内存泄漏故障静态分析研究

叶俊民 魏 鹏 金 聪 王敬华 张清国 张 维
(华中师范大学计算机科学系 武汉 430079)

摘 要 目前研究人员主要采用静态测试技术实施对内存泄漏故障的检测,其基本思想就是依据待测程序的控制流图来设计特定的算法以检测内存泄漏问题,但这些方法的不足之处主要是控制流图的表示方式上未含有进一步可用信息,因此所设计的算法不能很好地执行该故障的检测任务。为此,定义了一种用于内存泄漏故障检测的控制流图,提出控制流图可达路径生成算法,然后根据生成的路径进行内存泄漏故障的检测与分析。实验证实,该方法取得了理想的效果。

关键词 内存泄漏,故障检测,控制流图,故障模型,静态分析技术

Static Analysis of Memory Leak Fault

YE Jun-min WEI Peng JIN Cong WANG Jing-hua ZHANG Qing-guo ZHANG Wei
(Department of Computer Science, Central China Normal University, Wuhan 430079, China)

Abstract Recent researches detect memory leaks mainly through static test methods. The basic idea is to detect memory leaks by designing specific algorithm based on the control flow graph of program to be tested, but the limitations of these methods are that control flow graph doesn't contain useful information for further use, so that designed algorithms can't detect memory leaks efficiently. We defined a control flow graph for detecting memory leaks and proposed control flow graph reachable paths generating algorithm to detect and analyze memory leaks for generated paths. This method was proved by experiments to be very effective.

Keywords Memory leaks, Faults detect, Control flow graph, Fault model, Static analyses techniques

1 引言

内存泄漏故障会减少可用内存数量从而影响程序的正常运行,甚至会导致应用程序的崩溃。针对内存泄漏故障的检测,国内外的研究人员进行了相关研究[1-13],这些研究的主要做法是采用静态测试来检测内存泄漏故障,其基本思想就是依据待测程序的控制流图来设计特定的算法以检测内存泄漏问题,但这些方法的不足之处主要是控制流图的表示方式上未含有进一步可用信息,相关的检测算法不能很好地执行内存泄漏检测任务,此外内存泄漏这一问题涉及程序设计本身,这更加剧了该故障检测的难度。因此,有必要进一步研究和探索内存泄漏故障检测这一问题。

2 研究基础

2.1 指针操作的控制流图及其形式化定义

定义 1 控制流图为一个有向图,该图由一个节点集合 V 和一个边的集合 E 组成,其中节点代表蕴含指针操作的语句,边实际上是定义在节点集合 V 上的一个关系,可以表示为 $E=\langle Vstart, Vend\rangle$,即从 $Vstart$ 开始到 $Vend$ 结束,将该图记作 CFG,则有 $CFG=(V,E)$。

定义 2 对于控制流图中的任意一个节点,令 id 表示该节点在控制流图中的编号,num 表示该节点对应的源程序的行号,content 表示该节点蕴含的源程序语句信息,nextNodes 表示该节点指向的下一个节点,则支持内存泄漏故障检测的控制流图中的一个节点可定义成为一个四元组$\langle id, num, content, nextNodes\rangle$,记作 Node,则有 $\forall Node \in CFG$。

定义 3 所谓路径,是 CFG 中任意一条执行序列,记作 Sequence,则有 $Sequence=\{Node_{start}, \cdots, Node_{end}\}$,其中 $Node_{start}$ 与 $Node_{end}$ 分别代表首节点和尾节点。

通过如图 1(a)所示的程序说明了上述概念。

其中,对于"〈37〉 p = malloc(sizeof(int));",其各项的含义是:"〈37〉"表示该条语句在实际待测程序中的行号,"p = malloc(sizeof(int));"表示具体语句。上述程序段对应的控制流图如图 1(b)所示。

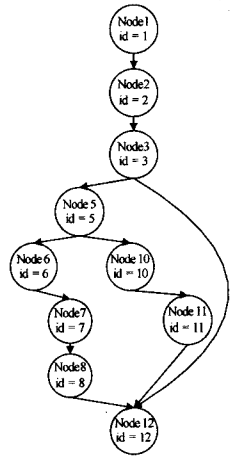
根据定义 2,首节点可描述为"〈1,37,'p = malloc(sizeof(int));',[2]〉",其中 1 表示该节点在控制流图中的编号,而[2]表示该节点所指向控制流图中的下一个节点。由此,该程序控制流图可用如表 1 所列的节点集合描述。

到稿日期:2009-07-22 返修日期:2009-10-28 本文受武汉大学计算机软件工程国家重点实验室开放基金项目(编号:SKLSE20080705),湖北省自然科学基金(编号:2007ABA034,2008CDB349),华中师范大学中央高校基本科研业务费项目(编号:CCNU09Y01009 和 CCNU09Y01013)资助。

叶俊民(1965-),男,博士,教授,主要研究方向为高可信软件工程、软件质量保障等,E-mail:jmye@mail.ccnu.edu.cn;魏 鹏 硕士生,主要研究方向为软件工程;金 聪 博士,教授,主权研究方向为软件工程。

```
.....
〈37〉 p = malloc(sizeof(int));
〈38〉 q = malloc(sizeof(int));
〈39〉 if(x == 5){
〈40〉 return 0;}
〈41〉 if(x == 7){
〈42〉 p = &x;
〈43〉 free(p);
〈44〉 free(q);
〈45〉 return 0;}
〈46〉 free(p);
〈47〉 free(q);
〈48〉 return 0;
.....
```

(a)



(b)

图 1　待测程序及其对应的控制流图

表 1　待测程序控制流图节点集合描述

节点名称	节点描述
$Node_1$	〈1,37,'p = malloc(sizeof(int));',[2]〉
$Node_2$	〈2,38,'q = malloc(sizeof(int));',[3]〉
$Node_3$	〈3,39,'if(x == 5){',[5,12]〉
$Node_5$	〈5,41,'if(x == 7){',[6,10]〉
$Node_6$	〈6,42,'p = &x;',[7]〉
$Node_7$	〈7,43,'free(p);',[8]〉
$Node_8$	〈8,44,'free(q);',[12]〉
$Node_{10}$	〈10,46,'free(p);',[11]〉
$Node_{11}$	〈11,47,'free(q);',[12]〉
$Node_{12}$	〈12,48,'return 0;',[]〉

由定义 1 可知，$CFG=\{Node1, Node2, Node3, Node5, Node6, Node7, Node8, Node10, Node11, Node12\}$，该集合中缺少 *Node*4 与 *Node*9，这是因为执行上述两条语句后，程序均会到达 *Node*12，而 *Node*12 作为该程序段对应控制流图的尾节点，因而到达 *Node*4 与 *Node*9 均等价于到达 *Node*12，所以将它们剔除出 CFG。在此基础上，可从表 1 中获取该程序控制流图的所有可达路径如下。

路径 1：节点 1->节点 2->节点 3->节点 12；

路径 2：节点 1->节点 2->节点 3->节点 5->节点 6->节点 7->节点 8->节点 12；

路径 3：节点 1->节点 2->节点 3->节点 5->节点 10->节点 11->节点 12。

依据定义 3，上述路径内容可以表示为：

$Sequence1:\{Node1, Node2, Node3, Node12\}$；

$Sequence2:\{Node1, Node2, Node3, Node5, Node6, Node7, Node8, Node12\}$；

$Sequence3:\{Node1, Node2, Node3, Node5, Node10, Node11, Node12\}$。

由定义 1、定义 2、定义 3 可以推知：

性质 1　$\bigcup_{i=1}^{n} Sequence\ i = CFG$，其中 n 为由 CFG 生成的所有可达路径的个数。

证明(反证法)：在程序控制流图 CFG 中，*Sequence*1，*Sequence*2 至 *Sequencen* 为 CFG 生成的所有可达路径，假设路径 *Sequencei* 上存在一个节点 $Node_j$，有 $Node_j \notin CFG$，其中 $1 \leqslant i \leqslant n$。因为路径 *Sequencei* 为 CFG 中的一条执行序列，所以该路径上所有的节点均属于 CFG，则有 $Node\ j \in CFG$，与题设 $Node\ j \notin CFG$ 矛盾，可知路径上所有的节点均属于 *CFG*，又因为 *Sequence*1 至 *Sequencen* 为 *CFG* 生成的所有路径，所以性质 1 得证。

性质 2　$\bigcap_{i=1}^{n} Sequence\ i \supseteq \{Node_{start}, Node_{end}\}$，其中 n 为由 *CFG* 生成的所有可达路径的个数，$Node_{start}$ 与 $Node_{end}$ 分别代表首节点和尾节点。

证明：根据定义 3 可知，路径是 *CFG* 执行序列上的节点的集合，且有 $Sequence=\{Node_{start}, \cdots, Node_{end}\}$，可以看出每条路径必包含首、尾节点，所以性质 2 得证。

2.2　指针映射集合及其操作定义

在 C/C++程序中，指针变量指向内存中的一块区域，程序通过操纵指针变量来控制指针所指向内存的内容。

定义 4　令 P 表示程序中的指针集合，M 表示堆内存地址集合，程序中任意指针变量 V_i 可表示成为 $V_i \in \{P, M\}$ 或者 $V_i=[P_i, M_i]$，其中 i 表示指针变量的编号。

本文将建立指针映射集合来支持对控制流图的静态分析，通过对指针映射集合的操作来模拟程序中对于指针的操作(如内存的申请、交换与释放)，进而模拟并解释程序运行时期的内存状态，由此作为内存泄漏故障的判断依据。

定义 5　将程序中所有分配的指针变量组成的集合，称为指针映射集合，记作 pointerSet，则有 $\forall V_i \in pointerSet$，其中 i 表示指针变量的编号。指针映射集合 *pointerSet* 中的操作定义如下：

(1)Add(〈*pointer*, M_i〉)，将指针变量〈*pointer*, M_i〉加入到指针映射集合 *pointerSet* 中，其中 $pointer \in P$，i 表示为堆内存地址集合中的任意一块。由于是静态分析，仅需要知道 pointer 指向一块内存单元 M_i 即可，无需关心分配的内存单元的大小与位置，因此 i 可以设置为一个递增变量，本文中将进行 Add 操作时的系统时间的毫秒数作为堆内存地址；

(2)Modify(〈$pointer_i$, M_i〉, 〈$pointer_j$, M_j〉)或者 $Modify(V_i, V_j)$，将指针 $pointer_i$ 指向的内存地址 M_i 修改为 Mj；

(3)Get($pointer_i$)，将 $pointer_i$ 所指向的堆内存地址返回；

(4)Delete($pointer_i$)，将 $pointer_i$ 指向的所有堆内存地址从指针映射集合 *pointerSet* 中删除。

上述 4 种操作对应的算法描述如下：

(1)增加指针变量(Add)

主要针对通过 malloc，realloc 和 new 关键字动态申请内存的情况，假设已经建立起指针映射集合 pointerSet，T 代表系统当前时间的毫秒数(该毫秒数开始于格林尼治时间 1970 年 1 月 1 日 0 时 0 分 0 秒)，pointerSet 代表指针映射集合，具体算法步骤如图 2 所示。

```
算法:新建指针变量,并将其插入指针映射集合中
输入:指针变量〈Pi,Mi〉
输出:无
Begin
    取得当前系统时间的毫秒数 T;
    将 Mi 赋值成为 T;
    建立指针变量 Vi=〈Pi,Mi〉,其中 Mi 的值为 T;
    将 Vi 加入指针映射集合 pointerSet 中;
End
```

图 2　增加指针变量算法

(2)修改指针变量(Modify)

主要针对指针变量之间的交换,假设已经建立起指针映射集合 *pointerSet*,具体算法步骤如图 3 所示。

```
算法:修改指针映射集合中指针指向的内存地址
输入:指针变量(pointer_i,M_i),(pointer_j,M_j)
输出:无
Begin
    //使用操作 Get(pointer_i)得到 pointer_j 所指向的堆内存地址 Mj
    M_j=Get(pointer_j);
    将(pointeri,M_i)修改为(pointer_i,M_j);
    Add((pointer_i,M_j));
    将指向堆内存 M_i 的指针修改为 Null;
End
```

图 3 修改指针变量算法

(3)获取内存地址(Get)

主要针对获取指针变量所指向的堆内存地址,假设已经建立起指针映射集合 *pointerSet*,指针变量 V_i,指针 P_i,具体算法步骤如图 4 所示。

```
算法:返回指针指向的内存地址
输入:指针 pointer_i
输出:无
Begin
顺序遍历指针映射集合 pointerSet 中的每个指针变量 V_i;
    if(指针变量 V_i 满足 P_i=pointer_i)
        返回 M_i,其中 V_i=(P_i,M_i);
    else
        //Null 代表空值,即不存在该指针变量
        返回 Null;
End
```

图 4 获取内存地址算法

(4)删除指针变量(Delete)

主要针对程序中使用 free 或 delete 关键字释放 $pointer_i$ 所指向的堆内存,假设已经建立起指针映射集合 *pointerSet*,堆内存地址 M_i,M_j,指针变量 V_j,指针 P_i,指针映射集合 *pointerSet*,具体算法步骤如图 5 所示。

```
算法:删除指针映射集合中指定的指针变量
输入:指针 pointer_i
输出:无
Begin
    M_i=Get(pointer_i);
    顺序遍历指针映射集合 pointerSet 中的每个指针变量 V_j;
    if(指针变量 V_j 满足 M_j=M_i)
        从指针映射集合 pointerSet 中删除该指针变量 V_j;
End
```

图 5 删除指针变量算法

下面通过图 1 所示的实例说明上述算法的使用。

```
(37) p = malloc(sizeof(int));
(38) q = malloc(sizeof(int));
(39) p = q;
(40) free(p);
```

待测程序段各语句的操作及其指针映射集合状态如表 2 所列。

表 2 待测程序段各语句的操作及其指针映射集合状态

程序语句	操作	指针映射集合状态
(37) p = malloc(sizeof(int));	Add((p,1211886733562))	{(p,1211886733562)}
(38) q = malloc(sizeof(int));	Add((q,121188673622))	{(p,1211886733562),(q,121188673622)}
(39) p = q;	Modify((p,1211886733562),(q,121188673622))	{(Null,1211886733562),(p,1211886733622),(q,1211886733622)}
(40) free(p);	Delete(p)	{(Null,1211886733562)}

2.3 内存泄漏故障的形式化定义

定义 6 设待测程序对应的控制流图 CFG 中的可达路径为 *Sequence*,*pointerSet* 为 *Sequence* 所对应的指针映射集合,当检测过程结束时,若 $pointerSet=\Phi$,则表明该待测程序可达路径 *Sequence* 上不存在内存泄漏;否则说明该路径 *Sequence* 上存在内存泄漏。

考虑图 1 中的待测程序段。根据定义 1、定义 2 与定义 3,该待测程序段对应的控制流图 $CFG=\{Node1,Node2,Node3,Node4\}$,其中各节点(*Node*)表示形式如下:

*Node*1:⟨1,37,'*p = malloc(sizeof(int));*',[2]⟩

*Node*2:⟨2,38,'*q = malloc(sizeof(int));*',[3]⟩

*Node*3:⟨3,39,'*p = q;*',[4]⟩

*Node*4:⟨4,40,'*free(p);*',[]⟩

由于该待测程序段是顺序执行,因此由 *CFG* 生成的可达路径只有一条,即 $Sequence=\{Node1,Node2,Node3,Node4\}$。

建立该路径 *Sequence* 所对应的指针映射集合 *pointerSet*,并令其为空。如图 1 中,依据定义 6 进行指针映射集合 *pointerSet* 的操作,操作完毕后,*pointerSet* 中仍存在指针变量⟨Null,1211886733562⟩,即 *pointerSet* = {⟨Null,1211886733562⟩},表明待测程序段路径 *Sequence* 上存在内存泄漏故障。

3 内存泄漏故障的静态分析

3.1 内存泄漏故障的基本检测思想

一般常说的内存泄漏是指堆内存的泄漏。堆内存是指程序从堆中分配的,大小任意的(内存块的大小可以在程序运行期决定),使用完后必须显示释放的内存。应用程序一般使用 malloc,realloc,new 等关键字从堆中分配到一块内存,使用完后,程序必须负责相应的调用 free 或 delete 释放该内存块,否则,这块内存就不能被再次使用,我们就说这块内存泄漏了。

因而基于内存泄漏故障的检测思路如下:

(1)依据待测程序关于内存操作语句的控制流图,生成从首节点开始到尾节点结束的所有可达路径;

(2)依次遍历各条可达路径,分别建立每条路径的指针映射集合;

(3)根据各条路径中各个节点对于内存指针的操作,对路径相对应的指针映射集合进行相对应的操作;

(4)最后检测各条可达路径所对应的指针映射集合中的内存指针状态,从而判定是否存在内存泄漏故障。

3.2 控制流图的可达路径生成步骤与算法

在进行指针映射集合的操作之前,必须提供待测程序控制流图的所有可达路径,因而获取待测程序控制流图的可达路径是内存泄漏故障检测的先决条件。

本文采取如下步骤获取待测程序控制流图的可达路径：

(1)根据定义 1，待测程序的控制流图被表示成为节点(Node)的集合；

(2)节点(Node)集合，即 *CFG*，采用 XML 方式进行描述，作为该可达路径生成算法的输入；

(3)检测程序利用可达路径生成算法处理输入；

(4)返回所有可达路径。

其中，步骤(2)中节点(Node)对应的 XML 表示形式如图 6 所示。

```
〈node〉
    〈id〉节点编号〈/id〉
    〈num〉节点对应源程序行号〈/num〉
    〈content〉节点内容，即程序语句〈/content〉
〈next-nodes〉下一个节点〈/next-nodes〉
〈/node〉
```

图 6　节点(Node)对应的 XML 表示形式

在 Node 元素中包含的 *id*，*num*，*content* 和 *next-nodes* 依次对应的是定义 1 中四元组的 *id*，*num*，*content* 和 *next-Nodes*。

步骤(3)中的控制流图可达路径生成算法如图 7 所示，其中 node 表示为节点，path 数组用来记录路径的回朔过程，pre 数组用来记录任意节点的前续节点。

```
算法：根据控制流图 CFG 的 XML 描述文件，生成控制流图所有的可达路径
输入：控制流图 CFG 的 XML 描述文件
输出：所有可达路径
Begin
    if (node 为 CFG 中最后一个节点){
        建立一条空路径 sequence;
        int i = 0;
        将该节点加入到 path[i];
        while (pre[node. getId()] ! = Null){
            i++;
            node = pre[node. getId()];
            path[i] = node;
        }
        将 path 数组中所存节点，逆序加入 sequence;
        返回一条路径 sequence;
    }
    else{
        while(该节点的子节点 aNode ! = Null){
            pre[aNode. getId()] = node;
            generateSequences(aNode);
        }
    }
End
```

图 7　可达路径生成算法

上述算法使用了递归，对控制流图 *CFG* 采取了深度遍历，由于控制流图 *CFG* 采取了邻接表的存储方式，每个节点与边均只被遍历了一次，因此该算法的时间复杂度为 $O(n+e)$，其中 n 代表图中点的个数，e 代表图中边的条数。

3.3　基于可达路径的内存泄漏检测算法

基于可达路径的内存泄漏检测算法如图 8 所示，其中 *pointerSet* 表示指针映射集合，*Node* 表示节点。

```
算法：检测指针映射集合中的指针变量状态
输入：路径 Sequence
输出：内存泄漏报告
Begin
    建立路径 Sequence 对应的指针映射集合 pointerSet，并初始化为空；
    while(Sequence 中存在节点 Node 未被遍历){
        遍历节点 Node，依据定义 5 对指针映射集合
        pointerSet 进行操作；
    }
    //如果指针映集合为空
    if(pointerSet == Null){
        路径 Sequence 上不存在内存泄漏故障；
    }
    //如果指针映射集合不为空
    else{
        路径 Sequence 上存在内存泄漏故障；
    }
End
```

图 8　基于可达路径的内存泄漏检测算法

上述算法过程采用顺序遍历的方式来遍历路径 *Sequence* 中的每个节点 *Node*，然后再根据 *Node* 中的语句信息对指针映射集合 *pointerSet* 进行相关操作，由于 *Sequence* 采取线形表的方式存储，因此该算法的时间复杂度为 $O(n)$，其中 n 代表 *Sequence* 中节点个数。

4　实验验证

根据图 1 所示实例，其对应的 XML 表述形式如图 9 所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<nodes>
<node>
  <id>1</id>
  <num>37</num>
  <content>p=malloc(sizeof(int));</content>
  <next-nodes>2</next-nodes>
</node>
<node>
  <id>2</id>
  <num>38</num>
  <content>q=malloc(sizeof(int));</content>
  <next-nodes>3</next-nodes>
</node>
<node>
  <id>3</id>
  <num>39</num>
  <content>if(x == 5)</content>
  <next-nodes>5, 12</next-nodes>
</node>
<node>
  <id>5</id>
  <num>40</num>
  <content>if(x == 7)</content>
  <next-nodes>6, 10</next-nodes>
</node>
<node>
  <id>6</id>
  <num>42</num>
  <content>p = &x;</content>
  <next-nodes>7</next-nodes>
</node>
<node>
  <id>7</id>
  <num>43</num>
  <content>free(p);</content>
  <next-nodes>8</next-nodes>
</node>
<node>
  <id>8</id>
  <num>44</num>
  <content>free(q);</content>
  <next-nodes>12</next-nodes>
</node>
<node>
  <id>10</id>
  <num>46</num>
  <content>free(p);</content>
  <next-nodes>11</next-nodes>
</node>
<node>
  <id>11</id>
  <num>47</num>
  <content>free(q);</content>
  <next-nodes>12</next-nodes>
</node>
<node>
  <id>12</id>
  <num>48</num>
  <content>return 0;</content>
  <next-nodes></next-nodes>
</node>
</nodes>
```

图 9　待测程序控制流图 XML 表述形式

以上述 XML 文件输入检测程序(检测程序界面如图 10 所示)，根据 3.2 节中的步骤(3)，生成待测程序对应控制流图 CFG 的所有可达路径，结果如下：

*Sequence*1：{*Node*1，*Node*2，*Node*3，*Node*12}；

*Sequence*2：{*Node*1，*Node*2，*Node*3，*Node*5，*Node*6，*Node*7，*Node*8，*Node*12}；

*Sequence*3：{*Node*1，*Node*2，*Node*3，*Node*5，*Node*10，*Node*11，*Node*12}。

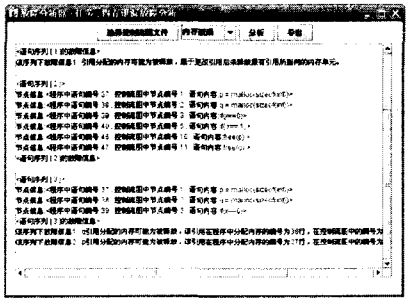


图 10　检测程序界面

选取 *Sequence*1 作为示例，根据 3.3 节中的基于可达路径的内存泄漏检测过程，有表 3 所列的结论。检查 *pointerSet*1，由于 *pointerSet*1 不为空，根据定义 6，该段程序存在内存泄漏故障。

表 3　路径 Sequence1 及其指针映射集合状态

节点(Node)	Sequence1 指针映射集合的状态
Node1	{⟨p,1211886733562⟩}
Node2	{⟨p,1211886733562⟩,⟨q,1211886733622⟩}
Node3	{⟨p,1211886733562⟩,⟨q,1211886733622⟩}
Node12	{⟨p,1211886733562⟩,⟨q,1211886733622⟩}

实验表明，*Sequence*1 存在两处内存泄漏故障。遍历路径 *Sequence*2 结束后，指针映射集合 *pointerSet*2 中任存在一个指针变量⟨Null,1211886733562⟩，根据定义 6，*Sequence*2 存在一处内存泄漏故障。遍历路径 *Sequence*3 结束后指针映射集合 *pointerSet*3 为空，根据定义 6，*Sequence*3 不存在内存泄漏故障。

5　相关工作对比

常见的检测内存泄漏故障的方法主要分为动态检测和静态检测，其中文献[13]采取了动态检测的方式，主要考虑针对函数调用时的内存堆栈信息来判定是否存在内存泄漏故障，但是该方法严重依赖测试用例来确保对待测程序的代码覆盖，从而在测试效率上存在缺陷。而静态检测方式大部分基于待测程序的控制流图，文献[1]通过待测程序的控制流图进行静态分析，较好地解决了测试的代码覆盖问题，然而该方法没有解决控制流图的可达路径生成问题。另一些研究人员通过别名集合[4-9]来分析待测程序的内存分配与释放，以此作为检测内存泄漏故障的依据，其优点是检测效率较高，但该方案存在误报的可能性，并且在别名分析[6]阶段，需要对待测程序的 PCG[10]与 SCFG[10]不断地进行迭代扫描，在算法效率上存在缺陷。本文采用重新定义的控制流图模型来检测内存泄漏故障，并设计控制流图的可达路径生成算法来实现对控制流图的遍历，定义了指针映射集合上模拟待测程序所执行的内存操作，以指针映射集合的状态来作为内存泄漏故障的判断依据，从而降低了误报率。

结束语　本文提出了一种新的内存泄漏故障测试方法，该方法基于对待测程序的控制流图的分析，生成控制流图的所有可达路径并建立其对应的指针映射集合，通过对指针映射集合的操作来反映待测程序相应路径的内存操作，从而给判断该路径是否存在内存泄漏故障提供了依据。根据本文中的指针映射集合，建立了内存泄漏故障模型，并按照该故障模型给出了检测算法以及步骤。下一步研究的主要目标是继续完善指针映射集合上的操作规则，从而增强检测程序的适应性，进而减少检测程序的误报率。另外，该检测程序目前用于检测内存泄漏故障，如何将控制流图的可达路径用于检测其他故障还有待进一步研究。

参 考 文 献

[1] 张威，卢庆龄，李梅，等. 基于指针分析的内存泄露故障测试方法研究[J]. 计算机应用研究，2006

[2] Hastings R, Joyce B. Purify: Fast Detection of Memory Leaks and Access Error[C]// Proceedings of the Winter USENIX Conference. 1999:125-136

[3] Sagiv M, Reps T, Wilhelm R. Solving Shape-analysis Problems in Language with Destructive Updating[C]// Symposium on Principles of Programming Languages. Florida, 1996:110-118

[4] Landi W, Ryder B G. Safe Approximate Algorithm for Interprocedural Pointer Aliasing[C]// ACM SIGPLAN Notices. 1992: 235-248

[5] Wilson R P, Lam M S. Efficient Context - sensitive Pointer Analysis for C Program[C]// Proceedings of the ACM SIGPLANE'95 Conference on Programming Language Design and Implementation(PLDI). California, 1995:18-21

[6] 张广梅，李晓维. 动态内存错误的静态分析[J]. 计算机辅助设计与图形学学报，2005(3)

[7] Heine D L, Lam M S. A practical flow sensitive and context sensitive C and C++ memory leak detector[J]. ACM SIGPLAN Notices, 2003:168-181

[8] Austin T M, Breach S E, Sohi Gurindar S. Efficient detection of all pointer and array access errors[J]. ACM SIGPLAN Notices, 1994, 29(6):290-301

[9] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors[C]// Winter Usenix Conference. SanFrancisco, California, 1992:125-136

[10] Hallem S, Chelf B, Xie Y, et al. A system and language for building system specific, static analyses[J]. ACM SIGPLAN Notices, 2002:69-82

[11] 肖庆，张威，宫云战，等. 内存泄漏的一种静态分析方法[J]. 装甲兵工程学院学报，2004(6)

[12] 宫云战. 一种面向故障的软件测试新方法[J]. 装甲兵工程学院学报，2004(3)

[13] 吴民，涂奉生. 内存泄漏的动态跟踪分析[J]. 计算机工程与应用，2005(14)

(上接第 135 页)

[5] Wu F L, Lakshmivarahan S, Dhall S K. Routing in a class of cayley graphs of semidirect products of finite groups[J]. Journal of Parallel and Distributed Computing, 2000, 60:539-565

[6] Lakshmivarahan S, Dhall S K. Ring, torus and hypercube architectures/algorithms for parallel computing[J]. Parallel Computing, 1999, 25:1877-1906

[7] Das S K, Ohring S, Baneriee A K. Embedding into hyper Petersen network: Yet another Hypercube-like topology[J]. Journal of VLSI Design, 1995, 2(4):335-351

[8] 王雷，林亚平，夏巍. 双环 Petersen 图互联网络及其路由算法研究[J]. 软件学报，2006，17(5)：1115-1123

[9] 王雷，林亚平，陈治平. 二维环/双环互连 Petersen 图网络及其路由算法[J]. 计算机学报，2004，27(9)：1290-1296

[10] 刘方爱，刘志勇，乔香珍. 光 RP(k)网络上 Hypercube 通信模式的波长指派算法[J]. 软件学报，2003，14(3)：575-581