Java 语言中数组越界故障的静态测试研究

赵鹏宇[1],李建茹[2],宫云战[3]
ZHAO Peng-yu[1],LI Jian-ru[2],GONG Yun-zhan[3]

1.总装工程兵科研一所,江苏 无锡 214035
2.唐山学院,河北 唐山 063020
3.北京邮电大学 网络与交换技术国家重点实验室,北京 100876
1.The First Engineers Scientific Research Institute of the General Armaments Department,Wuxi,Jiangsu 214035,China
2.Tangshan College,Tangshan,Hebei 063020,China
3.Network and Exchange Technology State Key Laboratory,Beijing University of Posts and Telecommunications,Beijing 100876,China
E-mail:zhaopengyu006@163.com

ZHAO Peng-yu,LI Jian-ru,GONG Yun-zhan.Research on static test about array index out of range in Java language. Computer Engineering and Applications,2008,44(27):87-90.

Abstract: The software test technology oriented to the special fault is a hotspot in the recent years.Array index out of range is a common fault in Java programme design,which can easily cause system collapse.The paper analyzes the classical array index out of range fault in Java programme,establishes a defect model combining the advantages of static test and gives a static method for searching for such faults.The method has been implemented in the defect-oriented test system.
Key words: static test;array index out of range;syntax tree;controlling stream graph;integer range

摘 要: 面向具体故障的软件测试技术是当今一个研究热点。数组越界是 Java 程序设计中的常见故障,该类故障极易导致计算结果错误或系统崩溃。针对 Java 语言中常见数组越界故障进行了分析,并从面向具体故障的测试思想出发,建立了 Java 语言中数组越界的故障模型,结合静态测试的特点,给出了一种静态查找此类故障的方法。此方法已实现,并已应用于面向故障的软件测试系统中。
关键词: 静态测试;数组越界;语法树;控制流图;整型区间
DOI: 10.3778/j.issn.1002-8331.2008.27.028 **文章编号:** 1002-8331(2008)27-0087-04 **文献标识码:** A **中图分类号:** TP302.8

软件测试可以分为静态测试方法和动态测试方法[1]。软件测试的目的在于发现错误,进一步可以针对某一种或者某几种错误进行专门的测试,这种测试方法叫做面向具体故障类型的测试[2]。缓冲区是程序运行时在内存中申请的一段连续的临时数据存储区域,它保存了给定类型的数据。缓冲区溢出是一种常见且危害很大的软件错误,如果程序的缓冲区被写入超出其长度的内容,就会造成缓冲区的溢出,破坏程序的堆栈,使程序转而执行其它的指令,从而导致软件系统崩溃。在 Java 语言中,给程序员提供了申请连续内存缓冲区的数据结构和方法,例如数组。而由于程序员的理解偏差和粗心,经常会造成数组越界的错误,导致程序输出结果错误或死机。

1 基本概念

1.1 静态分析

静态分析是指不运行实际被测的源程序,而是通过采用其它手段对程序结构进行分析的一种技术[2]。在测试阶段使用静态分析技术能有效的发现程序中存在的错误,特别是逻辑结构方面存在的一些错误,其中有些错误是动态测试发现不了的。对程序进行静态测试不仅省时、省力,避免了大量测试数据的选择,而且能有效的对故障源进行定位。依据静态分析技术,开发静态分析工具,可以对程序进行有效的分析。实现静态分析技术的工具在功能和应用范围上都是多种多样的,静态分析测试工具根据被测程序结构特性进行分析,并不考虑测试中程序的可执行性。

1.2 数组越界故障

数组越界是缓冲区溢出故障类型中的一种最常见的故障,它是由于对数组下标的操作超过了下标范围而引起的,数组越界在使用数组类型进行程序设计的软件中普遍存在。数组越界通常分为上溢和下溢。譬如声明一个数组,其大小为 len,则其下标范围为 $R=[0,len-1]$, 如果以 j 为数组下标引用数组元素时,$\forall j\in R$,则不会发生故障,如果 $j<0$,则发生故障,称之为下溢;如果 $j>len-1$,则称之为上溢。因此通常对数组边界的检查

基金项目: 国家高技术研究发展计划(863)(the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z184)。
作者简介: 赵鹏宇(1981-),男,硕士研究生,主要研究方向为软件工程、软件测试等;李建茹(1978-),女,助教,研究方向为软件工程;宫云战(1961-),男,博士,教授,博士生导师,主要研究方向为软件工程、软件测试、集成电路测试等。
收稿日期: 2007-11-15 **修回日期:** 2008-02-21

要包括两部分,即检查数组上界和下界,只有对下标的操作在其区间内,才能保证对数组的使用无误,否则就会发生数组越界故障。

例如:

```
11:public static void GetDirFiles(File dir,int n) {
12:System.err.println("Info:getting file number"+n
13:+"from directory"+dir);
14:File results[ ]=dir.listFiles();
15:if (results! =null && results.length >= n && n>=0){
16:System.out.println(n+"file is"+results[n]);
17:} else {
18:System.err.println("Error:not enough files");
19:}
20:}
```

在上述程序中,在第 14 行声明了一个名为 results 的数组,并用目录 dir 下的所有文件对象给数组中的每一个元素赋值。因为在目录 dir 下共有 n 个文件,所以程序在第 14 行执行结束后,数组 results 的长度为 n,下标的范围 $R=[0,n-1]$。在程序的 16 行,由于用 n 为下标来引用数组 results 中的元素,由于 $n>n-1$,发生了数组越界,即发生了上溢。

又如:

```
21:static boolean ReverveArray(int a[ ]) {
22:int result=new int[a.length];
23:int len=a.length;
24:for(int index=len-1;index>=0;index--){
25:        result[len-1-index]=a[index]}
26:result[index]=0;
27:}
```

在上述程序段的第 22 行声明了一个与数组 a 同长度的数组 result,长度是 a.length,因此数组 result 下标的范围为 [0,a.length-1]。而在程序执行到第 26 行时,index 值为-1,因此在此处数组 result 在用下标 index 引用其数组元素时发生了下溢。

1.3 语法树

语法树是程序代码的一种树形表示,是程序经过翻译后的一种中间表示形式。语法树描绘了如何从文法的开始符号开始推导出它的语言中的一个语句。形象地说,语法树是具有如下特性的树:(1)树根标记为开始符号;(2)每个叶结点由记号标记;(3)每个内结点由一个非终结符标记。

如果 A 是某个内结点的非终结符号标记,$X_1,X_2,\cdots,X_n$ 是该结点从左到右排列的所有子结点的标记,则 $A\to X_1X_2\cdots X_n$ 是一个产生式。这里,$X_1,X_2,\cdots,X_n$ 是一个终结符或非终结符。

依据 Java 语言的语法规则,语法分析器接受词法分析产生的记号串,并依照相应的规则将记号组织成具有确切含义的语句,并构造相应的语法树。语法树反映了程序语法的推导过程,一棵语法树从左到右的叶结点是这棵语法树生成的结果。

譬如赋值语句 position=initial+rate*60 可以用语法树表示如图 1 所示。

语法树是一个被简化和抽象了的程序结构,本文给出的语法树不仅仅反映了程序的语法构造过程,而且也给出了语法结点的一些属性信息。语法树包括表示非保留字终结符的叶子结点和表示语法结构的中间结点组成,它包含编译器前端从源代码所获得的全部信息,并且能够完全体现源程序的语法结构。

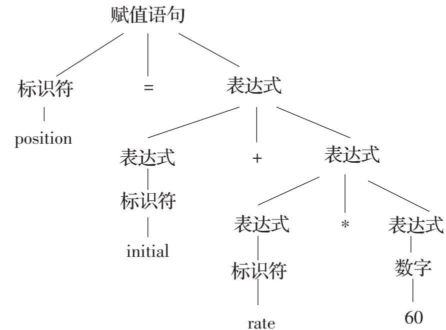

图 1 赋值语句的语法树

1.4 控制流图

通常一个程序的控制流图可表示为一个四元组 CFG=<N, E, $Entry$, $Exit$>。其中 N 是结点集,程序中的每个语句都对应图中的一个结点;边集 $E=\{<s_1,s_2>|s_1,s_2\in N$ 且语句 s_1 执行后,可能立即执行 $s_2\}$,反映程序中语句间的控制关系;如果 $<s_1,s_2>E$,则称 s_1 是 s_2 的直接前趋,s_2 是 s_1 的直接后继;语句 s 的所有直接前驱组成的集合称为 s 的直接前驱集,记为 $Pred(s)$,s 的所有直接后继组成的集合称为 s 的直接后继集,记为 $Succ(s)$;$Entry$ 称为程序的唯一入口结点,$Exit$ 为程序唯一的退出结点。简单的说:控制流图即是具有单一的、固定的入口结点和出口结点的有向图。例如下述程序段的程序控制流图如图 2 所示。

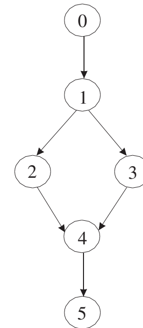

图 2 程序控制流图

```
Void function(int x,int y)
{
if(x>y)
   printf("%d",x);
else
   printf("%d",y);
}
```

2 数组变量抽象区间表示及区间分析

2.1 数组变量抽象区间表示

为了描述数组变量的静态特征以及在整个程序运行的过程中的动态特征,建立了一个四元组用来描绘数组类型的变量:

ArrayVar=<$varname$, $varvalue$, $refindex$, $deflen$>

其中,$varname$ 表示变量名称,可在符号表中查到;$varvalue$ 表示变量所对应的内存空间;$refindex$ 表示数组变量 $varname$ 在被引用时的数组下标;deflen 表示数组在定义时的长度。

需要指出的是在实际工作中构造数组变量的四元组时,实际上反映在程序算法上是构建变量符号表的过程。在构建变量符号表时,在记录变量的同时,不仅仅要包含上述四元组中的

内容，还包括其它信息，如变量声明的行号、所在的函数体、变量的生存期范围等等，因为这些信息与本文讨论的主题无关，因此这里并不作深入论述。

在程序 P 中任意一条属于语句集合 ST 的语句 s_i 处，如果存在数组变量 *varname*，则对数组变量 *varname* 的操作 *access*（*varname*），数组下标的大小不能超过定义时的长度范围，否则就会发生数组越界错误 F_{oob}。可定义为：

$(\forall s_i \in ST)(\exists varname \in V_{arr} \wedge access(varname) \wedge (refindex<0 \vee refindex>deflen-1)F_{oob})$

数组越界故障是在利用数组名和其下标引用数组中的元素时，下标值超出了数组的长度范围而产生的。当用数组名和其下标引用数组元素时，其下标形式可以归纳为以下 3 种形式：(1)数组下标是一常数，例如 *varname*[7]；(2)数组下标是一个变量，例如 *varname*[*j*]；(3)数组下标是一个表达式，例如 *varname*[*i**k+j*2]。

数组的下标无论是上述哪 3 种形式，下标都可以看作是整型表达式（常数是最简单的表达式）。但由于数组下标表达式中某些变量受用户输入或程序多分支的影响，其静态计算结果往往不是一个明确的值而是一个整型区间，即每一个数组变量映射一个区间 r，区间表示数组下标的整数范围，在对数组变量进行访问时，比较区间 r 和区间 $[0,len-1]$，len 表示数组变量定义的长度，如果 $r \subseteq [0,len-1]$，则对数组的操作在允许的范围之内，是正确的，否则就会发生数组越界故障。

2.2　区间运算

区间作为整型值的一种代数表现形式，满足一定的性质，这里将一些基本的代数运算如+、-、*、/等操作扩展到整型区间上，通过区间运算，实现对变量区间的更新和传播。

假设 $\min(x_1,x_2,\cdots)$ 和 $\max(x_1,x_2,\cdots)$ 这两个函数意义分别表示为求参数的最小值和最大值。设 $x\in[a,b]$，$y\in[c,d]$，$a\leqslant b$，$c\leqslant d$，对 $z=a+b$，假设 $z\in[e,f]$，区间 $[e,f]$ 可按以下规则确定：$[a,b]+[c,d]=[a+c,b+d]$；$[a,b]-[c,d]=[a-d,b-c]$；$[a,b]\times[c,d]=[\min(ac,ad,bc,bd),\max(ac,ad,bc,bd)]$；$ab>0$ 时 $1/[a,b]=[1/b,1/a]$，$ab<0$ 时，$1/[a,b]=(-\infty,1/a]\cup[1/b,\infty]$，$a=0$ 时 $1/[a,b]=[1/b,\infty)$，$b=0$ 时 $1/[a,b]=(-\infty,1/a]$；$[a,b]/[c,d]=$

$$\begin{cases}[\min(a/c,a/d,b/c,b/d),\max(a/c,a/d,b/c,b/d)], & c\leqslant d<0\\ [\min(a/c,a/d,b/c,b/d),\max(a/c,a/d,b/c,b/d)], & 0<c\leqslant d\\ [-\infty,+\infty], & c\leqslant 0\leqslant d\end{cases}$$

以上是最基本的区间运算规则，在这个基础上可以根据具体被测程序对上述规则进行扩充，用来计算数组下标表达式在程序的每条执行路径上具体区间范围 r。

2.3　区间传播

通过以上论述，可以把数组下标表达式映射到一个整型区间。所谓区间传播就是这个整型区间沿着程序路径得到不断更新和传播，使数组下标表达式中每一个变量映射一个最能反映变量此刻状态的值的区间。本文提到的区间传播采用基于程序控制流图的广度优先遍历分析技术，在分析过程中，将整型变量映射为一个整型区间[min，max]，然后通过对程序控制流图的分析，不断进行区间更新。这里，每一个整型变量对应一个区间，变量的区间可以通过多种操作进行改变，变量在定义时如果没有进行初始化，则变量区间被定义为 $[-\infty,+\infty]$。随着变量的变化，变量区间也不断变化，一般是通过赋值语句改变变量区间，对变量的区间进行重新定义，同时也可以通过其它的方法改变变量的区间，比如作为参数传递到函数体，在函数体里对参数进行操作，再传递到函数外面。通常，一个变量的定义伴随着一个区间的产生，而变量的结束暗示着区间的消亡。

在以上各个概念的基础上，本文提出如下结论。对于程序 P 任一数组变量 Var，其长度为 len，可为其构造一个区间集合 Ran，$Ran(Var)$ 是数组变量 Var 在程序 P 中的所有的程序执行路径的区间集合。令 $Ran'(Var)=Ran(Var)\cup\phi\cup\{[0,\mathrm{len}-1]\}$，则由 $Ran'(Var)$ 和关系 $\subseteq$ 组成的序偶 $<Ran'(Var),\subseteq>$ 构成格。由于篇幅有限，证明从略。

在格关系 $L=<Ran'(Var),\subseteq>$ 中，若 $Ran'(Var)$ 中的元素 $[0,len-1]$ 是集合 $Ran'(Var)$ 中所有元素的上界，则该程序不会发生数组越界故障，否则会有数组越界故障发生。

3　以语法树和控制流图为基础的数组越界故障分析

3.1　故障模型设计

为了更方便地说明数组越界的故障模型，本文给出以下定义：

定义 1　r_{in}、r_{out} 分别为数组变量 Var 在语句 s_i 使用前后下标表达式的区间变化。

定义 2　对任意数组变量 Var_i，其对应的整型区间由一个三元组构成，即 $<varname,len,r>$。其中，*varname* 表示数组变量名称；len 表示数组定义时的长度；r 表示在执行语句 s_i 前值为 r_{in}，在执行语句 s_i 后值为 r_{out}。

定义 3　数组变量 Var_i 的下标表达式对应的整型区间集为 AS_i，整型变量 v_i 对应的整型区间集为 IS_i。

为了能直观的反映整型区间集变化情况，定义以下操作函数：*GetRange*(v)表示获得变量 v 的区间；*SetRange*(v)表示设置变量 v 的区间；*CaculateExprRange*(*expr*)表示计算表达式 *expr* 的区间，常数、变量都可以看作表达式；*GetIndexExpr*(v)表示得到数组变量 v 目前对应的下标表达式；*MergeRange*($v.r_1,v.r_2,v.r_3,\cdots,v.r_i,\cdots,v.r_n$)表示区间融合，返回结果表示在程序分支交汇点处变量对应的区间，其中 $v.r_i$ 表示变量 v 在程序分支点 i 处的区间，区间融合函数返回分支交汇后的变量 v 的区间，结果是各分支区间的并集；*Access*(v)表示访问变量 v；*Add*(e,S)表示将元素 e 添加到集合 S 中；*Del*(e,S)表示将元素 e 从集合 S 中删除；*ChangeStatus*(e,e')表示将变量对应的状态由 e 改变为 e'，主要指变量对应区间的改变；*GetStatus*(v,S)表示取得变量 v 在集合 S 中的状态，返回值是 v 在 S 中的表述状态。*LookupVar*(v,S)表示在集合 S 中查找变量 v，如果存在变量 v，则函数返回变量 v，否则返回 NULL；*UpdateSet*(S)表示更新集合 S，完成对 S 的添加、更改操作，返回值为修改后的集合 S'；*IsEmbody*(r,r')表示区间 r' 是否包含区 r，如果包含则返回 true，否则返回 false；*GetStatementType*(s)表示得到语句的类型，返回值为语句类型，s 为表达式语句，语句类型包括：变量声明语句(DeclStmt)、赋值语句(AssignStmt)等等，语句类型可以声明为一个枚举变量。

数组下标表达式整型区间集 AS 生成规则如下：(1) $\phi\in AS$；(2) $\forall s_i\in ST$ 如果 *GetStatementType*(s_i)==*DeclStmt*，且 s_i 形如：$Tv[len]$ 或 Tv[]=new $T[len]$（T 表示数组类型，v 是数组变量，len 表示数组长度）或其它类型的数组变量声明，声明时数组长度为 len，则 $Add(<v,len,[0,0]>,AS)$，$Add(<v,len,[0,len-1]>,AS)$；(3) $\forall s_i\in ST$，$\exists v\in V_{arr}$，如果 $Access(v)$，且 e=*GetIndexExpr*(v)，则：①若 e 为常数 c，则 *LookupVar*(v,AS)，且 r_{in}=*Ge-*

$tRange(LookupVar(v,AS))$，则更新 v 的区间 $ChangeStatus(\langle v,len,r_{in}\rangle,\langle v,len,[c,c]\rangle)$，并进行操作 $Add(\langle v,len,[c,c]\rangle,AS)$；②若 e 为变量，且 v 的区间为 r_{in}，则从变量 e 的声明结点为起始点，以引用数组变量 v 的语句结点为终点，广度优先遍历程序控制流图，按照区间运算的定义，计算变量 v 新的整型区间 r_{out}。即：$ChangeStatus(\langle v,len,r_{in}\rangle,\langle v,len,r_{out}\rangle)$，并进行操作 $Add(\langle v,len,r_{out}\rangle,AS)$；③若 e 为表达式，则分别计算表达式中各个变量的整型区间，在根据区间运算法则计算表达式的整型区间即 $r_{out}=CaculateExprRange(e)$，$ChangeStatus(\langle v,len,r_{in}\rangle,\langle v,len,r_{out}\rangle)$，并进行操作 $Add(\langle v,len,r_{out}\rangle,AS)$。

（4）若 $\forall s_i\in ST$，$\exists v\in V_{arr}$，如果 $Access(v)$ 且：①s_i 为条件语句或 Switch 语句，且有 n 个分支，则分别计算第 $i(i=0,1,2,\cdots,n)$ 个程序分支上数组下标表达式的 $r_{outi}(i=0,1,2,\cdots,n)$，则在此语句的出口结点处，$r_{out}=r_{out1}\cup r_{out2}\cup r_{out3}\cup\cdots\cup r_{outn}$；②$s_i$ 为循环语句，则分别计算循环条件为真时循环体内程序执行路径上数组变量下标表达式的 r_{out1} 和循环条件为假程序执行路径上数组变量下标表达式的 r_{out2}，则在循环语句的出口处，$r_{out}=r_{out1}\cup r_{out2}$。

在计算得到 r_{out} 后，再进行 $ChangeStatus(\langle v,len,r_{in}\rangle,\langle v,len,r_{out}\rangle)$ 和 $Add(\langle v,len,r_{out}\rangle,AS)$ 操作。

以上是对数组下标表达式区间集 AS 的构造规则，数组区间集产生的过程动态反映了数组下标的变化。在实际测试过程中，可以根据具体的程序集对上述规则进行补充。在区间集的基础上给出数组越界的故障模型如下：

在程序 P 中，若 $\forall s_i\in ST$，若 $\exists v\in V_{arr}\wedge Access(v)$，且 $e=GetIndexExpr(v)$，则在 s_i 对应的程序控制流图结点处更新数组变量下标表达式区间集合为 $AS'=UpdateSet(AS)$，若对于中的数组变量 v，其在 AS' 中的某一结点处的状态为 $\langle v,len,r\rangle$，若 $r\not\subset[0,N-1]$，即 $IsEmbody(r,[0,len-1])$==false，则发生数组越界故障 F_{oob}。即：$\forall s_i\in ST\wedge\exists v\in V_{arr}\wedge Access(v)(\langle v,len,r\rangle\in AS\wedge(IsEmbody(r,[0,len-1])==\text{false})\Rightarrow F_{oob}$。

从定义中可以看出，对数组区间集中的任一元素 $\langle v,len,r\rangle$，要确保 $r\subseteq[0,len-1]$，即数组的下标要小于数组定义时的长度 len 并且不小于 0，否则就会发生数组越界，即由 $Ran'(Var)$ 和关系 $\subseteq$ 组成的序偶 $\langle Ran'(Var),\subseteq\rangle$ 不是格。

3.2 测试系统设计

系统设计的核心工作是对源程序进行词法分析和语法分析，生成被测程序的语法树和控制流图，并在此基础上通过一定算法查找数组越界故障。系统设计如图 3 所示。

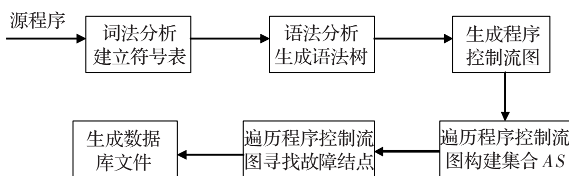

图 3 系统工作过程图

（1）词法分析：将预编译阶段产生的中间代码分解成单独的词的表示，形成各种符号表、类型表、关键字表、常数表、运算符表和分界符表。符号表中包含有利于错误查找的一些相关信息，如：位置、行号、错误类型，这些信息对于错误的查找和定位都有十分重要的作用。

（2）语法分析：这一步主要是将输入字符串识别为单词符号流，这里主要是找出变量声明语句，并相应的分离出各种类型的变量。按照标准的 Java 语法规则，对源程序作进一步分析，比如什么是变量定义、什么是赋值语句、什么是函数。语法分析的结果是生成语法树，每一个语法规则对应一个相应的处理函数，并作为语法树的一个结点挂在语法树上，提供对外的接口。

（3）生成控制流图：以语法树为基础，根据程序的控制结构，用递归的算法在遍历语法树时生成程序的控制流图。控制流图中的每个结点表示程序的一条语句，与语法树上相应的语句结点相对应。另外对每个函数的控制流图它还包含一个唯一入口结点和唯一的出口结点。在函数的调用的地方存在一个调用结点和一个返回结点。

（4）以第（2）步和第（3）步生成的语法树和控制流图为基础进行故障查找。从程序控制流图的入口结点开始，进行广度优先遍历。在遍历的过程中依据数组下标表达式整型区间集生成规则构建数组下标表达式整型区间集 AS，在数组变量的引用结点判断序偶 $\langle Ran'(Var),\subseteq\rangle$ 是否构成格。如果序偶 $\langle Ran'(Var),\subseteq\rangle$ 不满足格的条件，则在此结点必然会有数组越界的故障发生。

（5）如果找到相匹配的错误，则将错误信息记录到数据库文件中，最后得到数据库文件。数据库文件是得到的最终的结果，其中包括经过测试得到的一些相关信息，如数组变量名、该变量被定义的行号、该变量被使用的行号、错误类型等。最终由数据库文件生成错误信息报表，将错误信息反馈给用户。

4 实验结果和对比分析

以不同大小的 Java 程序模块来检测本文开发的 Java 数组越界静态软件测试系统。具体的作法是：首先运用软件测试工具对软件进行测试，标记出测试工具认为可能出现的错误点来，称之为检测点 IP（Inspection Points），经检测后提供一个中间文件给测试人员，由测试人员再对照这些 IP，复查源代码，最终确定这些 IP 点是否是一个 Defect。表 1 给出了对一些项目进行测试得出的 IP 数和经人工确认后的对比结果。

表 1 IP 数和确认以后结果对比

File Size/M	Inspection Points	Defects
5.27	13	4
13.70	10	5
32.40	47	28
64.00	79	36
183.00	132	52

File Size 指被测试程序的大小，以兆（M）为单位；Inspection Points 指软件测试工具检测出的 IP 总数；Defects 指经人工复查后确定的错误总数，这是最后要提交给客户的最终结果。由上面的实验结果可知，软件中存在的错误与文件大小成一定相关性。该测试软件能够找出被测程序的大部分数组越界的错误。

与 Reasoning 公司的测试结果进行了对比，如表 2 所示。

表 2 测试结果比较

项目	本文所述方法			Reasoning 工具		
编号	IP 总数	故障总数	准确率/(%)	IP 总数	故障总数	准确率/(%)
1	74	13	17.5	82	14	17.0
2	113	52	46.0	112	47	41.9
3	213	96	45.0	236	87	36.8
4	410	132	32.1	351	66	18.8
合计	810	293	36.1	781	214	27.4

（下转 108 页）