内存泄露静态检测模型

柯 平[1] 宫云战[1] 杨朝红[1,2]

(北京邮电大学网络与交换技术国家重点实验室 北京 100876)[1]

(装甲兵工程学院信息工程系 北京 100072)[2]

摘 要 内存泄漏故障是程序中某处申请的内存空间,没有释放或没有完全释放或多次释放,是程序中常见的故障,极易导致系统崩溃。从面向具体错误的测试思想出发,采用静态测试的方法,给出了内存泄漏的静态检测模型,同时还引入了区间运算来减少内存泄漏的误报率和漏报率。根据以上模型实现了一个自动测试工具,并已用于软件测试。

关键词 内存泄露,区间运算,静态测试,故障模型,别名分析

中图法分类号 TP311.5 **文献标识码** A

Common Model for Detecting Invalid Arithmetic Operation

KE Ping[1] GONG Yun-zhan[1] YANG Zhao-hong[1,2]

(State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China)[1]

(Department of Information Engineering of the Academy of Armored Force Engineering, Beijing 100072, China)[2]

Abstract Memory leak is a kind of error which does not free the memory apply by the program completely, or free the same memory more then one time. It is a common error which may cause system collapse. Based on the defect-oriented test strategy and combining the advantage of static analysis, the paper gave a static analysis model to analyse memory leak, and also added interval analysis to this model to reduce the rate of misinformation and omission. This model has been implemented and used in software testing.

Keywords Memory leak, Interval computation, Static detection, Fault mode, Alias analysis

1 引言

软件测试贯穿于软件生存期的各个阶段,软件测试方法也多种多样,比如:根据是否运行程序,可以分为静态测试与动态测试;根据对代码是否可见,可分为黑盒测试和白盒测试。近几年,随着人们对软件及软件错误认识的不断深入,产生了许多新的测试方法。比如:针对一种或几种故障进行专门的测试——面向具体错误类型的测试,该方法针对性强,可以避免其他测试方法对"小概率"故障检测效率比较低的情况[2]。

内存泄漏问题常常困扰着C程序员,当前有很多针对内存泄漏的动态测试工具。但实际中它们并不能解决所有的问题。首先,这些动态工具大都是基于插装技术,因而不可避免地会减低程序的运行效率;其次,某些软件运行于特殊环境下,而这些工具无法支持;另外它们不能发现没有执行到的路径上的故障。解决这些问题恰恰是静态方法所擅长的。

本项目针对内存泄漏故障进行总结、构建模型,使用静态测试的方法对程序进行检测,同时我们还在检测中引入区间运算来减少误报率和漏报率。

2 C语言的内存泄露故障分析

在C语言中,设在程序的某处申请了大小为 M 的空间,凡在程序结束时 M 或者 M 的一部分没被释放,或者多次释放 M 或 M 的一部分都是内存泄漏故障。

C语言中内存泄漏有三种情况:

- 遗漏故障:是指申请的内存没有被释放;
- 重复释放:申请的内存被多次释放;
- 不相等的释放错误:是指释放的空间和申请的空间大小不一样。

在本项目中用一类故障状态机描述:考虑程序中的所有已分配的内存,为每个分配的内存创建一个变量无关和路径相关的状态机实例,并为每个内存维护一个指向该内存的变量的别名集合,如果通过别名集合中的任何一个变量调用释放操作,则内存被正常释放,但如果被释放的内存对应的别名集合中的变量再次被释放就会报一个故障。如果别名集合为空,则说明当前内存已经超出作用域且没有显示释放,因此报告一个故障。

3 检测算法描述

3.1 抽象语法树(Abstract Syntax Tree, AST)

程序的语法树是一种树的数据结构,该数据结构充分地说明按照语法怎样看待程序的各个部分,语法树可以通过语法分析过程得到。完全符合文法规格要求的语法树(分析树)对于进一步处理并不是最适合的,实际中,经常会根据需要对语法树做适当的简化和修改,这种修改后的形式,称为抽象语

到稿日期:2008-10-10 本文受国家"863"计划(2006AA01Z184),国家"863"重点项目(2007AA010302)资助。

柯 平(1982-),男,硕士生,主要研究方向为软件测试,E-mail:molaokeping@163.com;宫云战(1962-),男,教授,CCF会员,主要研究方向为软件测试、容错计算;杨朝红(1976-),男,讲师,CCF会员,主要研究方向为软件测试、容错计算。

法树 AST。

本项目中的 AST 生成器是基于一个开放源码的编译器生成工具 JavaCC[4] 之上，JavaCC 提供三个工具：

1)jjtree 用来处理. jjt 文件，生成抽象语法树节点定义代码和 jj 文件；

2)javacc 用来处理语法文件(. jj)生成分析代码；

3)jjdoc 根据. jj 文件生成 bnf 范式文档(html)。

3.2 控制流图(Control Flow Graph，CFG)

控制流图对大多数程序分析和结构测试都是必不可少的基础数据结构。控制流图是一种反应程序逻辑控制流程的有向图。它的定义为：通常一个函数的控制流图可表示为(N, E,$Entry$,$Exit$)。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以及控制流汇合点等，E 代表有向边的集合，反映程序中语句间的控制流关系。$Entry$ 为函数的固定唯一入口节点，$Exit$ 为函数唯一的退出节点。简单通俗地说：控制流图即是具有单一的、固定的入口节点和出口节点的有向图。对于非单入口和单出口的程序，可以通过人为添加一个统一入口和出口的方法解决。

在控制流图中有一类特殊的路径，叫不可达路径，它是指在控制流图上那些组合条件矛盾的路径。不可达路径是造成误报和漏报的一个重要原因，因为不可达路径是永远不可能被执行的，所以不可达路径上的操作应该被忽略。在控制流图中加入变量的取值区间可以有效地检测出不可达路径。例如：

```
void main(){
    int a = 0;
    int * b = (int * )malloc(sizeof(int) * 100);
    if(a ! = 0) free(b);
}
```

程序 1　不可达路径示范程序

在此例中，a 的值一直是 0，第 4 行中 free(b)不会执行，因此会产生内存泄露，如果不能检测到 free(b)是不可达路径，则会认为该内存已被释放，会产生漏报。如果我们在流程图中记录相关变量的取值区间，那么就可以知道 if 语句中的 a 的取值区间是[0,0]，就可以检测到 free(b)在不可达路径上，不会被执行，从而能检测出内存泄露。因此判定不可达路径可以减少漏报。同样，判定不可达路径也可以减少误报，在此就不举例说明了。程序中不可达路径的判断需要区间预算的支持。

3.3 区间运算

区间运算的目的是对程序中出现的变量和表达式的取值范围进行跟踪，为进一步的程序分析提供精确的信息支持。所谓变量 v 的区间，就是指变量 v 的取值范围。如变量 v 的取值范围是 $1.5 \leqslant v \leqslant 4.3$，那么用区间表示 v 为：[1.5,4.3]。特别地，一个常数 x，可以表示为：$[x,x]$。

下面定义基本的区间运算。假设区间 A 为$[a,b]$，区间 B 为$[c,d]$，其中 $a \leqslant b, c \leqslant d$，则区间四则运算定义为[5]：

$A+B = [a+c, b+d]$

$A-B = [a-d, b-c]$

$A \times B = [\min(a \times c, a \times d, b \times c, b \times d), \max(a \times c, a \times d, b \times c, b \times d)]$

$A / B = [a,b] \times [1/d, 1/c]$，且 $0 \notin [c,d]$

另外，引入一个区间集的概念，它表示一个或多个不相交的区间的集合。

在我们的系统中对区间的表示做了一些修改以适应计算机中的表示与运算。与数学中的区间表示主要有 2 个不同点：

1)在计算机中取消开区间，所有的区间都是闭区间

在计算机中，数的精度是有限的，无法表示开区间无限接近且不相等某个数 a，所以我们用 $a-\lambda$ 或 $a+\lambda$ 来替代，其中 λ 为计算机能表示的最小精度为 λ。因此，(a,b)在我们的系统中将表示为$[a+\lambda, b-\lambda]$。

2)增加整数区间的概念

区间在数学上表示的是实数的集合。但在程序中变量的取值范围往往是离散的整数值，因此增加整数区间的概念，它在区间上的取值是离散的。比如整数区间[1,5]表示的就是 1,2,3,4,5 这 5 个数。

3.4 别名分析

为了更准确地分析和检测内存泄露故障，我们引入了别名分析。当两个或多个指针变量同指向一个内存空间时，我们说它们互为别名。互为别名的变量构成一个别名集合[6]。

在本项目中，为每一个状态机维护一个别名集合，别名集合中存放着指向该内存空间的变量，通过遍历控制流图，来时时改变别名集合中的变量。

由于实际程序中指针存在的复杂性，故只考虑下面几种操作的别名计算：

1)内存分配操作：产生状态机，并将对应的变量放入别名集合中，并开始别名集合分析。

2)和内存块相关指针变量的赋值操作($p=q$;)。对于考虑 p,q 与当前别名集合的所属关系，如表 1 所列。

表 1　别名集合判断表

	q 属于	q 不属于
p 属于	集合不变	从集合中去除 p
p 不属于	将 p 加入集合	集合不变

3)指针运算：当指针进行++、--运算时，会将该指针从所在的别名集合中去除。

4)作用域变化。从集合中去除那些生存范围结束的成员。

对于某块内存 M，进入节点 S 时的别名集合为 $IN(S)$，节点 S 上生成的 M 的别名集合记作：$Add\ (S)$；节点 S 上去除的 M 的别名集合记作：$Kill\ (S)$，则有走完节点 S 的别名集合 $OUT(S)$为：$OUT(S) = IN(S) + Add(S) - Kill(S)$；当别名集合为空时，说明有一块内存没有变量指向它，无法再被使用，就会产生内存泄露故障。

3.5 故障描述状态机

用故障模式状态机对故障进行描述，可以使描述更加准确、无二义性。状态机是对程序语义一种常用和易于理解的抽象表示，主要是给出了一系列状态和状态间的转换条件。它包含一组状态集、一个起始状态、一个或多个终止状态、一个状态间的转换表。此外我们还增加了一个错误状态，用来表征程序出现故障。以下是内存泄露状态机的描述：

状态集合：{Start,Allocated,FREE,ERROR,End}

其实状态：Start

结束状态:End

错误状态:ERROR

状态转移表如表 2 所列。

表 2　状态转移表

序号	状态转移	引起转移的条件
0	开始进入 Start 状态	状态机启动后进入 Start 状态
1	Start→Allocated	程序申请了一块内存空间
2	Allocated→FREE	程序释放了申请的内存空间
3	Allocated→ERROR	内存空间对应的别名集合为空
4	Allocated→End	内存空间被 return 到方法之外
5	FREE→ERROR	对已释放的空间再次释放
6	FREE→End	已释放的内存空间对应的别名集合为空
7	ERROR→End	检测完毕,退出状态机

状态转移图如图 1 所示。

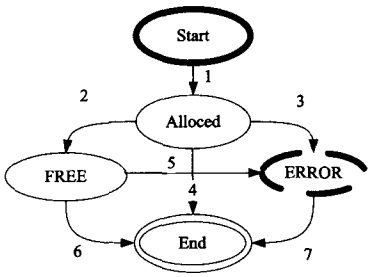

图 1　状态转移图

3.6　运行检测算法

系统的总的运行流程如图 2 所示。

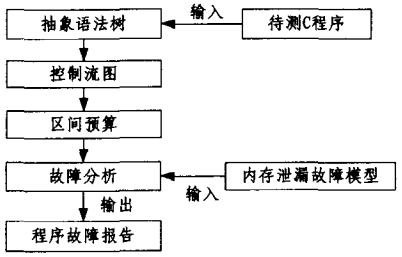

图 2　系统流程图

本项目中的故障分析部分是读入故障模型,生成故障描述状态机,然后根据状态机对程序进行分析。程序中每个内存申请语句都会产生一个独立的故障状态机。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和转换,当故障状态机走到错误状态,即 ERROR 状态时,就报告一个故障。对控制流的分析以函数为单位进行。

具体分析过程描述如下:

1)在函数控制流入口处,对函数中每个内存申请语句创建一个独立的状态机实例。

2)沿着控制流正向前进,在控制流图的每个节点上对状态机进行别名分析,同时调用相关函数,检测状态机转换条件是否满足,如果满足就进行状态转换。

3)在使用状态机检测的过程中,如果该节点进入"ERROR"状态,则报告一个内存泄漏故障。

4)当状态机进入 End 状态后销毁状态机。

4　实验结果及分析

目前在市面上出现了许多软件测试工具,包括开源的和商业化的。但通过静态方法测试 C 语言内存泄漏的工具不多。比如著名的测试工具 DevPartner 等都是用动态的方式来检测内存泄漏的。为了比较实验结果,我们针对不同的情况,构造下面的函数,来对本项目进行检验。

```
1. int * p;
2. void f1(){
3.  int * memleak_error1=NULL;
4.  memleak_error1=(int * )malloc(sizeof(int) * 100);
5.  free(memleak_error1);
6. }
7. void f2(){
8.  int * memleak_error2=(int * )malloc(sizeof(int) * 100);
9.  if(! memleak_error2)return;
10.  free(memleak_error2);
11. }
12. void f3(){
13.  int * memleak_error3=NULL, * memleak_error4=NULL;
14.  memleak_error3=(int * )malloc(100);
15.  memleak_error4=(int * )malloc(10);
16.  if(! memleak_error3&&! memleak_error4){
17.        return;
18.  }
19.  free(memleak_error3);
20.  free(memleak_error4);
21. }
22. void f4(){
23.  int * memleak_error5 = NULL, * not_memleak_error5 =
NULL;
24.  memleak_error5=(int * )malloc(100);
25.  p=memleak_error5;
26. }
27. int * f5(){
28.  int * memleak_error6=NULL;
29.  memleak_error6=(int * )malloc(100);
30.  if(memleak_error6 ! = NULL){
31.        return memleak_error6;
32.  }
33. }
34. void f6(){
35.  int * memleak_error7=NULL;
36.  if((memleak_error7=(int * )malloc(100))! =NULL){
37.        free(memleak_error7);
38.  }
39. }
40. void f7(int i){
41.  int a[
42. 5]={4,4,4,4,4,};
43.  int * memleak_error8=NULL;
44.  if(i>0){
45.        memleak_error8=a;
46.  }else{
47.        memleak_error8=(int * )malloc(100);
48.  }
49.  if(i<=0)
50.        free(memleak_error8);
51. }
52. void f8(){
53.  int * memleak_error9=NULL;
54.  memleak_error9=(int * )malloc(100);
55.  p=memleak_error9;
```

```
56.  memleak_error9++;
57.  free(p);
58.  }
59.  void f9(){
60.   int *q=NULL;
61.   int* memleak_error10=NULL;
62.   memleak_error10=(int *)malloc(100);
63.   q=(int *)memleak_error10;
64.   free(q);
65.  }
```

程序 2　被测程序范例

表 3　测试结果

泄漏内存申请时的变量	替换前	替换后
memleak_error1	Y	N
memleak_error2	Y	N
memleak_error3	Y	N
memleak_error4	Y	N
memleak_error5	Y	N
memleak_error6	Y	N
memleak_error7	Y	Y
memleak_error8	Y	N
memleak_error9	Y	N
memleak_error10	Y	N

下面的函数中每个 memleak_error 变量都有内存泄漏，如果将/* */中的语句替换成之前的语句，则内存泄漏故障就都消除了。在测试结果表(如表 3 所列)中分别列出了每个 memleak_error 变量在程序替换前和替换后的测试结果。其中"Y"表示报出了故障，"N"表示没有报出故障。

从测试结果看出，替换前的 10 个故障点全部检测出来，而替换后的 10 个非故障点有一个误报，误报率是 10%。因此可以看出本项目的检测效果还是非常好的。

结束语　本文提出基于区间运算的内存泄漏静态的检测模型，并对其检测效果进行了测试，测试结果表明该模型能较为准确地检测出方法内的内存泄漏故障，而基于区间运算的静态检测能有效地减少漏报率和误报率。但目前系统还仅限于方法内的故障检测，不能做方法间内存泄漏的检测，因此在现实的检测中有一定的限制，需要进一步的改进。

参 考 文 献

[1] 宫云战. 软件测试的故障模型[J]. 装甲兵工程学院学报，2004，18(2)：1-5

[2] 宫云战. 软件测试[M]. 北京：国防工业出版社，2006

[3] 肖庆. 内存泄漏的一种静态分析方法[J]. 装甲兵工程学院学报，2004，18(2)：23-26

[4] https://javacc.dev.java.net/

[5] 王德人，张连生，邓乃扬. 非线性方程的区间算法[M]. 上海：上海科学技术出版社，1987

[6] 周高嵌. 基于白箱测试的源代码在线评测系统[D]. 北京：北京化工大学，2005

(上接第 139 页)

口负责接收许可证、更新、认证等服务器端的命令和本地用户的播放、设备配置等命令，并转换成逻辑程序的事务(查询)命令。客户端的许可证存放在许可池中，许可池本身由管理许可证规则管理。用户播放数字内容的过程如下：用户首先向设备提出播放命令，由设备接口将该命令转换成相应的逻辑事务并进入该许可证的使用控制过程，否则拒绝该命令。

通过本文的转换方式，内容服务器、版权服务器和版权控制器之间的交互可使用 ODRL 且与相关使用 ODRL 的 DRM 系统进行互操作。

结束语　缺乏正式语义使基于 XML 的 ODRL 等 REL 语言的确切含义严重依赖特定理解，并容易产生二义性和不确定性。本文在我们提出的分布式使用控制和权利描述语言 LucScript 语言的基础上，实现 ODRL 语言与 LucScript 语言转换和互操作机制。由于 LucScript 不仅包含正式的逻辑语义，而且包含相应的逻辑实施框架。将 ODRL 语言转换成 LucScript，为 ODRL 提供正式逻辑语义，并提供 ODRL 权利保护策略实施的逻辑框架，形成 ODRL 权利保护策略表达和实施的统一逻辑框架，为 ODRL 权利保护策略实施提供可信和形式化分析基础。

下一步将对 XrML 语言与 LucScript 语言的转换和互操作机制进行研究。

参 考 文 献

[1] 俞银燕，汤帜. 数字版权保护技术研究综述[J]. 计算机学报，2005，28(12)：1962-1968

[2] Becker M Y，Fournet C，Gordon A D. Design and Semantics of a Decentralized Authorization Language// 20th IEEE Computer Security Foundations Symposium (CSF). 2007：3-15

[3] Chong C N，Corin R，et al. LicenseScript：a logical language for digital rights management. Annales des Telecommunications，2006，61(3/4)：284-331

[4] Pucella R，Weissman V. A Formal Foundation for ODRL // Proc. Workshop on Issues in the Theory of Security. 2004

[5] Halpern J，Weissman V. A formal Foundation for XrML// Proc. 17th IEEE Computer Security Foundations Workshop. 2004：251-265

[6] Holzer M，Katzenbeisser S，Schallhart C. Towards a Formal Semantics for ODRL // Proc. 1st International Workshop on ODRL. 2004：137-148

[7] Gunter C A，Weeks S T，Wright A K. Models and languages for digital rights // Proc. of the 34th Annual Hawaii International Conference on Systems Sciences. Maui，Hawaii，2001：4034-4038

[8] Garcia R，Gil R，Gallego I，et al. Formalizing ODRL Semantics using Web Ontologies// Proc. 2nd Intl. ODRL Workshop. 2005：1-10

[9] Alapan A，Andrew H. Extending ODRL to Enable Bi-Directional Communication// Proceedings of the 2nd International ODRL Workshop. Lisbon，Portugal，2006：43-52

[10] Alapan A，Andrew H. Extending ODRL and XrML to Enable Bi-directional Communication. Technical Report CS04-28-00. Cape Town，South Africa：Department of Computer Science，University of Cape Town，2004

[11] Zhong Y，Zhu Z，Lin D M，et al. A Method of Fair Use in Digital Rights Management// Proc. of the 10th International Conference on Asian Digital Libraries. LNCS 4822. Hanoi，Vietnam，2007：160-164

[12] Bertino E，Catania B，Gori R. Active-U-Datalog：integrating active rules in a logical update Languages// Lecture Notes in Computer Science. 1998：107-133

[13] 钟勇，秦小麟，郑吉平，等. 一种灵活的使用控制授权语言框架[J]. 计算机学报，2006，29(8)：1408-1418