源程序的语法和语义检查,同时生成中间表示,在此基础上建立控制流图.这部分中,我们使用了 gcc2.95.3 语法源文件(parse.y),通过重写语义动作,并利用 Bison① 工具自动地生成语法分析器,通过一遍扫描的办法构造出抽象语法树.之后通过遍历抽象语法树转换成中间表示,最后将中间表示划分为基本块并建立控制流图.我们之前的论文[6]中也介绍了这部分工作,并在此基础上生成测试数据.

3.2　路径生成

路径生成主要是根据控制流图生成一组检验路径,其本质上就是遍历控制流图的过程.在这部分中,有以下几个问题需要考虑:

(1)循环执行次数.由于控制流图中有环,可能会含有无限条路径,因此需要对循环的执行次数进行限定.对于某些特殊的循环可以从程序源码推测其执行次数,然而对于大多数循环,其执行次数都是很难估算的.为了减少分析的代价,避免循环展开的过程无限进行下去,需要限制循环体最大执行次数.在实现时,工具 LoopAnalyzer 采用命令行参数,由用户指定循环体的最大执行次数.

(2)路径生成策略.图的遍历算法主要有宽度优先(BFS)和深度优先(DFS)两种.我们实现了这

① GNU Bison. http://www.gnu.org/software/bison/

两种算法，并通过实验比较其优缺点. 我们发现当循环嵌套次数比较多的时候，宽度优先算法需要保存很多的内部节点，占用大量内存. 工具 LoopAnalyzer 最终选用了深度优先策略.

(3) 不可行路径删除. 为了提高路径生成算法的效率，可以在路径生成过程中检查路径的可行性. 具体做法是当遇到程序中的分支语句时，根据分支条件判断当前路径是否是可行的，只有当前路径是可行时才进一步扩展.

算法 GenPath(*loop*,*maxTimes*)根据给定的循环 *loop* 和循环最大执行次数 *maxTimes*，生成所有可行的检验路径，并返回检验路径的集合 *S*. 其中 *initNode* 表示控制流图的入口节点，集合 *SS* 存放中间数据. 当采用宽度优先和深度优先遍历算法时，*SS* 分别采用队列和堆栈来实现. *SS* 的元素是一个三元组(*bbIndex*, *path*, *times*)，其中，*bbIndex* 表示当前节点编号；*path* 是控制流图节点的数组，表示一条路径；*times* 是长度为 *n* 的数组，表示路径 *path* 中节点的访问次数，*n* 为控制流图的节点数目.

算法 1. GenPath(*loop*,*maxTimes*).

```
1. SS←{(initNode,[initNode],[1,0,0,…,0])};
2. S←∅;
3. while (!isEmpty(SS)){
   /* 从集合 SS 中取出一项并从 SS 中删除 */
4.    (bbIndex,path,times)←extractNode(SS);
5.    if (isBranch(bbIndex)&&!isFeasible(path))
6.       continue;
         /* 是分支节点并且不可行，丢弃该部分路径 */
7.    else if (isDiagnosisPath(path,loop))
8.       S←S∪{path};
         /* 是检验路径，添加到路径集合中 */
9.    if (isLoopHead(bbIndex)&&
         times[bbIndex]>maxTimes)
10.      continue;
         /* 是循环开始节点且执行次数大于阈值，
            丢弃 */
11.   newNodes←extendAll(path);
      /* 扩展路径，并添加到集合 SS 中 */
12.   SS←SS∪newNodes;
13. }
14. return S;
    /* 返回路径集合 S */
```

3.3 死循环识别

在生成路径之后，需要对这些路径进行分析，判断其是否满足某一个死循环模式. 一般来说，符合同一个死循环模式的部分路径，满足其路径条件的测试数据空间，具有一个公共非空子空间. 例如对于例 1 中的函数 gcd，所有符合模式〈s,1,(1B,1)*〉的路径，其路径条件的测试数据空间均包含子空间 $\{m\neq n\wedge(m\leq 0\vee n\leq 0)\}$. 在此条件下，程序将不会终止.

但是，自动化地寻找这一公共子空间是很困难的. 为了解决这一问题，我们采用一个近似的方法，即如果一组部分路径具有一个相同的测试数据(可以用路径分析工具求解得到)，则它们可能具有相同的死循环模式. 这一过程可以简要地描述为以下两步：

1. 划分路径集合. 对路径集合 *S* 按照不同测试数据进行分类，得到集合 *S* 的一个划分 $C=\{S_i\}$. 其中 S_i 满足 $\cup S_i=S$ 并且 $\forall i\neq j(S_i\cap S_j=\varnothing)$，同时每个集合 S_i 中的路径都具有相同的测试数据.

2. 循环模式检查. 对于 *C* 中的集合 S_i，判断其中的路径是否可以归约为一个死循环模式. 如果存在 *C* 中某个集合 S_i 中的路径可以归约为死循环模式，则返回 true，否则返回 false.

算法 CheckPattern(S_i,*loop*)根据给定的路径集合 S_i 和循环 *loop*，判断该路径集合是否符合 *loop* 的死循环模式，如果是则返回 true，否则返回 false. 函数 *ReducePath* 的功能是将路径的表示形式进行抽象，比如将图 4 中(a)和(b)所示的检验路径 p_1 和 p_2 抽象成为〈s,1,1B,1〉，因为路径 p_1 和 p_2 中，循环体都只执行一次. 函数 *RemoveSame*(*pattern*)去掉由于路径抽象而导致的重复路径(如图 4 中路径 p_1 和 p_2 都简化成同一条路径). 函数 *isLoopPattern* 判断抽象后的路径是否符合如表 1 所示的死循环模式.

算法 2. CheckPattern(S_i,*loop*).

```
1. m←|Si|;
2. for (j=0; j<m; ++j)
3.    pattern[j]←ReducePath(Si[j], loop);
      /* 抽象路径的表示形式 */
4. RemoveSame(pattern);
   /* 去掉重复路径 */
5. if (isLoopPattern(pattern, loop))
   /* 判断是否符合死循环模式 */
6.    return true;
7. else
8.    return false;
```

3.4 算法的分析

这一节简要分析算法 GenPath 和 CheckPattern

的时间复杂性.

算法 GenPath 产生所有的检验路径,因此其复杂度受限于检验路径的数目.由于我们限制了循环体的最大执行次数 M,因此检验路径是有限的,从而算法 GenPath 是能终止的.对于嵌套循环,由于内外层循环的执行次数是独立的;外层循环的循环体的每次执行中,内层循环都可能执行 M 次,因此总的复杂度是 M^M(注意,此处的复杂度是指算法 GenPath 对嵌套循环产生检验路径的复杂度,而不是指循环本身的复杂度).但是,由于程序中有很多的路径实际上是不可行的[1],所以通常情况下达不到 M^M这个最坏复杂度.从实验结果来看,路径数目和路径生成时间都在可接受范围内.

算法 CheckPattern 受限于输入路径集合的大小,而路径集合的大小又受限于总的路径条数 M^M.但由于我们在死循环模式检查之前,先对路径的可行性进行判断,同时将路径按照测试数据进行分类,在算法 CheckPattern 中,将路径进行抽象并去掉重复的路径,这样可以进一步减少由于循环体中含有分支而带来的指数爆炸.因此在调用函数*isLoopPattern*检查路径是否符合死循环模式时,输入集合中路径的数目并不是很多,在可接受范围内.

4　原型工具与实验结果

基于上述思想,我们使用 C++语言实现了死循环检测工具 LoopAnalyzer,该工具可以对给定的 C 语言函数,判断其中是否含有死循环.工具基本按照图 5 描述的框架划分模块,并且实现了 GenPath 和 CheckPattern 算法.在实现 GenPath 算法时,通过实验比较,最终选用了深度优先遍历算法,并通过输入参数来限制未知循环次数的循环体的最大执行次数;同时在存储上选用数组作为路径集合,保持了路径按照循环体执行次数由小到大的顺序.在实现算法 CheckPattern 时,记录了路径集合的测试数据,所以 CheckPattern 返回 true 时,还能给出这个死循环的测试数据.

4.1　工具示例

图 6 所示是一段包含死循环的二分查找 BinarySearch 的代码,其对应的控制流图如图 7 所示.

```
#define N 10
int binarySearch(int A[N], int x)
{
    int mid, present=0, left=0, right=N-1;
    while (left<=right && present==0)
    {
        mid=(left+right)/2;
        if (A[mid]>x) right=mid;
        /*错误,应为 right=mid+1 */
        else if (A[mid]<x) left=mid;
        /*错误,应为 left=mid-1*/
        else present=1;
    }
    if (left>right) return -1;
    else return mid;
}
```

图 6　BinarySearch 代码

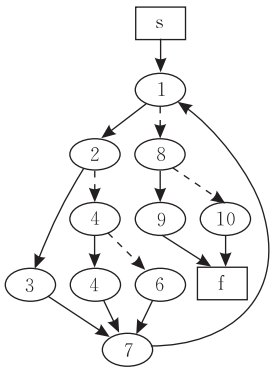


图 7　BinarySearch 控制流图

用 BinarySearch.c 作为输入运行工具 LoopAnalyzer,并限制循环体执行最大次数为 12,在 0.06s 内产生了 95 条可行的检验路径.对这些路径进行分析,得到了 6 组能使得 BinarySearch 无限执行的测试数据.其中一组数据如图 8 所示.其中 LoopMode 一行表示死循环模式,TestData 一行表示测试数据,没有标出的变量表示可以任意取值.

手动执行 BinarySearch,发现有如下结果:

1. left=0, right=9, mid=4, A[mid]=A[4]=2, x=1, A[mid]>x;
2. left=0, right=4, mid=2, A[mid]=A[2]=2, x=1, A[mid]>x;
3. left=0, right=2, mid=1, A[mid]=A[1]=0, x=1, A[mid]<x;
4. left=1, right=2, mid=1, A[mid]=A[1]=0, x=1, A[mid]<x;
5. …

从第 4 行开始,也就是循环体执行 3 之后,left 和 right 的值将保持为 1 和 2,同时条件 A[mid]<x 也将一直满足,以节点 2 为入口的循环将一直执行,程序陷入了死循环.

图 8 所示的结果表明,〈s,1,2,(2B,2)*〉是 BinarySearch 的一个循环模式,BinarySearch 在执

行 3 次循环体之后陷入了死循环. 工具检测的结果和人工分析代码得出的结论一致.

```
There are 6 test data for dead loops.

--------- Output Dead Loop Data --------
LoopHead: 2
TestData: int A[1]=0; int A[2]=2; int A[4]=2; int x=1;
LoopPattern: <s,1,2,(2B,2)*>
------- End of Output Dead Loop --------
```

图 8 BinarySearch 检测结果

4.2 实验数据

除了上述实验结果外,我们还对其它一些程序进行了测试,包括快速排序中 Partition 函数的错误版本等近 10 个程序. 工具 LoopAnalyzer 都能很快找出这些程序中的死循环.

此外,Velroyen 在文献[7]给出了一组基准程序,用来评判非终止性(死循环)检测工具. 我们采用这些基准程序进行了测试,并将实验结果与文献[7]的结果对比列在表 2 中. 其中,文献[7]没有给出运行时间,故仅仅用"√"表示能正确地判断出程序是否终止. 另外,符号"---"表示不能判断出程序是否终止,"*"表示需要对程序进行一定的处理. 我们实验的环境是 Intel Core 2 Duo E6550 2.33G CPU, 2GB Memory, Fedora 8 Linux.

表 2 实验结果

程序名	本文结果	文献[7]结果	程序名	本文结果	文献[7]结果
ex01	<0.01s	√	alternDivWide	<0.01s	---
ex02	0.02s	√	alternDivWidening	<0.01s	---
ex03	0.01s	√	twoFloatInterv	0.07s	√
ex04	<0.01s	√	upAndDownIneq	0.07s	√
ex05	<0.01s	√	whileIncrPart	<0.01s	√
ex06	0.02s	√	upAndDown	0.06s	√
ex07	0.04s	√	alternatingIncr	0.04s	√
ex08	0.11s	√	narrowKonv	0.02s	√
gauss	<0.01s	√	complInterv2	0.01s	√
even	<0.01s	√	complInterv3	<0.01s	√
fib	<0.01s	√	mirrorIntervSim	<0.01s	√
lcm	0.05s	√	mirrorInterv	<0.01s	---
flip	<0.01s	√	moduloLower	<0.01s	√
flip2	0.01s	---	complxStruc	0.06s	---
plait	<0.01s	---	whileNested	0.01s	√*
sunset	0.04s	√	whileNestedOffset	0.04s	√*
narrowing	0.34s	√	alternKonv	0.03s	√
ex09half	0.11s	---	moduloUp	0.02s	√*
trueDiv	<0.01s	√	whileIncr	<0.01s	√
marbie1	<0.01s	√	whilePart	<0.01s	√
marbie2	<0.01s	√	whileTrue	<0.01s	---
alternDiv	<0.01s	√	whileDecr	0.02s	---
cousot	0.02s	√	whileSingle	<0.01s	√
middle	---	√	whileBreak	0.01s	√
whileSum	---	---	doubleNeg	---	---
collatz	---	---	complInterv	---	√

在表 2 中,whileDecr 程序是唯一的对于任意输入肯定终止的循环程序,其它 52 个程序都存在无限循环. 从表 2 可以看出,LoopAnalyzer 能成功地检测出 1 个具有终止性的程序和 47 个死循环程序,准确率达到 90% 以上(而文献[7]的准确率不到 80%). 在 LoopAnalyzer 能检测出的死循环中,包括含有复杂控制结构的实例(如 complxStruc)和循环嵌套的程序(如 whileNested 等),对这些程序使用文献[7]中的方法有一定的困难.

LoopAnalyzer 不能正确检测的程序有两种情况,一部分程序含有取模和整数除法运算以及一些其它的非线性运算,而后端路径分析工具 EPAT 暂时不能处理这些非线性运算;另一部分程序虽然没有非线性运算,但是由于我们的分析方法采用了一些近似的策略(见前文第 3.3 节),通过测试数据来划分子集,这样,在路径集合中可能存在一部分路径(称之为干扰路径)并不属于这个死循环模式. 干扰路径的存在使得该路径子集不能归结为一个死循环模式,从而可能造成死循环的漏报,这也正是本文方法的一个不足之处. 同文献[7]一样,我们的工具也不能判断出程序 Collatz① 是否含有死循环(这个程序是否能终止,至今是个尚未解决的问题).

5 相关工作

死循环相关的工作主要集中在两个方面,程序的终止性证明和程序的非终止性分析.

关于终止性的研究,很多都是基于循环不变式(loop invariants)和阶函数(ranking functions),包括线性阶函数[8]、多项式阶函数[9]以及通过符号计算来产生阶函数[10]的方法. 在此基础上,Cook 等人[11]给出基于 Abstraction Refinement 的方法来证明程序终止性. 此外,Kapur[12]给出了一种利用项重写系统(term rewriting system)来生成循环不变式的方法,以证明程序的终止性问题.

关于非终止性的研究,我们之前的工作[13]给出了一种能检测出简单循环中出现死循环的充分条件. Velroyen[7]也给出了一种基于循环不变式的方法来检测出程序中的死循环,并给出了一组实验程序. 这两种方法都只能处理简单的循环,不能很

① Collatz Problem. http://mathworld.wolfram.com/CollatzProblem.html

好地检测出嵌套循环中的死循环.此外 Velroyen 的方法也无法处理含有复杂控制结构的程序.Gupta[14]等人给出了一种基于套索(lasso)的方法证明程序的非终止性.lasso 由两部分组成,前缀(stem)和环(loop),stem 和 loop 都是有限的路径,同时 loop 还必须是控制流图中的环.该方法分为两步:先找到程序中的 lasso;然后对 lasso 的可行性进行分析,看是否能导致死循环.Gupta 也指出判断 lasso 是否可行与寻找循环不变式是等价的.这种方法的缺点是,需要通过动态运行程序来枚举出程序中的 lasso,而这些 lasso 并不一定导致死循环.同时,文中也指出,并不是所有的死循环都能通过 lasso 检测出来.

6　结论及进一步工作

本文提出了一种基于程序静态分析的死循环检测方法,即通过循环展开和路径可行性分析技术,判断出程序中是否存在一组测试数据能让程序在某种程度上一直执行下去.实验结果表明,我们的方法能够处理一些控制结构复杂以及含有嵌套循环的程序,并检测出其中的死循环.我们实现的工具自动化程度高,能处理较为真实的 C 语言程序.从实验结果来看,工具检测死循环的效率和准确率较高.

将来的工作主要有两个方面,一方面是寻求一种好的识别干扰路径的方法,通过消除路径集合中的干扰路径,进一步提高工具检验死循环的准确率.另一方面考虑扩展该方法使之能够处理递归程序.对于含有函数调用语句的程序,可以在函数调用图基础上进行分析,通过找到函数调用图中的环,并分析该环对应的路径的可行性,进而检测由于两个函数之间无限递归而导致的死循环.

参　考　文　献

[1] Zhang Jian. Sharp static analysis of programs. Chinese Journal of Computers, 2008, 31(9): 1549-1553(in Chinese)
(张健.精确的程序静态分析.计算机学报, 2008, 31(9): 1549-1553)

[2] Zhang Jian, Wang Xiao-Xu. A constraint solver and its application to path feasibility analysis. International Journal of Software Engineer and Knowledge Engineer, 2001, 11(2): 139-156

[3] Liu Lei et al. Techniques of Program Analysis. Beijing: China Machine Press, 2005(in Chinese)
(刘磊等. 程序分析技术. 北京: 机械工业出版社, 2005)

[4] Beizer B. Software Testing Techniques. New York, NY, USA: John Wiley & Sons, Inc., 1989

[5] Zhang Jian. Symbolic execution of program paths involving pointer and structure variables//Proceedings of the 4th International Conference on Quality Software (QSIC2004). IEEE Computer Society, Braunschweig, Germany, 2004: 87-92

[6] Ruan Hui, Zhang Jian, Yan Jun. Test data generation for C programs with string-handling functions//Proceedings of the 2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE2008). Nanjing, China: IEEE Computer Society, 2008: 219-226

[7] Velroyen H. Automatic non-termination analysis of imperative programs [M. S. dissertation]. Chalmers University of Technology, Goteborg, 2007

[8] Bradley A, Manna Z, Sipma H. Linear ranking with reachability//Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005). Edinburgh, Scotland, UK, 2005: 491-504

[9] Bradley A, Manna Z, Sipma H. The polyranking principle//Proceedings of 32nd International Colloquium on Automata, Language and Programming (ICALP 2005). Lisbon, Portugal, 2005: 1349-1361

[10] Yang Lu, Zhan Naijun, Xia Bican, Zhou Chaochen. Program verification by using DISCOVERER//Proceedings of the first International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2005). Zurich, Switzerland, 2005: 528-538

[11] Cook B, Podelski A, Rybalchenko A. Abstraction refinement for termination//Proceedings of the 12th International Symposium on Static Analysis (SAS 2005). London, UK, 2005: 87-101

[12] Rodríguez-Carbonell E, Kapur D. Program verification using automatic generation of invariants//Proceedings of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC 2004). Guiyang, China, 2004: 325-340

[13] Zhang Jian. A path-based approach to the detection of infinite looping//Proceedings of the 2nd Asia-Pacific Conference on Quality Software (APAQS 2001). IEEE Computer Society. Hong Kong, China, 2001: 88-96

[14] Gupta A, Henzinger T, Majumdar R, Rybalchenko A, Xu R G. Proving non-termination//Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008). San Francisco, California, USA, 2008: 147-158

RUAN Hui, born in 1984, M. S. candidate. His main research interests include program analysis and automatic software testing.

YAN Jun, born in 1980, Ph. D., assistant professor. His research interests include constraint processing, program analysis and software testing.

ZHANG Jian, born in 1969, Ph. D., professor, Ph. D. supervisor. His main research interests include automated reasoning, formal methods, program analysis and software testing.

Background

Program correctness is one of the most important issues for computer scientists and engineers. For the past 10 years, we have been working on the automatic generation of test data for programs, and on finding bugs in programs through static checking.

Termination is a key property of programs. While many methods and techniques have been proposed to prove the termination of programs, there has been little research on how to show that a program does not terminate. Based on the authors' previous works, this paper proposes a new method for finding possibly non-terminating programs. The method is automatic and static. It generates a set of paths from the program's flow graph, and compares the program's behavior with a set of non-termination patterns. Like previous approaches, the method is not complete; but it can deal with programs having nested loops, and the experimental results show that the method is quite accurate and efficient.

This work is partially supported by the National Natural Science Foundation of China under grant No. 60633010 and the National High Technology Research and Development Program (863 Program) of China with project No. 2009AA01Z148.