

# 一种基于故障模式状态机的测试方法

肖 庆 杨朝红 毕学军

(装甲兵工程学院信息工程系, 北京 100072)

**摘 要:** 基于状态机对故障模式进行了统一的描述,使得故障描述更准确、无二义性。为了避免路径组合爆炸,提出基于控制流的状态集迭代分析算法进行故障检测,可以使算法的计算复杂性由  $O(P)$  ( $P$  是路径数目)减少为  $O((N+E)N)$  ( $N$  是控制流图节点数,  $E$  是控制流图边数)。由于状态机的独立性,对控制流图进行一遍迭代可以同时计算多个故障模式,大大提高测试效率。同时,该方法还采用了带条件的状态计算可以较好地减少误报的情况。

**关键词:** 软件测试; 静态分析; 数据流分析; 故障查找

**中图分类号:** TP311.5

## 引 言

软件测试的手段很多,基本上从是否要实际运行被测软件来分,可分为动态测试和静态测试。动态测试需要实际运行被测软件,而静态测试则相反。本文主要讨论的是一种面向故障的静态分析方法。

近年来,面向故障的静态分析测试技术得到快速的发展,大量的软件测试工具被研制出来,并且在对一些大型商业软件和开源软件的测试中发现了大量的以前测试没有发现的软件故障和安全隐患。例如,SDV 在对 Windows 操作系统 126 个使用多年的驱动程序测试中发现了 65 个故障,包括 12 个严重故障<sup>[1]</sup>;ExPLODE 在一些常用的文件存储系统发现了大量的严重故障<sup>[2]</sup>;MC 在 Linux、OpenBSD 和 Xok exokernel 软件中发现了近 500 个故障<sup>[3]</sup>以及 100 多个安全漏洞<sup>[4]</sup>。FindBugs 在 Eclipse、J2SE 和 JBoss 等开源软件中发现上百的故障<sup>[5]</sup>。与形式验证试图证明整个软件没有故障不同,面向故障的静态分析技术仅试图找出软件中某类故障,如果检测算法是完全的,则可以证明该软件不存在该类故障。

## 1 基本概念

### 1.1 故障模式

所谓的故障模式,就是总结那些经常出现、并具有一定模式的故障。故障模式通常由具有领域程序

设计经验的人或者测试人员总结出来的,如下面的一些故障模式:在 C 语言程序中的一条执行路径上一个指向动态内存的指针释放后再使用将造成访问悬挂指针故障;在 Java 语言程序中实现 clone() 时需调用 super.clone() 否则造成一个故障;在 linux 内核程序中如果当前禁用中断则不能调用任何可能造成阻塞的操作,否则造成一个故障,等等。

### 1.2 控制流图

控制流图是对程序控制结构的图形表示,它是一种有向图。通常一个程序的控制流图可表示为  $(N, E, \text{Entry}, \text{Exit})$ 。其中  $N$  代表节点的集合,反映程序中的简单语句和复合语句的条件判断;  $E$  代表有向边的集合,反映程序中语句间的控制流关系; Entry 为程序的固定的唯一入口节点; Exit 为程序唯一的退出节点。

### 1.3 故障描述状态机

由于故障模式涉及应用领域的逻辑和语义范畴,因此故障模式的种类繁多,也有一定的主观性。为了保证后续检测算法执行的无二义性,首要的工作就是,需要对故障模式进行统一的描述,使得故障描述更准确、无二义性。在已有的静态分析工具中,被广泛采用的方法是故障描述状态机<sup>[6-7]</sup>。状态机是对程序语义一种常用和易于理解的抽象表示,故障描述状态机实际上给出了一系列状态和状态间的转换条件。对于前面提到的有关悬挂指针故障可以转化为一个简单的状态机,如图 1 所示。其中的 free( $p$ )指的是释放指针  $p$  指向的内存空间, dereference( $p$ )指的是对指针  $p$  指向的内存空间进行访问。整个状态机所描述的含义为:对于任何指向动态分配

收稿日期: 2007-05-09

基金项目: 国家“863”计划(2006AA01Z184)

第一作者: 男, 1979 年生, 讲师

E-mail: xqing@sina.com

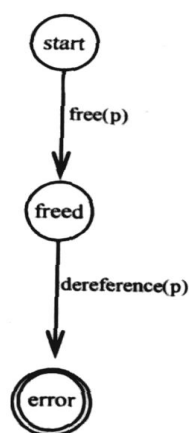


图 1 悬挂指针故障状态机

Fig. 1 State machine of dangling pointer

内存的指针  $p$ , 起始进入  $start$  状态; 当遇到  $free(p)$  函数调用时即转入  $freed$  状态; 在  $freed$  状态下碰到对  $p$  指向的内存进行任何访问操作时则进入终态 —  $error$  状态。

又如, “在 linux 内核程序中如果当前禁用中断则不能调用任何可能造成阻塞的操作”, 也可以转化为图 2 所示状态机。

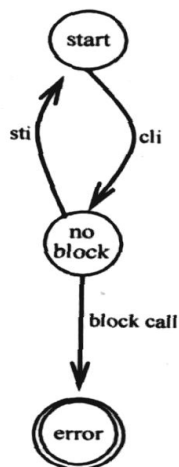


图 2 不合适的阻塞操作调用故障状态机

Fig. 2 State machine of improper block operation invoked

#### 1.4 带条件的状态计算

有了故障描述状态机后, 分析的过程就是沿着控制流按照故障状态机给出的条件进行状态计算, 如果发现在控制流图的任何节点上出现故障模式定义的故障状态就报告一个故障。

但是静态分析的一个需要解决的重要问题是误报问题 (false positive)。引起误报的一个主要原因是程序中存在大量不可达路径<sup>[6]</sup>。由于不可达路

径的存在导致许多沿这些路径的计算都是无效的。为了减少误报本文引入“带条件的状态”。每一个状态都具有一个前提条件, 该前提条件由一系列变量及其取值区间组成 (当前只考虑数值型变量)。如:

```

void f(int x) {
    int * p = (int *) malloc(sizeof(int));    n1
    if (x > 0) {                                n2
        free(p);                                n3
    }                                            n4
}
  
```

图 3 给出了一个简单的程序片断及其对应的控制流程图。对应图 1 状态机, 图 3 右边给出了沿控制流进行的状态计算。在  $n1$  和  $n2$  节点上状态集合中包含一个可能的状态即  $start$  状态, 状态的条件为  $x$  的取值范围  $[-MaxInt-1, MaxInt]$ 。在  $n3$  节点上由于  $n3$  节点对应的语句为  $free(p)$ , 因此满足状态机状态转换条件,  $start$  状态应转换为  $freed$  状态, 另外由于分支判断条件  $x > 0$  的限制, 在  $n3$  节点上的状态集合变化为  $\{freed(x: (0, MaxInt))\}$ 。一旦某个状态的条件中出现某个变量的区间为空的时候, 称该状态为无效状态。去除无效状态实际上就是排除那些矛盾的控制流组合。

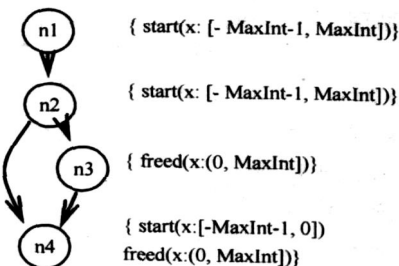


图 3 带条件的状态计算

Fig. 3 Conditional state computing

## 2 面向控制流的故障查找算法

为了简化描述, 本文仅考虑单个函数内部的控制流分析。在现有的一些故障分析系统中, 采用的故障查找算法大都是基于路径, 即首先产生一条路径, 然后针对该路径的控制流根据故障描述状态机进行状态分析<sup>[6-7]</sup>。该类算法的复杂性至少是  $O(P)$  ( $P$  是路径数目)。

为了提高分析效率, 结合数据流分析的思想, 本文提出对控制流程图进行迭代的方法, 核心是在控制流图上对当前节点可能的状态集合进行迭代。

数据流分析中,数据流信息通过建立和解方程来收集,方程联系程序不同点的信息。在本算法中要计算的状态集合是沿着控制流正向进行的,因此基本的方程形式为<sup>[8]</sup>:

$$\begin{aligned} in[n] &= \bigcup_{p \in pred[n]} out[p] \\ out[n] &= gen[n] - kill[n] \end{aligned}$$

其中的几个集合含义解释为:

$in[n]$ :到达节点  $n$  之前的所有可能状态集合。

$out[n]$ :到达节点  $n$  之后所有可能状态集合。

$gen[n]$ :节点  $n$  中新产生或改变得到的新状态集合。

$kill[n]$ :节点  $n$  中“注销”或“被改变”的状态集合。

$pred[n]$ :节点  $n$  的所有前驱节点集合。

上述方程组中的  $gen[n]$  和  $kill[n]$  一旦确定则即可通过迭代的方法求出控制流图中每个节点的  $in[n]$  和  $out[n]$  集合。迭代算法如算法 1 所示。

算法 1:计算状态集合

输入:程序控制流图和故障模式描述状态机

输出:每个语句的  $in[n]$  和  $out[n]$ 。

begin

for 每个节点  $n$  do  $in[n] = out[n] = \emptyset$ ; /\* 初始化 \*/

$out[Entry] = \{ start(\dots) \}$  /\* 入口节点的初始状态集合 \*/

change := true;

while change do begin

change := false;

for 除 Entry 的每个节点  $n$  do begin

$in[n] := \bigcup_{p \in pred[n]} out[p]$ ;

$oldout := out[n]$ ;

$out[n] := gen[n] - (in[n] - kill[n])$ ;

if  $out[n] \neq oldout$  then change := true

end

end

end

### 3 举例

下面用一个例子来说明整个算法的执行过程。设有一段程序如下,对于图 1 给出的状态机描述:

```
void f(int x) {
    int * p = (int *) malloc(sizeof(int));    n1
```

```
    if (x > 0) {                                n2
        free(p);                               n3
    }                                           n4
    if (x < 0) {                                n5
        *p = 2;                                n6
    }                                           n7
        *p = 3;                                n8
    }
```

则迭代结束后最终的  $in$  和  $out$  集合结果如图 4 所示。

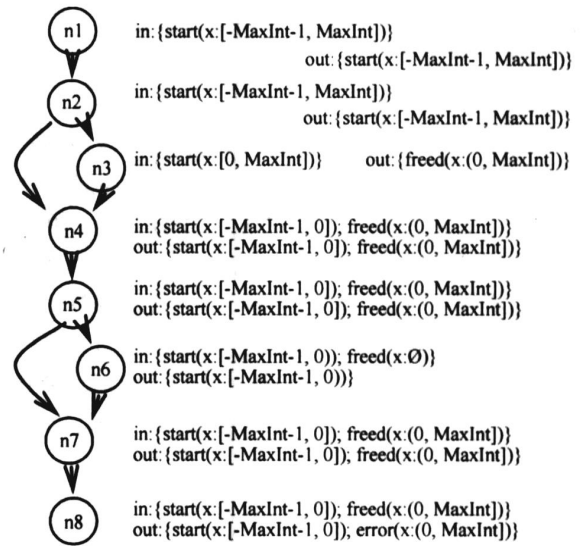


图 4 算法举例

Fig. 4 An example

如果不考虑条件,在  $n6$  节点中, $in$  状态集合中有一个状态为  $freed$  状态,而  $n6$  节点对应的语句为 ( $*p = 2$ ),即对  $p$  指向的内存进行了使用,根据状态机描述  $freed$  状态要转换为  $error$  状态并报告一个故障,但事实上该报告为一个误报。而在考虑条件后, $freed$  状态的条件为 ( $x: \emptyset$ ),意味着该状态为无效状态,会被丢弃,这样就避免了误报。实际上从另外一个角度分析:如果在  $n6$  节点中,状态机要到达  $freed$  状态,则要求第一个  $if$  语句条件  $x > 0$  为真,第二个  $if$  语句条件  $x < 0$  也为真。而这两个条件是互相矛盾的,也就是说这个控制流组合是不可达的。可以看出,算法中对状态条件的计算实际上就是对应了面向路径算法中对不可达路径的判断,因此引入带条件的状态计算可以较好地减少误报的情况。

### 4 关于算法的性质讨论

对于算法 1 这类数据流迭代算法,算法最终会

停止。因为所有定义的集合是有穷的,所以最终一定会有`while`循环,使得对每个 $n$ 都有`oldout = out[n]`,于是,`change`将保持为`False`,从而算法会停止。最坏情况下,算法的时间复杂度是 $O(n^4)$ ,但如果按照深度优先顺序安排节点的计算顺序,则在实际程序上迭代的平均数将小于5,这样算法1的效率实际是比较高的<sup>[8]</sup>,在平均的情况下该类算法的复杂度为 $O((N + E)N)$ <sup>[8]</sup>。上述讨论的例子虽然只针对了一个故障模式,但由于状态机的独立性,实际上对控制流图进行一遍迭代可以同时计算多个故障模式,因此可大大提高测试效率。

## 5 结束语

MC和ESP系统的分析过程是基于路径进行的,由于程序中的路径数通常是巨大的,因此笔者认为这种方法效率不高。FindBugs、PMD等一些开放源码的项目针对特定的具体模式进行分析,这些工具并没有统一的故障模式查找方法,其检测算法都依赖于具体故障模式,对控制流的分析能力也比较弱。本文提出的这种面向控制流进行迭代的故障查找方法并不依赖于具体的故障模式,而其复杂度要大大低于MC和ESP系统。

## 参考文献:

- [1] BALL T, BOUNIMOVA E, COOK B. Thorough static analysis of device drivers[C]. EuroSys, 2006: 73 - 85.
- [2] YANG J, SAR C, ENGLER D. Explode: a lightweight, general system for finding serious storage system errors[C]. OSDI, 2006: 131 - 146.
- [3] ENGLER D, CHELF B, CHOU A, et al. Checking system rules using system-specific, programmer-written compiler extensions[C]. The Fourth Symposium on Operating Systems Design and Implementation, 2000: 1 - 16.
- [4] ASHCRAFT K, ENGLER D. Using programmer-written compiler extensions to catch security holes[C]. IEEE Symposium on Security and Privacy, 2002: 143 - 159.
- [5] HOVEMEYER D, PUGH W. Finding bugs is easy[J]. ACM SIGPLAN Notices, 2004, 39(12): 92 - 106.
- [6] HALLEM S, CHELF B, XIE Y, et al. A system and language for building system-specific, static analyses[C]. PLDI, 2002: 69 - 82.
- [7] DAS M, LERNER S, SEIGLE M. Path-sensitive program verification in polynomial time[C]. PLDI, 2002: 57 - 68.
- [8] AHO A V, SETHI R, ULLMAN J D. Compilers: principles, techniques, and tools [M]. Beijing: Posts & Telecom Press, Pearson Education, 2002: 608 - 633.

## Study of a fault pattern state machine based testing method

XIAO Qing YANG ZhaoHong BI XueJun

(Department of Information Engineering, Academy of Armored Forces Engineering, Beijing 100072, China)

**Abstract:** This paper uses state machines to give a formal and unified description of fault patterns. Then, a unified testing method based on iteration of state set is proposed to avoid the path-explosion problem. The algorithm's computing complexity is  $O((N + E)N)$  ( $N$  is the number of nodes in control flow graph,  $E$  is number of edges in control flow graph). Because of the independency of the state machine, one traversal of control flow can test many fault patterns, therefore, testing efficiency can be improved. Moreover, the paper uses conditional state computing to reduce the problem of false positives.

**Key words:** software test; static analysis; dataflow analysis; error detection