基于指针映射集的动态内存故障测试方法研究

张 威[1),2)] 宫云战[2)] 卢庆龄[1)] 万 琳[1)]

[1)](装甲兵工程学院信息工程系 北京 100072)

[2)](北京邮电大学网络与交换技术国家重点实验室 北京 100876)

摘 要 动态内存故障在使用指针的程序中是普遍存在的,采用动态测试方法进行测试难以准确定位故障源.而现有的静态分析方法主要存在漏报和误报过多的情况.针对这些问题,提出了指针映射代数系统的概念,全面地反映了指针与内存之间的映射关系,并给出了面向不同故障的指针映射集的构造规则,以此为基础建立了动态内存故障模型.通过指针映射集和故障模型,可以自动检测内存释放异常、内存泄露和空指针引用等动态内存故障,提高了测试效率.在分析过程中,还综合应用了控制流图和路径条件,提高了测试结果的精度.实验结果表明,该方法能够有效检测动态内存故障,而且由于规则定义较为全面,漏报和误报率也较低.

关键词 软件测试;静态分析;指针映射集;内存泄漏;空指针引用

中图法分类号 TP302 **DOI 号**: 10.3724/SP.J.1016.2009.02274

Research on Dynamic Memory Faults Testing Method Based on Pointer Mapping Sets

ZHANG Wei[1),2)] GONG Yun-Zhan[2)] LU Qing-Ling[1)] WAN Lin[1)]

[1)](*Department of Information Engineering, Academy of Armored Force Engineering, Beijing* 100072)

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

Abstract Dynamic memory faults are ubiquitous in the program with pointers. It is difficult to locate faults sources adopting dynamic testing method. Static analysis methods nowadays often miss some faults and produce too many false alarms. Considering of these problems, this paper puts forward the notion of pointer mapping algebra system that reflects the mapping relationship of pointer and memory completely, and gives the construction rules of pointer mapping sets for different faults class, and then establishes dynamic memory faults model. Through pointer mapping sets and fault model, it can detect bad deallocation, memory leak and null pointer dereference faults automatically and increase the testing efficiency. In the process of analyzing, it adopts synthetically control flow chart and path condition in order to increase the precision of testing results. Results of experimentation show that this method can detect dynamic memory faults effectively. Since rule definition is general, the probability of missing faults and producing false alarms is lower.

Keywords software testing; static analysis; pointer mapping sets; memory leak; null pointer dereference

---

收稿日期:2005-04-29;最终修改稿收到日期:2009-08-03.本课题得到国家"八六三"高技术研究发展计划探索导向类项目基金(2007AA010302,2009AA012404)资助.**张 威**,男,1968 年生,博士,教授,主要研究领域为软件工程、软件测试.E-mail:zw2051@126.com.**宫云战**,男,1961 年生,博士,教授,博士生导师,主要研究领域为软件工程、软件测试、集成电路测试等.**卢庆龄**,女,1969 年生,博士,副教授,主要研究方向为软件工程、人工智能.**万 琳**,女,1974 年生,博士,副教授,主要研究方向为软件工程、软件测试.

1 引　言

随着信息技术的发展,软件的规模不断扩大,如何保证和提高软件质量成为软件界最为关心的问题之一.软件测试作为保证软件质量的关键技术之一,能够有效地发现软件中的故障.根据 Boehm 的统计,在软件开发总成本中,用在测试上的开销要占 30%～50%[1].对于某些关系人的生命安全的关键软件,其测试费用甚至高达所有其他软件工程阶段费用总和的 3～5 倍.因此,提高软件测试的有效性和测试效率,降低软件开发成本已成为软件工程师迫切需要解决的任务之一.

软件测试方法可以分为两大类,即动态方法和静态方法[1].动态方法的主要缺点是只能依靠特定的测试用例来检测故障,因而不能检测所有故障,只能检测测试用例覆盖到的故障.此外,动态测试工具的执行开销也相当高,有时是不可接受的.静态方法不运行被测程序,而是分析程序源代码,并从中找出程序故障.但这并不意味着不利用计算机作为分析工具,它和人工测试有着本质的区别.静态分析算法不需要任何运行时的开销,但需要做大量的分析工作.

许多重要的编程语言都通过指针操作来支持动态内存管理,以提高语言的表达能力,但拥有指针的编程语言也会引起大量的动态内存故障.通过对 30 余万行 C++语言源代码中的动态内存故障进行分析可以看出,每千行源代码(KLOC)故障数为 0.921,其中内存释放异常 0.015,内存泄漏 0.355,空指针引用 0.551,这说明动态内存故障在程序中是普遍存在的,研究针对这些故障的测试方法具有非常重要的意义.检测动态内存故障是非常困难的,也难以准确识别出程序中的故障源[2].静态检测动态内存故障需要进行指针分析,一些研究人员已经对基于堆操作的指针分析进行了研究[3-5],其基本思路就是开发特定的算法来检测特定的内存问题(如内存释放异常、内存泄漏和空指针引用等),这些算法不需要执行程序,也不需要借助程序注释和程序文档.这些方法的不足之处主要在于控制流模型不够精确,导致漏报和误报过多的情况发生.

Fradet 等人[6]提出了一种类 Hoare 逻辑来检测 C 程序中无效指针引用故障.这种分析方法基于别名和连接关系分析,能够处理循环数据结构,但循环不变量需要由用户提供,这种工作量有时是无法忍受的.Ghiya 等人[7]提出的方法综合考虑了指针分析、指向分析和连接关系分析.指向堆栈的指针采用基于存储的指向分析模型,堆指针采用堆分析、连接关系分析和形状分析.这些方法的主要缺点是没有考虑控制流信息.Scholz 和 Zhu 等人[8-9]提出了一种采用符号运算的方法来进行指针分析.符号运算是一种很有效的数据流和控制流分析工具,它能判断出一个给定程序在运行时的特性而无须执行该程序.但这种方法涉及的代数系统过于抽象,并且不易区分内存分配、释放和指针变量赋值等操作.

上述研究工作所要解决的问题和本文所要解决的问题有某些相似性,但本文提出的指针映射代数系统将内存分配、释放和指针变量赋值等操作进行了细化,更全面地反映了指针与内存之间的映射关系.此外,提出了面向故障的指针映射集的概念,给出了面向不同故障的指针映射集的构造规则,以此为基础建立了动态内存故障模型.通过指针映射集和故障模型,可以检测内存释放异常、内存泄露和空指针引用等动态内存故障,提高了测试效率.在分析过程中,还综合应用了控制流图和路径条件,提高了测试结果的精度.

本文第 2 节介绍基本概念和指针状态及其变化;第 3 节介绍指针映射集,包括指针映射代数系统和面向故障的指针映射集;第 4 节建立故障模型;第 5 节给出测试算法;第 6 节通过实例分析测试过程,并给出测试结果;第 7 节总结全文.

2 指针状态分析

2.1 基本概念

动态内存故障有很多种类,本文论述与动态分配内存相关的几类故障的静态分析方法,包括内存释放异常(Bad Deallocation,F_{BD})、内存泄露(Memory Leak,F_{ML})和空指针引用(Null Pointer Dereference,F_{NPD}),这几类故障都与指针变量相关(以下将指针变量简称为指针).下面给出这几类故障的形式化定义.

定义 1.　与动态内存故障相关的谓词定义如下.

Malloc(p):为指针 p 动态分配堆地址单元;

Static(p):使指针 p 指向静态变量;

Null(p):使指针 p 指向特殊地址单元 Φ

Free(p):释放指针 p 所指向的存储单元;

Status(p):取指针 p 的当前状态;

Access(p)：访问指针 p；

Invalid(p)：指针 p 的生存期结束；

Point(p,h)：指针 p 指向堆地址空间 h；

Belong(x,y)：x 属于 y.

定义 2. 令 S 表示静态变量的集合，V 表示指针的集合，$p\in V$，$x\in S$，在程序中的 S_i 点，如果 Free(p)，且 p 当前指向 x 或 Φ，则称在 S_i 点存在 F_{BD} 故障. 记作

$$\text{Free}(p)\wedge(\text{Point}(p,x)\vee\text{Point}(p,\Phi))\Rightarrow F_{BD}.$$

定义 3. 令 H 表示所有堆地址的集合，H_t 表示已经分配的堆地址的集合，$H_t\subseteq H$，$p\in V$，执行程序中 S_i 点的语句之后，如果 $\exists h\in H_t$，且 $\neg\exists p(\text{Point}(p,h))$，则称在 S_i 点存在 F_{ML} 故障. 记作

$$\exists h(\text{Belong}(h,H_t)\wedge\neg\exists p(\text{Belong}(p,V)\wedge\text{Point}(p,h)))\Rightarrow F_{ML}.$$

定义 4. 令 $p\in V$，在程序中的 S_i 点，如果 Access(p)，且 p 当前指向 Φ，则称在 S_i 点存在 F_{NPD} 故障. 记作

$$\text{Access}(p)\wedge\text{Point}(p,\Phi)\Rightarrow F_{NPD}.$$

2.2 指针的状态及其变化

指针在其存在的过程中，由于各条与指针相关的语句的作用，使得它们的状态不断发生变化. 通常一个指针可以划分为 3 种基本状态：① 空状态(记为 S_{uncert})：指针未指向任何有效的地址单元. 指针声明和释放之后都变为 S_{uncert} 状态；② 堆栈状态(记为 S_{stack})：指针指向静态变量；③ 堆状态(记为 S_{heap})：指针指向堆地址. 这 3 种基本状态反映了指针在生存期内的动态特性，但还不能揭示所有的动态内存故障. 为此引入了反映指针产生和消亡的两种状态：④ 声明状态(记为 S_{decl})：标识指针的产生；⑤ 失效状态(记为 $S_{invalid}$)：指针生存期结束.

指针状态及其变化如图 1 所示. 图中实线箭头表示正常的状态变化过程，虚线箭头表示可能产生动态内存故障的状态变化过程. 下面着重讨论可能产生故障的变化过程.

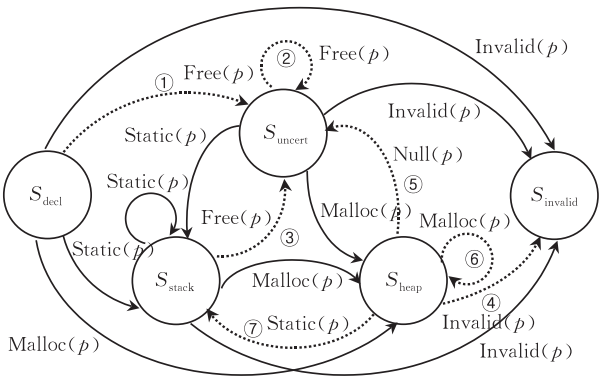

图 1 指针 P 的状态及其变化

变化 1. 指针 p 由 S_{decl} 状态变为 S_{uncert} 状态. 如果执行 Free(p)操作，就会导致内存释放异常故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{decl})\wedge\text{Free}(p)\Rightarrow F_{BD}.$$

变化 2. 指针 p 由 S_{uncert} 状态变为 S_{uncert} 状态. 如果执行 Free(p)操作，就会导致内存释放异常故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{uncert})\wedge\text{Free}(p)\wedge\cdots\wedge\text{Free}(p)\Rightarrow F_{BD}.$$

变化 3. 指针 p 由 S_{stack} 状态变为 S_{uncert} 状态. 如果执行 Free(p)操作，就会导致内存释放异常故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{stack})\wedge\text{Free}(p)\Rightarrow F_{BD}.$$

变化 4. 指针 p 由 S_{heap} 状态变为 $S_{invalid}$ 状态. 为指针 p 分配了堆地址空间，但没有执行释放操作指针 p 即失效，导致内存泄露故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{heap})\wedge\text{Invalid}(p)\Rightarrow F_{ML}.$$

变化 5. 指针 p 由 S_{heap} 状态变为 S_{uncert} 状态. 如果执行 Null(p)操作，并且此时没有其它指针 q 指向 p 所指的堆地址空间 h，就会导致内存泄漏故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{heap})\wedge\text{Null}(p)\wedge(\neg\exists q\ (\text{Point}(q,h)))\Rightarrow F_{ML}.$$

变化 6. 指针 p 由 S_{heap} 状态变为 S_{heap} 状态. 即再次为同一个指针 p 分配堆地址空间，如果此时没有其它指针 q 指向 p 原来所指的堆地址空间 h，就会导致内存泄露故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{heap})\wedge\text{Malloc}(p)\wedge(\neg\exists q\ (\text{Point}(q,h)))\Rightarrow F_{ML}.$$

变化 7. 指针 p 由 S_{heap} 状态变为 S_{stack} 状态. 即指针 p 已经指向了堆地址空间 h，又令其指向静态变量，如果此时没有其它指针 q 指向 h，就会导致内存泄露故障，形式化描述如下：

$$\forall p,(\text{Status}(p)=S_{heap})\wedge\text{Static}(p)\wedge(\neg\exists q\ (\text{Point}(q,h)))\Rightarrow F_{ML}.$$

上述变化可能产生内存释放异常和内存泄漏故障，除此之外，如果访问处于 S_{decl} 和 S_{uncert} 状态的指针，会产生空指针引用故障，形式化描述如下：

$$\forall p,((\text{Status}(p)=S_{decl})\vee(\text{Status}(p)=S_{uncert}))\wedge\text{Access}(p)\Rightarrow F_{NPD}.$$

3　指针映射集

3.1　指针映射代数系统

定义 5.　令 S 表示静态变量的集合,V 表示指针的集合,H 表示所有堆地址的集合,则广义指针映射集定义为 $(V\cup\Phi)\times(H\cup S\cup\Phi)=\{\langle x,y\rangle\mid x\in(V\cup\Phi)\wedge y\in(H\cup S\cup\Phi)\}$,记作 Ψ.

定义 6.　广义指针映射集 Ψ 及定义在该集合上的运算△、☆和⊙称为指针映射代数系统,记作 $\langle\Psi,\triangle,\star,\odot\rangle$. 其中运算定义如下:

△:动态分配运算,$\langle x,y\rangle\triangle\langle\Phi,z\rangle=\langle x,z\rangle$,其中 $x\in V,y\in(H\cup S\cup\Phi),z\in H$,且 $x\neq\Phi$;

☆:赋值运算,$\langle x_1,y_1\rangle\star\langle x_2,y_2\rangle=\langle x_1,y_2\rangle$,其中 $x_1\in V,x_2\in V,y_1\in(H\cup S\cup\Phi),y_2\in(H\cup S\cup\Phi)$,且 $x_1\neq\Phi$;

⊙:释放运算,$\langle x,y\rangle\odot=\langle x,\Phi\rangle$,其中 $x\in V,y\in H$,且 $x\neq\Phi$.

从定义 5、定义 6 可以看出:

① 指针只能指向静态变量、堆地址单元或特殊地址单元 Φ;

② △运算指调用内存分配函数的操作,C++程序设计语言提供的内存分配函数有 *new*、*strdup*、*malloc*、*calloc*、*realloc* 等. 该运算实际上是将分配操作和赋值操作组合在一起,因为单纯的分配操作没有任何意义. 例如,$p=malloc(sizeof(int))$ 可描述为 $\langle p,y\rangle\triangle\langle\Phi,z\rangle$,$z$ 即为动态分配的堆地址单元,运算结果为 $\langle p,z\rangle$;

③ ☆运算即指针的赋值操作,改变指针所指向的内容;

④ ⊙运算是单目运算符,运算的结果是回收堆地址单元,并使指针指向特殊地址单元 Φ.

性质 1.　指针映射代数系统 $\langle\Psi,\triangle,\star,\odot\rangle$ 是封闭的.

性质 2.　指针映射代数系统 $\langle\Psi,\triangle,\star,\odot\rangle$ 是不可交换的.

3.2　面向故障的指针映射集

定义 7.　动态反映程序中指针映射关系的集合称为面向故障的指针映射集,记作 T,$T\subseteq\Psi$.

面向故障的指针映射集 T 是广义指针映射集 Ψ 的子集,是分析程序语句的过程中动态生成的,它反映了程序中的指针与内存地址之间的映射关系. 本文通过面向故障的指针映射集 T 来分析动态内存故障.

定义 8.　对面向故障的指针映射集 T 进行操作的函数定义如下:

① Add($\langle p,h^{(i)}\rangle$). 添加元素 $\langle p,h^{(i)}\rangle$ 到集合 T. 其中 $p\in V,h^{(i)}\in H$,由于检测动态内存故障只需知道分配了一个存储单元即可,而不必考虑该单元的具体地址,因此 i 的作用只在于区分不同的堆地址单元,可设置为一个递增变量.

② Change($\langle p_1,h^{(i)}\rangle,\langle p_1,h^{(j)}\rangle$). 将 $\langle p_1,h^{(i)}\rangle$ 变为 $\langle p_1,h^{(j)}\rangle$. 其中 $p_1\in V,h^{(i)}\in H,h^{(j)}\in H$.

③ Delete($\langle p,h^{(i)}\rangle$). 从集合 T 中删除元素 $\langle p,h^{(i)}\rangle$. 其中 $p\in V,h^{(i)}\in H$.

④ Get(p). 从集合 T 中取 p 所指向的堆地址. 其中 $p\in V$. 如不存在,则返回 Φ.

⑤ JudgeNull(p). 判断 p 是否指向 Φ,如果指向 Φ,返回 True,否则返回 False.

为了更全面地找出动态内存故障,需要分别建立面向故障的指针映射集,内存释放异常和内存泄漏故障共用一个指针映射集 T_{BM},空指针引用故障单独使用一个指针映射集 T_{NPD},两者的主要区别是对动态分配函数的处理,前者在没有判断的情况下,假定分配成功;而后者在没有判断的情况下,假定分配不成功. 下一节将详细介绍 T_{BM} 和 T_{NPD} 的构造规则并给出故障模型.

4　故障模型

4.1　内存释放异常和内存泄漏故障模型

指针映射集 T_{BM} 按以下规则构造:

M1. 初始值为 Φ;

M2. 令 $p\in V,h^{(i)}\in H,h^{(j)}\in H$,如果 Malloc($p$),使 p 指向 $h^{(i)}$,且 $\neg\exists\langle p,h^{(j)}\rangle$(Belong($\langle p,h^{(j)}\rangle,T_{BM}$)),则 Add($\langle p,h^{(i)}\rangle$);

M3. 令 $p\in V,h^{(i)}\in H,h^{(j)}\in H$,如果 Malloc($p$),使 p 指向 $h^{(i)}$,且 $\exists\langle p,h^{(j)}\rangle$(Belong($\langle p,h^{(j)}\rangle,T_{BM}$)),则 Change($\langle p,h^{(j)}\rangle,\langle\Phi,h^{(j)}\rangle$),Add($\langle p,h^{(i)}\rangle$);

M4. 令 $p\in V,h^{(i)}\in H$,如果 Free(p),使 p 不再指向 $h^{(i)}$,且 $\exists\langle p,h^{(i)}\rangle$(Belong($\langle p,h^{(i)}\rangle,T_{BM}$)),则 Delete($\langle p,h^{(i)}\rangle$);

M5. 令 $p\in V,x\in S,h^{(i)}\in H$,如果 $p=\&x$,使 p 指向 x,则

如果 $\exists\langle p,h^{(i)}\rangle$(Belong($\langle p,h^{(i)}\rangle,T_{BM}$)),且 $\neg\exists\langle q,h^{(i)}\rangle$(Belong($\langle q,h^{(i)}\rangle,T_{BM}$)),则Change($\langle p,h^{(i)}\rangle,\langle\Phi,h^{(i)}\rangle$);

如果 $\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，且 $\exists\langle q,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，则 Delete($\langle p,h^{(i)}\rangle$).

M6. 令 $p\in V,q\in V,h^{(i)}\in H,h^{(j)}\in H$，如果 $p=q$，使 p 从指向 $h^{(i)}$，变为指向 $h^{(j)}$，则 $h^{(j)}=Get(q)$；

如果 $\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，且 $\neg\exists\langle r,h^{(i)}\rangle(\text{Belong}(\langle r,h^{(i)}\rangle,T_{BM}))$，则Change($\langle p,h^{(i)}\rangle,\langle\Phi,h^{(i)}\rangle$)；

如果 $\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，且 $\exists\langle r,h^{(i)}\rangle(\text{Belong}(\langle r,h^{(i)}\rangle,T_{BM}))$，则 Delete($\langle p,h^{(i)}\rangle$)；

如果 $\exists\langle q,h^{(j)}\rangle(\text{Belong}(\langle q,h^{(j)}\rangle,T_{BM}))$，且 $h^{(j)}\neq\Phi$，则 Add($\langle p,h^{(j)}\rangle$).

M7. 如果 JudgeNull(p)为 True，且 $\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，则 Delete($\langle p,h^{(i)}\rangle$)；

M8. 令 $p\in V,h^{(i)}\in H$，如果 Invalid(p)，使 p 不再指向 $h^{(i)}$，且 $\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}))$，则 Change($\langle p,h^{(i)}\rangle,\langle\Phi,h^{(i)}\rangle$)；

根据以上规则，可得内存释放异常故障模型：

在程序中的 S_i 点，当前指针映射集为 $T_{BM}(S_i)$，$p\in V,h^{(i)}\in H$，如果 Free(p)，且 $\neg\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}(S_i)))$，则在 S_i 点存在 F_{BD}. 记作

$$\text{Free}(p)\wedge\neg\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}(S_i)))\Rightarrow F_{BD}\quad(1)$$

根据 T_{BM} 构造规则，$\neg\exists\langle p,h^{(i)}\rangle(\text{Belong}(\langle p,h^{(i)}\rangle,T_{BM}(S_i)))$，说明指针 p 没有指向堆地址空间，即 Point(p,x)或 Point(p,Φ))，其中 $x\in S$，根据定义 2 可知存在 F_{BD}故障.

内存泄漏故障模型：

在程序中 S_i 点，当前指针映射集为 $T_{BM}(S_i)$，如果 $\exists\langle\Phi,h^{(i)}\rangle(\text{Belong}(\langle\Phi,h^{(i)}\rangle,T_{BM}(S_i)))$，$h^{(i)}\in H$，则在 S_i 点存在 F_{ML}. 记作

$$\exists\langle\Phi,h^{(i)}\rangle(\text{Belong}(\langle\Phi,h^{(i)}\rangle,T_{BM}(S_i)))\Rightarrow F_{ML}\quad(2)$$

根据 T_{BM} 构造规则，$\exists\langle\Phi,h^{(i)}\rangle(\text{Belong}(\langle\Phi,h^{(i)}\rangle,T_{BM}(S_i)))$，则一定有动态分配操作，且没有释放操作，左变元为 Φ，说明没有指针指向堆地址单元 $h^{(i)}$，根据定义 3 可知存在 F_{ML}故障.

4.2 空指针引用故障模型

指针映射集 T_{NPD}按以下规则构造：

N1. 初始值为 Φ；

N2. 令 $p\in V,x\in S$，如果 JudgeNull(p)为 False，且 $\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，则 Add($\langle p,x\rangle$)；

N3. 令 $p\in V,x\in S$，如果 JudgeNull(p)为 True，且 $\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，则 Delete($\langle p,x\rangle$)；

N4. 令 $p\in V,x\in S$，如果 Free(p)，且 $\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，则 Delete($\langle p,x\rangle$)；

N5. 令 $p\in V,x\in S$，如果 $p=\&x$，且$\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，则 Add($\langle p,x\rangle$)；

N6. 令 $p\in V,q\in V,x\in S$，如果 $p=q$，$\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，且 $\exists\langle q,x\rangle(\text{Belong}(\langle q,x\rangle,T_{NPD}))$，则 Add($\langle p,x\rangle$)；

N7. 令 $p\in V,x\in S$，如果 Invalid(p)，且 $\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}))$，则 Delete($\langle p,x\rangle$)；

根据以上规则，可得空指针引用故障模型：

在程序中 S_i 点，当前指针映射集为 $T_{NPD}(S_i)$，如果 Access(p)，且 $\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}(S_i)))$，则在 S_i 点存在 F_{NPD}故障. 记作

$$\text{Access}(p)\wedge\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}(S_i)))\Rightarrow F_{NPD}\quad(3)$$

根据 T_{NPD}构造规则，T_{NPD}只在确保非空的情况下才添加元素，因此 $\neg\exists\langle p,x\rangle(\text{Belong}(\langle p,x\rangle,T_{NPD}(S_i)))$，说明 Point($p,\Phi$)，根据定义 4 可知存在 F_{NPD}故障.

本节讨论了面向故障的指针映射集的构造规则，据此建立了内存释放异常、内存泄漏和空指针引用故障模型，模型的建立是进一步检测工作的基础.

5 测试算法

测试算法的操作步骤如下：

1. 预编译. 由于源程序中存在宏定义、文件包含和条件编译等预处理命令，因此在进行词法分析前必须进行预处理，将宏进行展开，这样有利于变量的查找；

2. 词法分析. 将预编译阶段产生的中间代码进行分解，形成各种符号表，为语法分析作准备. 符号表的结构主要有标识符表、类型表、关键字表、常数表、运算符表和分界符表；

3. 语法分析. 这一步主要是将输入字符串识别为单词符号流，并按照标准的 C++语法规则，对源程序作进一步分析，区分出变量定义、赋值语句、函数等等. 语法分析的结果是生成语法树，并提供对外的接口. 此外，通过语法树可以生成程序的控制流图和变量的定义使用链，为下一步的故障查找作准备；

4. 取控制流图中的一条路径；

5. 按顺序取路径中的一个结点，如果是与指针相关的操作，则按 M 规则和 N 规则修改面向故障的指针映射集 T_{BM}和 T_{NPD}；

6. 按式(1)、(2)和(3)分别检测是否存在 F_{BD}、F_{ML} 和 F_{NPD} 故障;

7. 返回步 5 直到所有结点检测完毕;

8. 返回步 4 直到所有路径检测完毕.

6　实例与实验结果分析

6.1　实例分析

下面通过一个实际程序来分析一下测试过程,程序代码如下:

```
s1    p=malloc(sizeof(int));
s2    q=malloc(sizeof(int));
s3    if (q==NULL)
s4      return;
s5    if (x==1)
s6    {   p=&x;
s7        free(p);
s8        free(q);
s9        return; }
s10   y=*p;
s11   free(p);
s12   free(q);
```

程序控制流图如图 2 所示.

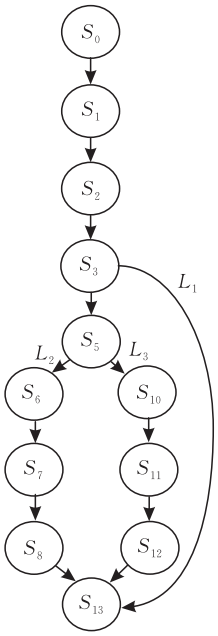

图 2　程序控制流图

该程序有 3 条路径 L_1、L_2 和 L_3,在各条路径中,指针映射集 T_{BM} 和 T_{NPD} 的变化情况如表 1 所示.

表 1　T_{BM} 和 T_{NPD} 变化情况及检测结果

路径	语句	T_{BM}	规则	T_{NPD}	规则	结果
L_1	s_1	$\{\langle p,h^{(0)}\rangle\}$	M1	$\{\}$		
	s_2	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M1	$\{\}$		
	s_3	$\{\langle p,h^{(0)}\rangle\}$	M7	$\{\}$	N1	
	s_4	$\{\langle \Phi,h^{(0)}\rangle\}$	M8	$\{\}$	N7	T_{BM} 中出现左变元为 Φ 的元素,因此产生 F_{ML} 故障
L_2	s_1	$\{\langle p,h^{(0)}\rangle\}$	M1	$\{\}$		
	s_2	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M1	$\{\}$		
	s_3	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M7	$\{\langle q,x\rangle\}$	N2	
	s_5	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$		$\{\langle q,x\rangle\}$		
	s_6	$\{\langle \Phi,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M5	$\{\langle p,x\rangle,\langle q,x\rangle\}$	N5	T_{BM} 中出现左变元为 Φ 的元素,因此产生 F_{ML} 故障
	s_7	$\{\langle \Phi,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M4	$\{\langle q,x\rangle\}$	N4	释放 p,且 T_{BM} 中不存在左变元为 p 的元素,因此产生 F_{BD} 故障
	s_8	$\{\langle \Phi,h^{(0)}\rangle\}$	M4	$\{\}$	N4	
	s_9	$\{\langle \Phi,h^{(0)}\rangle\}$	M8	$\{\}$	N7	
L_3	s_1	$\{\langle p,h^{(0)}\rangle\}$	M1	$\{\}$		
	s_2	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M1	$\{\}$		
	s_3	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$	M7	$\{\langle q,x\rangle\}$	N2	
	s_5	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$		$\{\langle q,x\rangle\}$		
	s_{10}	$\{\langle p,h^{(0)}\rangle,\langle q,h^{(1)}\rangle\}$		$\{\langle q,x\rangle\}$		访问 p,且 T_{NPD} 中不存在左变元为 p 的元素,因此产生 F_{NPD} 故障
	s_{11}	$\{\langle q,h^{(1)}\rangle\}$	M4	$\{\langle q,x\rangle\}$	N4	
	s_{12}	$\{\}$	M4	$\{\}$	N4	
	s_{13}	$\{\}$	M8	$\{\}$	N7	

从指针映射集 T_{BM} 和 T_{NPD} 的变化可以看出,在路径 L_1 的结点 s_4 处,T_{BM} 中存在一个元素的左变元为 Φ,因此产生一个 F_{ML} 故障. 在路径 L_2 的结点 s_6 处,T_{BM} 中存在一个元素的左变元为 Φ,因此产生一个 F_{ML} 故障;结点 s_7 处释放指针 p,但 T_{BM} 中不存在左变元为 p 的元素,因此产生一个 F_{BD} 故障. 在路径 L_3 的结点 s_{10} 处,访问指针 p,但 T_{NPD} 中不存在左变元为 p 的元素,因此产生 F_{NPD} 故障.

6.2　实验结果及分析

采用本文所述的方法,对国内外不同领域的 7 个

项目进行了实验测试，测试结果如表 2 所示，其中前 4 个项目为国外项目，后 3 个项目为国内项目，由于有些项目头文件不全，因此统计中剔除了头文件不全的源文件.

表 2 实验测试结果

项目编号	源代码行数	F_{BD}	F_{ML}	F_{NPD}	IP 总数	故障总数
1	50455	0	2	13	75	15
2	55863	1	14	41	118	56
3	42799	0	10	26	115	36
4	85270	3	76	56	420	135
5	11733	0	4	16	36	20
6	11859	0	4	6	22	10
7	68706	1	6	22	49	29
合计	326685	5	116	180	835	301
故障率		0.015	0.355	0.551		0.921

表 2 中 IP(Inspection Points)表示自动发现的故障点，故障表示经人工确认后的故障点，用故障率和准确率来表示测试的效果，其定义如下：

$$\text{故障率} = \frac{\text{故障总数}}{\text{源代码行数}} \times 1000(\text{个/KLOC}),$$

$$\text{准确率} = \frac{\text{故障总数}}{\text{IP 总数}} \times 100(\%),$$

其中故障率即千行源代码(KLOC)的故障数，准确率即故障在 IP 中所占比重，它也可以反映出误报的情况，误报率=(1−准确率).

测试代码的总行数有 32 万余行，从中找出 301 个动态内存故障，平均每千行源代码 0.921 个故障. 这些数据具有一定的统计意义，说明这些故障在项目中是普遍存在的，开发针对这些故障的专用测试工具具有很高的实用价值.

对于 4 个国外项目，我们与 Reasoning 公司的测试结果进行了对比，如表 3 所示.

表 3 测试结果比较

项目编号	本文所述方法			Reasoning 工具		
	IP 总数	故障总数	准确率/%	IP 总数	故障总数	准确率/%
1	75	15	20	107	9	8.4
2	118	56	47.5	110	53	48.2
3	115	36	31.3	164	29	17.9
4	420	135	32.1	359	66	18.4
合计	728	242	33.2	740	157	21.2

表 3 中 Reasoning 工具的测试数据引自该公司测试结果数据库. 从中可以看出，采用本文所述的方法，从 20 余万行源代码中找出 728 个 IP，经人工确认，其中有 242 个故障，准确率为 33.2%，而 Reasoning 工具找出 740 个 IP，其中有 157 个故障，准确率为 21.2%. 这说明本文所述方法在测试动态内存故障方面还是比较有效的.

7 结束语

本文提出了一种新的动态内存故障测试方法，该方法基于指针映射关系分析. 通过构造面向故障的指针映射集，建立了动态内存故障模型，并按照故障模型，给出了测试算法，该算法可以自动检测内存释放异常、内存泄露和空指针引用等动态内存故障，提高了测试效率. 在分析过程中，还综合应用了控制流图和路径条件，提高了测试结果的精度. 但由于程序语法的复杂性，采用静态分析方法难以覆盖所有的语法现象，因而还存在误报和漏报的情况. 下一步的研究目标是根据程序中的各种语法现象，扩展指针映射集的构造规则，以减少误报和漏报，使之具有更强的适应性.

参 考 文 献

[1] Zheng Ren-Jie. Computer Software Testing Technology. Beijing: Tsinghua University Press, 1992(in Chinese)
(郑人杰. 计算机软件测试技术. 北京：清华大学出版社，1992)

[2] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors//Proceedings of the Winter USENIX Technical Conference. Monterey, California, USA, 1999: 125-136

[3] Gotsman A, Berdine J, Cook B. Interprocedural shape analysis with separated heap abstractions//Proceedings of the 13th International Static Analysis Symposium (SAS'06). Seoul, Korea, 2006: 241-260

[4] Landi W, Ryder B G. Safe approximate algorithm for interprocedural pointer aliasing. ACM SIGPLAN Notices, 1992, 27(7): 235-248

[5] Wilson R P, Lam M S. Efficient context-sensitive pointer analysis for C program//Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation(PLDI). La Jolla, California, USA, 1995: 18-21

[6] Fradet P, Caugne R, Metayer D L. Static detection of pointer errors: An axiomatisation and a checking algorithm//Proceedings of the 6th European Symposium on Programming Languages and Systems-ESOP'96. Linkoping, Sweden, 1996: 22-24

[7] Ghiya R, Hendren L. Putting pointer analysis to work//Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, CA, 1998: 121-133

[8] Scholz B, Blieberger J, Fahringer T. Symbolic pointer analysis for detecting memory leaks//Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Seman-

tics-Based Program Manipulation. Boston, Massachusetts, USA, 2000: 104-113

[9] Zhu J, Calman S. Symbolic pointer analysis revisited//Proceedings of the Programming Language Design and Implementation (PLDI). Washington, DC, USA, 2004: 145-157

[10] Emami M, Ghiya R, Hendren L J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices, 1994, 29(6): 242-256

[11] Sridharan M, Bodik R. Refinement-based context sensitive points-to analysis for java//Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 2006: 387-400

[12] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors. Software-Practice and Experience, 2000, 30(7): 775-802

[13] Nie Chang-Hai, Xu Bao-Wen. A minimal test suite generation method. Chinese Journal of Computers, 2003, 26(12): 1690-1695(in Chinese)
(聂长海,徐宝文. 一种最小测试用例集生成方法. 计算机学报, 2003, 26(12): 1690-1695)

ZHANG Wei, born in 1968, Ph. D., professor. His research interests include software engineering and software testing.

GONG Yun-Zhan, born in 1961, Ph. D., professor. His research interests include software engineering, software testing and integrated circuit testing.

LU Qing-Ling, born in 1969, Ph. D., associate professor. Her current research interests include software engineering and artificial intelligence.

WAN Lin, born in 1974, Ph. D., associate professor. Her current research interests include software engineering and software testing.

Background

Software testing as one of the key technique of guaranteeing the quality of software, it can effectively detect faults. According to the statistics of Boehm, cost used in testing accounts for 30% to 50% of the total software development cost. For some key software related with life's safety, the testing cost even is three to five times of the total cost of all the other software engineering phases. Therefore, improving the effectiveness of software testing and testing efficiency, reducing the cost of software development has become one of the urgent tasks needs to be solved.

Statistical data shows that the dynamic memory faults is common in the program with pointers, it is difficult to locate accurately the source of faults by adopting dynamic testing methods. Static analysis methods nowadays sometimes miss some faults or give too many false alarms. So, studying faults testing method is a very important theoretical and practical significance.

The authors of this paper have been involved in GIS Company software faults testing work, namely, confirmed the check points (IP) found by static analysis tool of Reasoning Company. It involved software of many international well-known companies and its code amounted to more than 200 million lines. The authors found that the tool omitted some faults and had a higher rate of false alarm, so, analyzed deeply faults type and characteristics and put forward a new testing method. Comparing with testing results checked by static analysis tool of Reasoning Company, it could find more faults and has higher accuracy.

At present, the authors have developed Defect Testing System(DTS) based on static analysis, and it has been put into use. The work is supported by the National High Technology Research and Development Program(863 Program) of China under grant No. 2007AA010302 and No. 2009AA012404.