一种基于逆向程序流的程序切片算法*

白文科, 杨献春, 许满武
(南京大学 计算机系 计算机软件新技术国家重点实验室, 南京 210093)

摘 要: 传统的程序切片方法一般基于程序依赖图(PDG)和系统依赖图(SDG)的可达性算法,但是在建立 PDG 和 SDG 的过程中会计算一些与切片无关的数据依赖,造成时空资源的浪费及切片效率的降低。提出了一种基于程序逆向流的切片算法,它事先建立逆向程序流,再从切片点开始沿逆向程序流扫描程序以获得程序切片,只计算与切片相关的数据依赖,从而提高了切片计算的时空效率。通过实验发现该算法具有一定的可行性和实用性。本算法适用于包括 Fortran、C 等编程语言在内的命令式程序的切片生成。

关键词: 程序切片; 程序逆向流; 谓词依赖集; 过程内切片; 过程间切片
中图分类号: TP311.5 文献标志码: A 文章编号: 1001-3695(2009)03-0920-03

Algorithm of program slicing based on reverse program flow

BAI Wen-ke, YANG Xian-chun, XU Man-wu
(State Key Laboratory for Novel Software Technology, Dept. of Computer Science, Nanjing University, Nanjing 210093, China)

Abstract: The traditional method of program slicing was based on reachability algorithm of program dependence graph(PDG) and system dependence graph(SDG). However, to construct PDG and SDG, some data dependenced which were irrelevant to the slicing may be computed. The redundant computing wasted time and memory, and reduced slicing efficiency. To address this problem, this paper presented a slicing algorithm based on reverse program flow. It firstly constructed reverse flow of the program, then scaned the program along reverse flow from the slicing point, and only computed the data dependences which were relevant to slicing. So it improved slicing efficiency. The result of experiment shows this algorithm is feasible and practical. This algorithm can be used for generating slicing of imperative programs, such as Fortran and C program.

Key words: program slicing; reverse program flow; predicate dependence set(PDS); intraprocedural slicing; interprocedural slicing

0 引言

程序切片是一种重要的程序分析理解方法,它广泛应用于程序调试、测试、软件维护、软件度量、逆向工程和再工程等领域。其概念最早由 Mark Weiser[1] 提出,定义为: 对程序 P、程序点 l 和变量 x, 其切片是指可能影响变量 x 在 l 点状态的所有语句和谓词的集合,记做 slice(l, x)。其中 $\langle l, x\rangle$ 为一个切片标准。程序切片可以缩小考察的程序范围,有利于程序员对错误的查找和定位。K. J. Ottenstein 等人[2] 提出了通过遍历程序依赖图(PDG)来获得过程内切片的方法。后来 Horwitz 等人[3] 通过把 PDG 扩展为系统依赖图(SDG)以得到过程间切片。由于顺序程序中依赖关系具有传递性,基于程序依赖图的函数内切片算法是一个简单的图的可达性问题。在系统依赖图上,函数间的切片问题也是一个图的可达性问题[3,4]。

通过程序依赖图和系统依赖图来获得切片,需要先计算程序的控制依赖和数据依赖。但在生成程序依赖图和系统依赖图的过程中,可能会分析执行流不经过切片点的代码,或者计算执行流经过切片点的代码中与切片无关的数据依赖。这些不必要的计算造成了时间和空间资源的浪费,并严重影响了切片计算的效率。事实上,程序的数据依赖受控制依赖制约,因为若交换一段程序语句的执行顺序,则不同的交换方法将导致不同的数据依赖关系。所以对于计算程序切片来说,可以事先生成整个程序的逆向控制流,再从切片点开始沿逆向控制流扫描程序,仅计算与切片相关的数据依赖,在此过程中用控制依赖和刚生成的数据依赖来计算切片,这样就提高了切片算法的效率。本文提出了一种基于此原理的命令式程序的切片方法,并依次给出了过程内和过程间的切片算法,它们消除了传统 PDG、SDG 切片算法中无关的数据依赖计算,提高了算法的性能。

1 基于逆向程序流的切片算法原理

本章先给出命令式程序的语法,然后描述了基于逆向程序流的切片算法原理。本章只讨论了过程内切片,过程间切片在第 3 章讨论。

1.1 命令式程序的语法

下面是所要分析的命令式程序过程内语句的语法,用 BNF 表示如下:

收稿日期: 2008-06-04; 修回日期: 2008-07-14 基金项目: 国家自然科学基金重大研究计划资助项目(90718021)

作者简介: 白文科(1984-),男,陕西汉中人,硕士研究生,主要研究方向为软件方法学、程序分析(bwk1984@gmail.com); 杨献春(1955-),男,高级工程师, CCF 高级会员,主要研究方向为网络计算、分布式及并行系统; 许满武(1944-),男,教授,博导,博士,主要研究方向为软件方法学和新型程序设计.

$a ::= x \mid n \mid a_1\ \mathrm{op}_a\ a_2$

$b ::= \mathrm{true} \mid \mathrm{false} \mid \mathrm{not}\ b \mid b_1\ \mathrm{op}\ b\ b_2 \mid a_1\ \mathrm{op}_r\ a_2$

$S ::= [x = a]^l \mid S_1;\ S_2 \mid \mathrm{if}\ ([b]^l)\{S_1\}\ \mathrm{else}\{S_2\} \mid \mathrm{while}\ ([b]^l)\ \{S\}$

其中:a 为算术表达式,x 为变量,n 为数字常量;b 为逻辑表达式,op_a、op_b 和 op_r 分别表示算术运算符、逻辑运算符和算术比较符;S 为程序语句。标在语句右上方的 l 为一个自然数,它是该语句的编号(label),用于标志对应的程序语句。每条语句的编号在程序中是惟一的。

1.2　切片算法原理

命令式程序的一个显著特点是它有赋值语句。赋值语句是命令式程序中惟一改变内存变量值的语句,由于切片是指所有可能影响所考察变量在切片点取值的语句和谓词,赋值语句是本切片算法关注的主要对象。本文将程序语句的操作分为赋值操作和非赋值操作:赋值操作改变程序的状态;非赋值操作控制程序的执行流或者进行计算。因为影响切片点变量值的直接操作是赋值操作,所以当赋值语句的左式为欲求其切片的变量时,切片中应加入该赋值语句;若同时存在某谓词决定该赋值操作是否执行,则切片中还要加入该谓词;如果该赋值语句的右式或刚加入的谓词中含有变量,则再计算这些变量的切片,最后把这些切片并入目标切片中,由此就可得到目标切片。令切片点的语句编号为 l,变量 x 在该点的切片记做 S,S 初始时为空集。为标记要计算哪些切片,增加一个数据结构:待处理表 U,它用于记录当前应该计算哪些变量的切片,其初始值为$\{(l, x)\}$。计算 S 时,沿逆向程序流扫描分析程序,不断加入可能影响 x 在 l 处值的语句或谓词,直至集合 S 不再扩大为止,如图 1 所示。

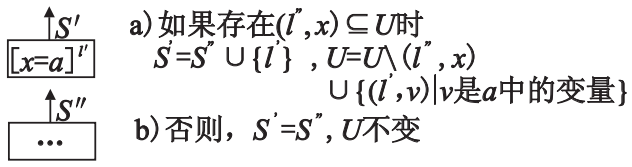

图1　基于逆向程序流的切片算法原理

图 1 中的箭头表示沿逆向程序流扫描程序,S 为 l 点之后的切片,S 为 l 点之前的切片。逆向扫描时,若遇到非赋值语句,则直接把扫描点前移一步。若遇到赋值语句,由于它是程序中惟一改变变量值的操作,有可能处于切片中,需分情况处理:a) 当赋值语句的左式为 S 中某个欲求其切片的变量时,就在 S 中加入该语句的编号,并删除其在待处理表 U 中的待处理标记(l'', x),同时在 U 中加入赋值语句的右式中变量的切片待求标记$\{(l', v) \mid v$ 是 a 中的变量$\}$;b) 否则,S、U 不变,扫描点沿逆向程序流移动一步。

上述切片计算过程是逆向程序流进行的,为此必须事先生成整个程序的执行流 flow,再获得程序逆向流 flow^R。flow 为 (l, l') 的集合,(l, l') 表示执行完 l 处的语句后将可能执行 l' 处的语句。$\mathrm{flow}^R = \{(l', l) \mid (l, l') \in \mathrm{flow}\}$。flow 和 flow^R 的详细说明见文献[4]的第 2 章,这里不再赘述。

现在还有一个问题要解决:如何处理分支语句和循环语句。这些语句中的谓词决定了某些赋值语句是否被执行,所以它们也可能是影响变量取值的语句,需纳入对它们的处理,但同时又要不影响沿逆向程序流求切片的基本思想。为此本文建立了谓词依赖集(PDS),其定义为 $\mathrm{PDS} = \{\langle l, l_b\rangle \mid$ 存在决定 l 处语句是否执行的谓词,谓词编号为 $l_b\}$。有了 PDS 后就可以处理分支语句和循环语句了:当某条语句被加入到切片集合中,且 PDS 中存在该语句编号 l 对应的 $\langle l, l_b\rangle$,则应在切片集合中加入 l_b,并分析 l_b 对应的谓词:若谓词中含有变量,则应该以 l_b 为切片点计算这些变量的切片,最后再并入目标切片集合中。

1.3　最小解原则

最小解原则是用来处理循环语句和递归程序的切片求解。它是指在切片计算过程中,不断扩大的语句和谓词的集合应该是最小的,即不能出现这种情况:沿 flow^R 循环无休止地计算某个变量的切片。举例如下:

程序:$[i = -3]^1$; while($[i < 0]^2$) {$[i = i + 1]^3$;}

求 slice(2, i)

解:$\mathrm{flow} = \{(1, 2), (2, 3), (3, 2)\}$, $\mathrm{flow}^R = \{(2, 1), (3, 2), (2, 3)\}$, $\mathrm{PDS} = \{\langle 3, 2\rangle\}$

依照前面给出的切片计算方法,slice(2, i) 的计算过程为

$\mathrm{slice}(2, i) = \mathrm{slice}(1, i) \cup \mathrm{slice}(3, i) = \{1\} \cup (\{3\} \cup \mathrm{slice}(2, i)) = \{1, 3\} \cup \underline{\mathrm{slice}(2, i)}$

笔者发现计算结果中又包含了 slice(2, i) 的计算,产生了递归。遇到这种情况时,正确的做法应该是中止 $\underline{\mathrm{slice}(2, i)}$ 的计算,这样 slice(2, i) 的最终计算结果为$\{1, 3\}$。

2　过程内程序切片的生成算法

根据 1.2 节的算法原理,计算切片前只需生成 flow 和 PDS 这两个集合就可以了。无论是 flow 还是 PDS,它们实际上都是一种控制流。下面给出了过程内程序切片的生成算法:

输入:程序 P,切片点 l,变量 x

输出:切片 S,S 为语句和谓词的编号集合

变量说明:U 为待处理表,用于记录算法当前应该计算哪些变量的切片;V 为已处理表,用于保存哪些切片已经被处理过,以便在下次又遇到该切片计算时中止它,从而遵循前述的最小解原则。

```
begin slice
1  语法分析程序 P, 给语句和谓词标定编号, 生成程序执行流 flow 和谓词依赖集 PDS;
2  由 flow 生成 flow^R;
3  进行初始化工作: S = ∅, U = {(l, x)}, V = ∅;
4  while U≠
5    for U 中每个(l, x) do
6      U = U \{(l, x)}; V = V ∪{(l, x)};
7      if l 处是给 x 赋值的语句
8        S = S∪{l};
9        if 存在(l', l) ∈ flow^R then
10         for a 中的每个变量 v
11           if (l', v) | V then U = U∪{(l', v)};
12           end if
13         end for
14       end if
15     else if 存在(l', l) ∈ flow^R 且(l', x) ∉ V then
16         U = U∪{(l', x)};
17     end if
18     if l 被加入 S, 且在 PDS 中有 ⟨l, l_b⟩, 且 l_b ∉ S then
19         S = S∪{l_b};
20         if 存在(l', l) ∈ flow^R then
21           for [b]^l_b 中的每个变量 v do
22           if (l', v) | V then U = U∪{(l', v)};
```

```
23              end if
24            end for
25          end if
26        end if
27        end for
28    end while
  end slice
```

算法说明: 步骤 1、2 完成了程序的语法分析并生成了程序控制流 flow、$flow^R$ 和谓词依赖集 PDS; 步骤 3 完成了各分析表的初始化; 步骤 4 ～28 进行过程内切片的生成, 直至待处理表 U 为空, 其中步骤 7 ～14 处理赋值语句, 步骤 15 ～17 对于非赋值语句或者不给变量 x 赋值的赋值语句, 扫描点上移; 步骤 18 ～26, 若 l 被加入 S, 且存在谓词 l_b 决定 l 的执行与否, 则将 l_b 加入到 S 中, 此时若 $[b]^{l_b}$ 中还有变量, 则要标记还需要计算这些变量的切片, 即在 U 中加入这些变量的待处理标记 $\{(l', v)\}$。

算法的优点: 只用生成两个控制流, 即程序逆向流 $flow^R$ 和谓词依赖集 PDS, 就可以进行切片分析: 从切片点开始沿 $flow^R$ 扫描程序, 不断在切片集合中加入影响变量 x 在切片点取值的语句, 直到切片集合不能再扩大为止。该算法消除了传统 PDG、SDG 切片算法中不相关数据依赖的计算, 提高了切片的效率。

3 过程间切片方法

对于含有函数定义和函数调用的单线程程序, 其执行流仍然是顺序的, 所以切片的计算仍可以沿逆向程序流来进行。切片算法的框架与过程内切片算法框架一样, 只是增加了对函数调用、函数返回等控制流的处理。下面先给出包含函数定义和函数调用的程序语法, 用 BNF 表示如下:

function declare :: = p(形参列表)$\{ {}^{l_n} S \}^{l_x}$

a:: = … | [call p(实参列表)] $^{l_c}_{l_r}$

S:: = … |[call p(实参列表)] $^{l_c}_{l_r}$ |[return 语句] ^{l_x}

其中: "…"表示省略第 2 章已经给出的过程内的语法部分; l_n、l_x 分别表示函数的进入点和退出点; l_c、l_r 分别表示函数调用点和返回点。若函数定义中"}"的前面有"return 语句", 则不在"}"的右上角标记 l_x, 否则要标记。所有"return 语句"的右上角都要标记 l_x。一个函数内的所有 l_x 的值均不相同。

flow 需要在原来的基础上再加入 $(l_c; l_n)$ 和 $(l_x; l_r)$, 它们分别表示函数调用和函数返回, 中间的";"用来区别于过程内的 (l', l)。

计算切片时, 所要新加的处理是函数调用语句: 若函数调用处为赋值语句, 且赋值语句左式为待求其切片的变量, 则应该在过程内程序切片生成算法处理后, 再在 S 中并入所调用函数的集合 $\{l_x\}$, 并根据 l_x 处返回值的形式作不同处理: a) 若返回值为常数, 则不作处理; b) 若返回值为变量 v, 则 $U = U \cup \{(l', v) \mid$ 存在 $(l', l) \in flow^R$ 且 $(l', v) \mid V\}$, 其中 $(l', v) \mid V$ 的作用是禁止重复计算切片。

若函数调用处未使用函数返回值, 则仍按照过程内程序切片生成算法处理, 即扫描点上移。所要注意的是:

a) 待处理表 U 只有一个, 所以要分清局部变量和全局变量。当逆向分析到了某变量的定义处, 若待处理表 U 中仍有该变量的切片值未计算, 则应删除它, 以防止该变量的切片分析越出它的作用范围。这种情况也说明该变量未被初始化。

b) 增加一个数据结构 C 来记录函数调用情况。C 初始时为空。当分析到某函数调用时, 应该把 (l_c, l_n) 放入 C 中。而当逆向扫描到函数入口 l_n 时, 若 C 中存在对应的 (l_c, l_n), 则应该将待处理表 U 中的 (l_n, v) 转换为 (l_c, v), 并删除 C 中的 (l_c, l_n); 若 C 中不存在对应的 (l_c, l_n), 则应根据 $flow^R$ 中所有的 $(l_n; l_c')$, 把 (l_n, v) 转换为 (l_c', v)。其中 v 为全局变量或者形式参数, 若 v 为形式参数时, 则要把 v 转换为实参变量。

4 实例

为说明该切片算法的计算过程, 下面给出一个过程间切片的例子:

```
int count;
void m( ) { [1]
    [count = 0] [2];
    int [x = 2] [3], [v = 100 /x] [4]
    int [y = [f(v)] [5 6]] [7];
} [8]
int f(int n) { [9]
    int [s = 0] [10];
    while([n > 0] [11]) {
    [s = s + n] [12];
    [n = n - 1] [13];
}
    [count = count + 1] [14];
    [return s] [15];
}
```

计算 slice(7, y)

解: 依照文献[4] 第 2 章中给出的 flow 的定义, 首先生成程序流 flow, 如下:

flow = {(1,2), (2,3), (3,4), (4,5), (5;9), (6,7), (7,8), (9,10), (10,11), (11,12), (11,14), (12,13), (13,11), (14,15), (15;6)}

得程序逆向流:

$flow^R$ = {(2,1), (3,2), (4,3), (5,4), (9;5), (7,6), (8,7), (10,9), (11,10), (12,11), (14,11), (13,12), (11,13), (15,14), (6;15)}

谓词依赖集 PDS = {(12,11), (13,11)}。

切片的计算过程如表 1 所示。

表 1 程序切片计算的实例

step	待处理表 U	函数调用表 C	切片 S
初始	$\{(7, y)\}$		
1	$\{(14, s)\}$	$\{(5, 9)\}$	$\{7, 15\}$
2	$\{(11, s)\}$	$\{(5, 9)\}$	$\{7, 15\}$
3	$\{(10, s), (13, s)\}$	$\{(5, 9)\}$	$\{7, 15\}$
4	$\{(12, s)\}$	$\{(5, 9)\}$	$\{7, 15, 10\}$
5	$\{(11, n)\}$	$\{(5, 9)\}$	$\{7, 15, 10, 12, 11\}$
6	$\{(10, n), (13, n)\}$	$\{(5, 9)\}$	$\{7, 15, 10, 12, 11\}$
7	$\{(9, n), (12, n)\}$	$\{(5, 9)\}$	$\{7, 15, 10, 12, 11, 13\}$
8	$\{(5, v)\}$		$\{7, 15, 10, 12, 11, 13\}$
9	$\{(4, v)\}$		$\{7, 15, 10, 12, 11, 13\}$
10	$\{(3, x)\}$		$\{7, 15, 10, 12, 11, 13, 4\}$
11			$\{7, 15, 10, 12, 11, 13, 4, 3\}$

表的第 1 列表示计算时的执行顺序。整个计算过程不断循环处理待处理表 U, 直至 U 为空。表的第 1 ～11 行每一行表示对前一行中 U 处理后得到的分析结果。已处理表 V 在表 1 中未列出, 它初始时为 , 以后每一行的 V 为前一行 V 和 U 的并集, 当求解过程中遇到待求的切片已经在 V 中时, 就中止该切片的计算。第 11 行为计算结果: 切片 slice(7, y) = S = {7, 15, 10, 12, 11, 13, 4, 3}, S 中元素的顺序为其被加入的顺序。

(下转第 926 页)

3.2 不同角度测试策略的实例

针对 2.2 节中描述了几种不同角度的测试,这里仅以基于关联关系的测试为例进行详细说明。在该项目的第一阶段测试中期,发现 bug 总计 19 个。此时采用了横向展开测试,通过 bug 的测试者对每个 bug 进行说明和演示,使得其他测试人员了解这些 bug 的现象,然后在自己负责测试的功能模块中确认是否有同样或者类似的问题。在第一阶段测试结束时,又发现 bug 总计 28 个,其中有 15 个都是通过横向展开测试发现的,占其中的 54%。

随着测试工作的深入,测试人员对该大型软件也越来越熟悉。但是因为该软件功能强大,各个功能之间都有一定的联系,所以常常因为不知道这些联系而遗漏了一些测试。这种情况下纵向展开测试可以发挥比较好的效果。本次纵向展开测试具体实施是通过大家共同完成该系统的各功能模块的关联关系图的方式进行的。每个人贡献自己知道的一部分,合起来便完成了整个系统各功能模块的关联关系图。与此同时,各位测试人员对各种纵向关联关系也就清楚了。在接下来对第一阶段测试发现的 bug 进行确认测试过程中,通过这种纵向展开测试,又新发现了 14 个 bug。

通过实践基于关联关系的测试策略,充分验证了软件测试中的 80-20 原则。所以对于已经发现的 bug 进行充分的周边展开测试是非常有效的。

3.3 测试人员分布策略的实例

该项目的测试中,在第一阶段结束以后,测试人员大多都认为自己负责的部分该测的都已经测过了,应该不会再有问题了,因此出现了思想懈怠,同时因为长期面对一件事情,心理上也出现了反感的情绪。为了提高测试的质量,采用了人员交叉的分布策略,即交换每个人负责测试的功能模块,这个策略使得测试取得了出其不意的效果。例如,对于该软件的品质管理功能,在采用人员交叉策略之后,经过交叉人员的共同努力,第二阶段发现的 bug 达到了第一阶段的两倍。

通过这个策略的实施,每个人不仅接触了新的内容,更重要的是通过新旧结合产生了更多的组合观点,这个调整也进一步推动了纵向关联关系策略更加深入地进行。

另外,关于结对测试的策略,在该项目的测试中扩展为分组测试,即 2、3 人为一个组,组内确认 bug 无误后再向委托方提交。经过这个策略的运用,提交给委托方的软件问题,他们需要再次与测试人员进行确认的比率由最初的 60% 下降到后来的 15%,这样为双方的工作节约了大量的时间,很好地提高了工作效率。

4 结束语

为了提高软件测试的质量和效率,本文从三个不同方面提出了几个关于软件测试的策略。这些策略与具体的黑盒测试的各种方法相结合,使得软件测试更加系统化、灵活化,测试的效率和质量都会得到明显提升。目前自动化测试是软件测试领域蓬勃发展的一个分支。自动化测试带来的便利之处有目共睹,但是自动化测试也并非所有的测试都适用,所以对于如何有效地进行自动化测试将是需要继续研究的一个问题。

参考文献:

[1] ATTON R. 软件测试[M]. 北京:机械工业出版社,2004.

[2] 张海番. 软件工程导论[M]. 3 版. 北京:清华大学出版社,1998.

[3] 董晓霞. 相邻因素组合测试用例集的最优生成方法[J]. 计算机学报, 2007,30(2):200-210.

[4] 杨玲萍,韩阳. 基于功能点分析测试设计充分性模糊评判建模[J]. 计算机工程与应用,2007,43(3):106-111.

[5] 张永强,陈永革,姚立新. 军用软件的测试与实践方法[J]. 火力与指挥控制,2006,31(9):91-93.

[6] 朱海燕. 关于两两测试的研究[J]. 计算机工程与设计, 2006,27(15):2802-2804.

[7] 董晓霞. 软件测试工程化的研究和实践[J]. 计算机工程与设计, 2006,27(11):2008-2011.

(上接第 922 页)

5 结束语

针对传统 PDG、SDG 程序切片算法会计算无关数据依赖的缺点,本文提出了一种基于程序逆向流的切片算法。它事先生成两个控制流集合,即程序逆向流 $flow^R$ 和谓词依赖集 PDS,然后从切片点开始逆向扫描程序,只计算与切片相关的数据依赖,从而消除了 PDG、SDG 中不相关的数据依赖计算,提高了切片计算的效率。本文讨论了过程内切片算法的计算原理,并给出了其算法,在此基础上分析了过程间切片的算法。最后给出了一个 C 程序的切片实例,实例结果表明该方法具有一定的可行性和实用性。进一步的工作是把程序切片技术与程序验证技术结合起来,以消减无关代码,降低程序正确性证明的复杂性。

参考文献:

[1] EISER M. Program slicing: formal, psychological and practical investigations of an automatic program abstraction method[D]. Michigan: University of Michigan, 1979.

[2] OTTENSTAIN K J, OTTENSTAIN L M. The program dependence graph in a software development environment[J]. ACM SIGPLAN Notices, 1984,19(5):177-184.

[3] HORWITZ S, REPS T, BINKLEY D. Interprocedural slicing using dependence graphs[J]. ACM Trans on Programming Language and System, 1990, 2(1):26-60.

[4] 姜淑娟,徐宝文,史亮. 一种基于异常传播分析的数据流分析方法[J]. 软件学报,2007,18(4):832-841.

[5] WEISER M. Program slicing[J]. IEEE Trans on Software Engineering, 1984, 10(4):352-357.

[6] NIELSON F, NIELSON H R, HANKIN C. Principles of program analysis[M]. [S.l.]: Springer-Verlag, 1999.

[7] 李必信,郑国梁,王云峰,等. 一种分析和理解程序的方法——程序切片[J]. 计算机研究与发展,2000,37(3):284-291.

[8] TIP F. A survey of program slicing techniques[J]. Journal of Programming Languages, 1995,3(3):121-189.

[9] XU Bao-wen, QIAN Ju, ZHANG Xiao-fang, *et al*. A brief survey of program slicing[J]. SIGSOFT Softw Eng Notes, 2005,30(2):1-36.

[10] HARMAN M, HIERONS R M. An overview of program slicing[J]. Software Focus, 2001, 2(3):85-92.