面向对象程序蜕变关系构造方法

张兴隆 于 磊 侯雪梅 侯韶凡
(中国人民解放军信息工程大学 郑州 450001)
(中国人民解放军信息工程大学数学工程与先进计算国家重点实验室 郑州 450001)

摘 要 针对面向对象软件类级测试中蜕变关系构造不充分的问题,提出一种由错误类型指导面向对象程序蜕变关系构造的方法。首先分析类方法中包含的基本操作,根据错误发生位置和作用效果将错误分成3类;其次按照执行效果的不同将类中方法分成两类;再针对错误类型对每个方法分别构造蜕变关系;最后通过Rectangle类的实验对比该方法与其他方法。实验证明提出的由错误类型指导蜕变关系构造的方法的检错率有所提高,并且有助于错误定位。

关键词 软件测试,蜕变测试,蜕变关系,变异测试
中图法分类号 TP311.5 **文献标识码** A

Method of Metamorphic Relations Constructing for Object-oriented Software Testing

ZHANG Xing-long YU Lei HOU Xue-mei HOU Shao-fan
(PLA Information Engineering University, Zhengzhou 450001, China)
(State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou 450001, China)

Abstract To solve the question of insufficiency in object-oriented metamorphic relations constructing, a method of metamorphic relations constructing for object-oriented software testing based on error types was proposed. Firstly, the basic operations in every method are analyzed and the faults are divided into three categories according to the location where errors occurred and the effect of errors. Then, the methods can be divided into two types according to the effect of implementation. Metamorphic relations are constructed for each method of every error type. Finally, we compared this method with other methods through the experiment of Rectangle. Experiment results show that new metamorphic relation construction method guided by error type has improved in the error detection rate and is helpful for finding fault location.

Keywords Software testing, Metamorphic testing, Metamorphic relation, Mutation testing

1 引言

软件测试是保证软件质量的一个重要环节,传统的测试方法存在局限性,例如无法很好地解决Oracle问题。Oracle问题是指在某些情况下,测试人员构造预期结果的代价太大,不能得到程序任意输入的预期结果,因而无法确定程序执行结果和预期结果是否相同。Chen等人提出了蜕变测试(Metamorphic Testing)的概念[1],提供了解决此类问题的新思路。该方法通过检查程序多个执行结果之间的关系来检验程序正确性,不需要构造预期输出。

在蜕变测试中,蜕变关系是蜕变测试的关键,用于产生测试用例和验证结果的正确性。蜕变关系的构造方法及选取准则目前有很多研究成果。Upulee等给出了一系列方法[2-8]。Chen等提出了一系列选取准则[9-12]。由于面向对象软件的基本构件是类,因此类测试是面向对象软件测试的关键。然而现有蜕变关系构造准则很多不能直接应用在面向对象软件类级测试中。例如,Chen提出两次执行尽可能不同的蜕变关系比较有效,而执行差异性的概念在面向对象程序构造的方法序列蜕变关系中还没有很好的定义。

侯雪梅等针对面向对象软件类级测试中方法序列调用时的Oracle问题,提出了一种基于代数规格的蜕变关系构造方法[13]。根据所提出的4条准则,将由公理导出的蜕变关系消除冗余,增加了蜕变关系充分性,弥补了GFT算法的不足。实验中发现上述方法对某些含有分支语句的方法检测率较低。统计整理数据发现:1)蜕变关系和变异体之间联系,特定形式的蜕变关系可以杀死某种错误特征的变异体;2)未发现变异体错误特征与其他不同,需要针对此类特征构造蜕变关系。

本文受国家自然科学基金项目(61402525),河南省科技攻关项目(162102210184)资助。
张兴隆(1992—),男,硕士生,主要研究方向为软件测试,E-mail:paper92@163.com;**于 磊**(1973—),男,博士生,副教授,主要研究方向为软件工程、软件质量管理;**侯雪梅**(1981—),女,硕士生,副教授,主要研究方向为软件测试、软件可靠性理论;**侯韶凡**(1991—),女,硕士生,主要研究方向为软件测试。

针对以上问题，提出了由错误类型指导蜕变关系构造的方法。首先，分析总结方法都包含的基本操作，根据其作用位置及其造成结果的不同合并成三大类错误类型，即对象状态错误、计算错误和分支判断错误；而后将每个方法针对 3 种错误类型分别构造最简形式的蜕变关系。这种由错误类型指导蜕变关系构造的方式相比其他方法在检错率上有相应的提高，并且有助于错误定位。

2 基本概念

定义 1(蜕变关系)[10] 假设程序 P 用来计算函数 f，x_1，x_2，…，x_n($n>1$)是 f 的 n 个变量。当 x_1，x_2，…，x_n 之间满足关系 r 时(r 为输入关系)，$f(x_1)$，$f(x_2)$，…，$f(x_n)$满足关系 r_f(r_f 为输出关系)，即：

$$r(x_1,x_2,\cdots,x_n)\Rightarrow r_f(f(x_1),f(x_2),\cdots,f(x_n))$$

成立，则称(r,r_f)是 P 的蜕变关系。

如果 P 是正确的，则其一定满足：

$$r(I_1,I_2,\cdots,I_n)\Rightarrow r_f(P(I_1),P(I_2),\cdots,P(I_n))$$

其中，I_1，I_2，…，I_n 是程序中对应于 x_1，x_2，…，x_n 的输入。若 r 是一个二元等式关系($r(x_1,x_2)$)，则称(r,r_f)是程序 P 的二元蜕变关系。

定义 2(原始/衍生测试用例)[10] 使用蜕变关系(r,r_f)测试程序 P 时，起初给定的测试用例是原始测试用例(利用其他测试用例生成办法获得)，记为 I；由原始测试用例根据关系 r 计算出的用例是该原始测试用例基于蜕变关系(r,r_f)生成的 P 衍生测试用例，记为 $FU=(I,(r,r_f))$。

定义 3(方法序列蜕变关系) 在面向对象的测试中，设 F 为类中方法的集合。$\_.p_i(),\_.q_j()\in F(i=1,2,\cdots,m,j=1,2,\cdots,n)$，$_.p_1().p_2()\cdots_.p_m()=_.q_1()\cdots.q_n()$或者 $\_.p_1().p_2()\cdots\_.p_m()==\_.q_1().q_2()\cdots.q_n()$，称该等式为方法序列组成的蜕变关系。

定义 4(属性方法) 不改变对象状态值，仅输出与对象属性或状态有关的方法，例如栈中的 $GetTop(S,\&x)$操作，读出栈顶元素，不改变栈中数据的值。

定义 5(主类方法) 构造函数和改变对象状态变量的方法，例如栈中的 $Push(\&S,x)$操作，进栈，改变栈对象的值，使得栈中数据加 1。

定义 6(运算符“≡”) 在方法序列蜕变关系中，设 F 为类中方法的集合。$\_.p_i(),\_.q_j()\in F(i=1,2,\cdots,m,j=1,2,\cdots,n)$，$\_.p_1().p_2()\cdots\_.p_m()\equiv\_.q_1()\cdots.q_n()$，运算符“≡”用来判断蜕变关系两边对象状态是否相等。

定义 7(运算符“=”) 在方法序列蜕变关系中，设 F 为类中方法的集合。$\_.p_i(),\_.q_j()\in F(i=1,2,\cdots,m,j=1,2,\cdots,n)$，$\_.p_1().p_2()\cdots\_.p_m()=\_.q_1()\cdots.q_n()$，运算符“=”用来判断蜕变关系两边执行结果是否相等。

定义 8(路径对)[10] 假设测试程序为 P，蜕变关系为 MR，以原始输入 I、衍生输入 $FU=(I,(r,r_f))$运行 P 时执行到的路径分别为 $Path_1$，$Path_2$，则称 $Path_1$，$Path_2$ 是基于 MR 的一个路径对，记作 $PairPath:Path_1\overset{MR}{\leftrightarrow}Path_2$，且交换 $Path_1$，$Path_2$ 两者等价。

3 面向对象类测试蜕变关系构造方法

3.1 错误类型分类

在一个类中，某个方法一般包含以下几个操作中的一个或多个：输入操作、输出操作、赋值操作、分支判断操作、循环操作。

输入操作为实现方法的输入信息功能，一般格式为 INPUT：变量。在输入语句中错误一般出现在变量上。

输出操作为实现方法的输出结果功能，一般格式为 PRINT：表达式。输出语句输出常量、变量或表达式的值及字符，错误一般出现在变量上。

赋值操作是将表达式所代表的值赋给变量，一般格式为变量＝表达式。赋值语句右边可以是一个数据、常量和算式，左边为变量。错误可能出现在变量、表达式中。

分支判断操作的作用为处理条件分支逻辑结构，基本格式如下。

```
IF 条件 THEN
    语句 1
ELSE
    语句 2
END IF
```

```
IF 条件 THEN
    语句
END IF
```

基本格式中的语句为其他操作中的一个或多个，错误分析与其他操作一致，主要考虑条件判断语句，即考虑错误在变量、逻辑运算符和关系运算符上。

循环操作的作用是处理循环逻辑结构，基本格式如下。

```
WHILE 条件
    循环体
WEND
```

```
DO
    循环体
LOOP UNTIL 条件
```

循环语句和条件判断语句类似，主要分析判断条件，循环体中的语句不再考虑。

在对错误进行分类时考虑发生的位置和产生的影响。由上面对 5 种基本操作错误分析可知，错误发生的位置概括为：变量、变量之间和判断条件。

变量包含类状态变量和局部变量，所有方法都有可能对状态变量进行修改，因此每个方法执行后都要对状态变量进行修改，故将其单独作为一类错误。局部变量在一个方法中的影响和表达式中的错误一致，都会造成结果错误，将其归为一类。虽然有些判断条件错误也涉及变量或变量间操作，但此处我们考虑对执行路径的影响，将其划分为一类。

综上，将方法中可能发生的错误分为 3 类：1)状态变量修改错误；2)输出修改错误；3)分支条件修改错误。

3.2 构造不等形式蜕变关系

实验中发现利用文献[13]的方法构造蜕变关系对某些含有分支方法的检错率较低，这是因为与蜕变关系相关的源和衍生测试用例执行路径的差异性太小，表现在两方面：1)源和衍生测试用例执行路径相同；2)基于蜕变关系的路径对过于单一。

当第三类错误发生时会出现两种情况：1)使得两个测试用例执行不同路径；2)使得两个测试用例都执行另外一条相同路径。情况 1)检出错误的可能性大于情况 2)，因此构造不

等蜕变关系，使得产生路径对更加多样化，提高情况 1)发生的概率。

图 1 是某程序的控制流图，假设该程序有 4 条可执行路径，分别为 $Path_1$，$Path_2$，$Path_3$，$Path_4$，MR 为该程序的一条二元蜕变关系($r(x_1,x_2),r_f$)。由图 1 可知分支判断节点为节点 0,1,3，规定判断条件取真时变量取值范围为 T_i，取假时为 F_i，i 为节点编号。则 4 条路径变量的条件组合为 $Path_1$：$T_0T_1T_3$；$Path_2$：$T_0T_1F_3$；$Path_3$：T_0F_1；$Path_4$：F_0。构造不等蜕变关系要满足不等关系成立，增加路径对的多样性。假设图 1 程序有 MR：$((a',b')=(b,a),r_f)$，根据每条路径变量的取值范围，将两路径变量之间的转换关系 r 作为由源测试用例得到的衍生测试用例的转换关系，而后将 MR 等式关系取反得到新的不等蜕变关系。例如三角形程序中有 MR：$(a',b',c')=(2a,2b,2c)$，路径 $Path_1$ 为一般三角形，变量约束为：$a+b>c,a+c>b,c+b>a,a\neq b\neq c$；$Path_2$ 为等腰三角形，变量约束为：$a+b>c,a+c>b,c+b>a,a=b\neq c$。可以得到两路径之间的变量转化关系 r：$a=b=\max(a,b)$，则 MR 的不等蜕变关系为 $!MR$：$(a',b',c')\neq(2\max(a,b),2\max(a,b),2c)$ $(a,b,c)\in Path_1$。

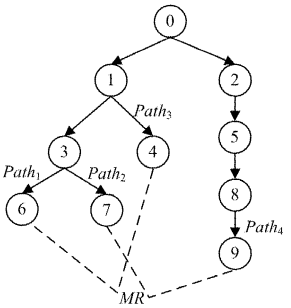

图 1　路径示意图

例如当 I_1 执行 $Path_1$，$FU=(I_1,(r,r_f))$也执行 $Path_1$，当有某第三类错误作用在图 1 方法中时会使 I_1 与 $FU=(I_1,(r,r_f))$都执行 $Path_2$。构造不等蜕变关系 MR_1 时，构造 I_1 执行 $Path_1$，FU 执行 $Path_2$。当发生上述错误时，让两测试用例都执行 $Path_2$ 行即可，满足此条件的测试用例构造简单。

3.3　蜕变关系构造步骤

假定给定类 C，方法 $\_.p_i():type_1\rightarrow type_2\in C(i=1,2,\cdots,m)$，左边为输入参数的类型，右边为输出参数的类型。得到类 C 的方法集合 Operations 和状态变量 x 和 y：

Class C

Operations

C()：type$_1$→type$_2$

_. p$_i$()：type$_1$→type$_2$(i=1,2,…,m)

Variables

x：type$_x$

y：type$_y$

(1)将方法集分成两类：属性方法和主类方法。所有方法都有可能错误地改变对象状态，所以都要进行第一类错误的检测。因为属性方法中每一个方法都不会对状态变量进行修改，故每个方法的最简蜕变关系为：$C(x_1,\cdots,x_n).p_i(x_i,\cdots,x_j)\equiv C(x_1,\cdots,x_n)$。主类方法中对于构造方法的第一类错误检测构造的蜕变关系如下：$C(x_1,\cdots,x_n).p_i(x_1',\cdots,x_n),\cdots,p_j(x_1,\cdots,x_n')\equiv C(x_1',\cdots,x_n')$，其中 $x_1,\cdots,x_n$ 为构造方法中的参数；p_i,p_j 为主类方法。其他主类方法构造蜕变关系如下：$C(x_1,\cdots,x_n).p_i(x_1,\cdots,x_i',\cdots,x_n)\equiv C(x_1,\cdots,x_i',\cdots,x_n)$。

(2)第二、三类错误主要根据方法或类的自身性质构造的蜕变关系进行检测。这类蜕变关系能使测试用例执行不同的路径，差异性较大。

(3)构造不等形式蜕变关系。遍历所有方法及其所构造的蜕变关系，分别构造蜕变关系。

按照以上步骤对 Rectangle 类构造蜕变关系。

1)首先得到包含方法和状态变量。

Class Rectangle

Operations

Rectangle(_,_) Integer Integer → Rectangle

setLength(_) Integer → Rectangle

setWidth(_)Integer → Rectangle

getLength() Rectangel → Integer

getWidth()Rectangel → Integer

getType()Rectangel → Integer

getArea()Rectangel → Integer

getPerimeter()Rectangel → Integer

Variables

l：Integer

w：Integer

属性方法：

getLength()Rectangel → Integer

getWidth()Rectangel → Integer

getType()Rectangel → Integer

getArea()Rectangel → Integer

getPerimeter()Rectangel → Integer

主类方法：

Rectangle(_,_) Integer Integer → Rectangle

setLength(_) Integer → Rectangle

setWidth(_)Integer → Rectangle

对每个属性方法构造第一类错误的最简蜕变关系：

$Rectangle(l,w).getLength()\equiv Rectangle(l,w)$

$Rectangle(l,w).getWidth()\equiv Rectangle(l,w)$

$Rectangle(l,w).getType()\equiv Rectangle(l,w)$

$Rectangle(l,w).getArea()\equiv Rectangle(l,w)$

$Rectangle(l,w).getPerimeter()\equiv Rectangle(l,w)$

主类方法蜕变关系：

$Rectangle(l,w).setLength(l').setWidth(w')\equiv Rectangle(l',w')$

$Rectangle(l,w).setLength(l')\equiv Rectangle(l',w)$

$Rectangle(l,w).setWidth(w')\equiv Rectangle(l,w')$

2)针对第二、三类错误构造蜕变关系。

$Rectangle(l,w).getLength()=l$

$Rectangle(l,w).getWidth()=w$

$Rectangle(l,w).getType()=Rectangle(n*l,n*w).getType()$

$n^2*Rectangle(l,w).getArea()=Rectangle(n*l,n*w)getArea()$

$2*n+Rectangle(l,w).getPerimeter()=Rectangle(n*l,n*w).getPerimeter()$

3)构造不等形式蜕变关系,分析 $getType$ 方法含有 3 个分支,只包含 $Path_1 \overset{MR}{\leftrightarrow} Path_1$,$Path_2 \overset{MR}{\leftrightarrow} Path_2$,$Path_3 \overset{MR}{\leftrightarrow} Path_3$ 路径对,没有不同路径间交互的路径对,路径对过于单一,需增加蜕变关系。$getType$ 方法 3 条路径的变量约束为:$Path_1$:$w=0 \parallel l=0$;$Path_2$:$w!=0\&\&l!=0\&\&w!=l$;$Path_3$:$w!=0\&\&l!=0\&\&w=l$。

得到 $Path_1 \overset{MR}{\leftrightarrow} Path_2$,$Path_1 \overset{MR}{\leftrightarrow} Path_3$,$Path_2 \overset{MR}{\leftrightarrow} Path_3$ 路径对约束转换条件:$r(Path_1 \rightarrow Path_2)$:$(w',l')=(w+1,l+1)$,$r(Path_1 \rightarrow Path_3)$:$(w',l')=(\max(w,l),\max(w,l))$,$r(Path_2 \rightarrow Path_3)$:$(w',l')=(l,l)$。构造如下不等蜕变关系:$Rectangle(l,w).getType()!=Rectangle(n*l',n*w').getType()$。

3.4 蜕变关系分析

(1)属性方法针对第一类错误构造的蜕变关系以 $getLength()$ 方法为例,蜕变关系为 $Rectangle(l,w).getLength()\equiv Rectangle(l,w)$,属性方法不改变状态变量的值,因此初始化后,两边任意一边构造函数后调用属性方法,“=”两边状态变量仍然相等,故可以检测出变异算子修改状态变量的错误。

主类方法针对第一类错误构造的蜕变关系以 $setLength()$ 方法为例,蜕变关系为 $Rectangle(l,w).setLength(l')\equiv Rectangle(l',w)$,主类函数改变状态变量值,初始化后,两边任意一边构造函数后调用主类方法需要根据方法自身性质进行语义检查,使“≡”两边成立。如果变异算子作用于 $setLength()$ 方法上,会使蜕变关系左边状态变量 l' 的值与右边不等,检测出错误。

(2)针对第二、三类错误构造的蜕变关系以 $getArea()$ 方法为例,在矩形中根据性质,当长和宽变为原来的 n 倍时面积变为 n^2 倍。当第二类变异算子作用于 $getArea()$ 方法时,虽然“=”两边都执行了该方法,但结果已不满足等式关系,故可以检测出错误。

(3)针对第三类错误构造的非等式蜕变关系,如 $Rectangle(l,w).getType()!=Rectangle(n*l',n*w').getType()$ 作为 $getType$ 等式蜕变关系的补充。等式蜕变关系可以检测出原本执行相同路径的测试路径执行不同路径,不等蜕变关系检测出原本执行相同路径的测试路径执行另外一条相同路径的错误。

4 实验分析

4.1 利用变异算子生成变异体

为了验证蜕变关系的检错能力,通常利用变异测试的方法进行分析。其基本原则是测试中所注入的故障(即变异)代表了程序员经常犯的错误。所有的变异体均根据变异算子生成,变异算子如表 1 所列[14]。

表 1 变异算子

变异算子	描述	变异算子	描述
AOR	算术运算符替换	COD	条件运算符删除
COR	条件运算符替换	AOI	算术运算符插入
AOD	算术运算符删除	ROR	关系运算符替换
COI	条件运算符插入	LOI	逻辑运算符插入
LOR	逻辑运算符替换	LOD	逻辑运算符删除

蜕变测试中运用变异体进行蜕变关系检测率的判断,运行结果一般有 3 种形式:1)蜕变关系运行变异体后源和衍生测试用例仍然满足蜕变关系,称为未发现该变异体;2)蜕变关系运行变异体后源和衍生测试用例不满足原来的蜕变关系,称为发现该变异体;3)变异体直接运行报错,称为发现该变异体。为了保证变异体生成全面覆盖变异算子,利用变异体生成工具 mujava 自动生成 $Rectangle$ 类变异体,生成的变异体类型多样全面,共 142 个,除去等价变异体共有 130 个变异体,如表 2 所列。

表 2 生成变异体

变异算子	个数	变异算子	个数
AOR	12	COD	1
COR	2	AOI	63
AOD	12	ROR	21
COI	4	LOI	15
LOR	4	LOD	8

4.2 度量指标

(1)在变异分析方法中,一般使用检错率 MS(Mutation Score)来评价检测结果,公式如下:

$$MS=\frac{M_k}{M_t-M_q} \tag{1}$$

其中,M_k 为测试用例检测出变异体的个数,M_t 为生成变异体总数,M_q 为等价变异体个数。由于已去除等价变异体,故在此 $MS=M_k/M_t$,测试中设置阈值,当检测率达到预定阈值时,则认为可以停止检测,设定阈值为 0.95。

(2)变异体蜕变关系发现率(Mutation Metamorphic Relations Failure-detecting Capability,MMRFC):

$$MMRFC(m)=\frac{N_m}{N} \tag{2}$$

其中,N 是蜕变关系条数,N_m 是 N 条蜕变关系作用于某个特定变异体 m 上变异体被杀死的个数。该度量公式描述某一个变异体有多少蜕变关系可以检测到。

4.3 文献[13]的中方法和现有方法的比较

根据文献[13]中提出的 4 条准则,构造蜕变关系如下:

$Rectangle(l,w).getLength()=l$;

$Rectangle(l,w).getWidth()=w$;

$Rectangle(l,w).setLength(l').getLength()=l'$;

$Rectangle(l,w).setWidth(w').getWidth()=w'$;

$Rectangle(l,w).setLength(l')\equiv Rectangle(l',w)$;

$Rectangle(l,w).setwidth(w')\equiv Rectangle(l,w')$;

$n^2*Rectangle(l,w).setLength(l').getLength().getType.setwidth(w').getWidth().getArea()=Rectangle(n*l',n*w').getArea()$;

$n*Rectangle(l,w).setLength(l').getArea().setwidth(w').getPerimeter()=Rectangle(n*l',n*w').getPerimeter()Rectangle(l,w).setLength(l').getPerimeter().setwidth(w').getType()=Rectangle(n*w',n*l').getType()$。

图 2 可知，文献[13]中的方法在 *Rectangle* 类测试中没有达到预定阈值，这说明测试不够充分。而本文方法对于 *Rectangle* 类的变异检测率为 0.962，达到阈值，验证了本文方法的有效性。

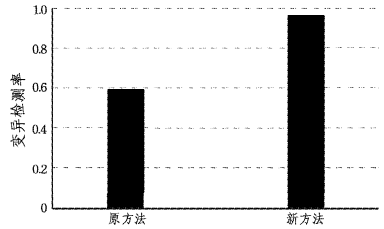

图 2　变异检错率

图 3 和图 4 中灰色柱状图分别表示文献[13]的方法和本文方法的结果。图 3 给出文献[13]方法生成的蜕变关系对于 *getArea* 和 *getType* 方法的检错能力较低，其他方法和本文方法检错率相等。这是因为在蜕变关系序列中 *getArea* 和 *getType* 方法在 *setwidth* 方法后，状态变量 w 更新后隐藏了两个方法对该变量错误修改的变异算子。文献[13]的准则并未对方法插入顺序进行规定，导致检错率较低。而本文方法生成的蜕变关系对于所有方法都有很好的检测率，有效避免了方法序列蜕变关系调用序列的问题。

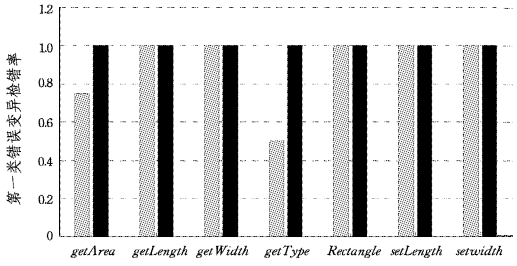

图 3　对于第一类错误两方法的变异检错率

从图 4 可知，在第二、三类错误中，文献[13]方法生成的蜕变关系对于 *getParameter* 和 *getType* 方法的检错率较本文方法更低。因为 *getType* 方法含有分支，且构造的蜕变关系使源和衍生测试用例执行相同路径掩盖了错误。根据文献[13]的准则 1 将 *getParameter* 方法插入到了 *getType* 蜕变关系中保证 *getParameter* 方法运行后对象状态能够被测试到；而 *getParameter* 方法中错误被 *getType* 覆盖，证明了本文方法的有效性。

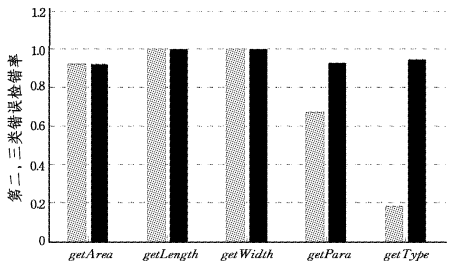

图 4　对于第二、三类错误两方法的变异检错率

由图 5 可知，有 54 个变异体 *MMRFC* 为 0，说明没有被任何蜕变关系检测出，而图 6 中只有 48，98，99 号变异体为 0，这说明本文方法的检错率要高于文献[13]中的方法。通过比较得知，图 5 中能够被至少两条蜕变关系发现的变异体占总发现数的比重较大，也即同一个错误能够被至少两条蜕变关系检测发现。而图 6 中大部分变异体仅能被一条蜕变关系检测发现。因为本文方法在构造蜕变关系时针对单个方法构造，目的性强，当蜕变关系发现错误时，相比于文献[13]中的蜕变关系，本文构造的蜕变关系可以较为方便地定位错误。

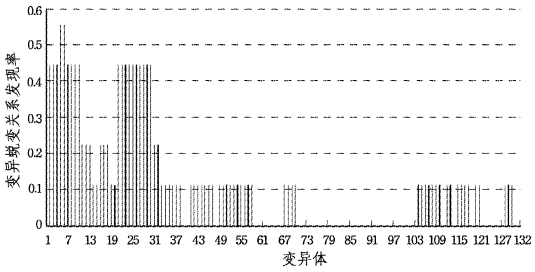

图 5　变异蜕变关系发现率

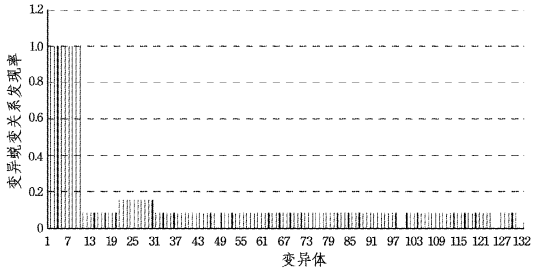

图 6　变异蜕变关系发现率

另外，由图 6 可知前 10 个变异体的 *MMRFC* 为 1，说明这 10 个变异体能够被所有蜕变关系检测发现，这是因为这 10 个变异体来自构造函数，所以任何蜕变关系都要包含构造函数。对应图 5 中发现 *MMRFC* 不到 50%，说明在检错过程中有近一半的蜕变关系对构造函数的错误进行了隐藏。

本文针对现有面向对象程序蜕变测试中出现的充分性问题，提出错误类型指导蜕变关系构造的方法。与其他构造方法的不同点在于本文对方法的针对性强，每个方法都对基本错误类型构造蜕变关系，当发现错误时也比其他方法更便于错误的定位。本文通过 Rectangle，SaveAcct 等类与其他构造方法进行了对比实验，实验结果表明，越复杂的类因为包含方法的种类多，本文方法的优越性越明显，且本文方法的检错率较其他方法更高。

参 考 文 献

[1] CHEN T Y，CHEUNG S C，YIU S M. Metamorphic testing：A new approach for generating next test cases：Technical Report HKUST-CS98-01[R]. Hong Kong University，Hong Kong，1998.

[2] KANEWALA U，BIEMAN J M. Using machine learning techniques to detect metamorphic relations for programs without test oracles[C]//International Symposium on Software Reliability Engineering. IEEE，2013：1-10.

[3] LIU H，LIU X，CHEN T Y. A New Method for Constructing Metamorphic Relations[C]//International Conference on Quality Software. IEEE，2012：59-68.

[4] GAGANDEEP，SINGH G. An Automated Metamorphic Testing Technique for Designing Effective Metamorphic Relations[M]// Contemporary Computing. Springer Berlin Heidelberg，2012：152-163.

（下转第 515 页）

总体看来，本文提出的调度策略对于计算密集型的应用效果不佳，如 Kmeans 算法和 PageRank 算法，这些算法需要大量的迭代运算，属于计算密集型作业。下一步，我们计划深入研究如何提高共享集群中计算密集型作业的性能。

结束语　在数据中心这样的共享集群中，往往运行着大量的数据处理作业，调度器的目的是提升集群的资源利用率和吞储量，集群的资源利用率和吞储量不但受调度速度的影响，也受调度质量的影响。调度的质量影响着作业的性能，进一步影响着集群的资源利用率和吞吐量。本文提出了基于数据块密度的调度策略，目的是提高共享集群中作业的数据本地性，从而优化其性能。该策略通过计算资源根据数据块的密度为作业进行分配，提升应用运行过程中的数据本地性，从而提升资源调度的调度质量。在实验测试过程中应用作业达到 90%的数据本地性，加快了应用的运行速度，并且可以有效降低集群的网络 I/O，对于数据密集型任务，比 Spread 策略缩短了 20%左右的作业运行时间。

基于数据块密度的调度策略能够在一定程度上提升作业的本地性。本文只对 Spark 作业进行了微基准测试，在下一步的工作中，我们将对更多的算法和计算框架进行测试，研究基于数据块密度的调度策略对不同类型算法性能的影响，并且深入研究如何提高共享集群中计算密集型作业的性能。

参 考 文 献

[1] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[2] WHITE T. Hadoop: The definitive guide[M]. O'Reilly Media, Inc., 2012.

[3] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets[J]. HotCloud, 2010, 15(1): 10.

[4] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]// 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). IEEE, 2010: 1-10.

[5] DELIMTROU C, SANCHEZ D, KOZYRAKIS C. Tarcil: reconciling scheduling speed and quality in large shared clusters[C]// Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015: 97-110.

[6] SCHWARZKOPF M, KONWINSKI A, ABD-EL-MALEK M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]// Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

[7] ZAHARIA M, BORTHAKUR D, SEN SARMA J, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[C]// Proceedings of the 5th European conference on Computer systems. ACM, 2010: 265-278.

[8] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[9] BEZERRA A, HERNÁNDEZ P, Espinosa A, et al. Job scheduling for optimizing data locality in Hadoop clusters[C]// Proceedings of the 20th European MPI Users' Group Meeting. ACM, 2013: 271-276.

[10] Kubernetes[OL]. http://kubernetes.io/

[11] HUANG S, HUANG J, DAI J, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis[C]// 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2010: 41-51.

[12] Apache Hadoop 官方网站[OL]. https://hadoop.apache.org/

[13] 浙江大学 SEL 实验室. Docker——容器与容器云[M]. 北京：人民邮电出版社，2016.

[14] 孙瑞琦，杨杰，高瞻，等. 一种提高虚拟化 Hadoop 系统数据本地性的资源调度方法[J]. 计算机研究与发展，2014(S2)：189-198.

[15] Docker[OL]. https://www.docker.com.

[16] REY J, Cogorno M, Nesmachnow S, et al. Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop[C]// IEEE International Conference on Cloud Engineering. 2015: 369-376.

(上接第 489 页)

[5] HUI Z W, HUANG S. A Formal Model For Metamorphic Relation[C]// Proc. of Decomposition Software Engineering (WCSE 2013). IEEE, 2013: 64-68.

[6] ASRAFI M, LIU H, KUO F C. On Testing Effectiveness of Metamorphic Relations: A Case Study[C]// 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement. IEEE Computer Society Washington, DC, USA, 2011: 147-156.

[7] KANEWALA U, BIEMAN J M. Using Machine Leaning Techniques to Detect Metamorphic Relations for Programs without Test Oracles[C]// Proceeding of the Software Reliability Engineering (ISSRE 2013). IEEE, 2013: 1-10.

[8] HUI Z W, HUANG S. A Formal Model For Metamorphic Relation[C]// Decomposition Software Engineering (WCSE 2013). IEEE, 2013: 64-68.

[9] CHEN T Y, HUANG D H, TSE T H, et al. Case studies on the selection of useful relations in metamorphic testing[C]// Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004). IEEE, 2004: 569-583.

[10] 董国伟，聂长海，徐宝文，等. 基于程序路径分析的有效蜕变测试[J]. 计算机学报，2009，32(3)：1002-1013.

[11] MAYER J, GUDERLEI R. An empirical study on the selection of good metamorphic relations[C]// Proc. of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06). Chicago, USA, 2006: 475-484.

[12] CAO Y, ZHOU Z Q, CHEN T Y. On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions[C]// International Conference on Quality Software. 2013: 153-162.

[13] 侯雪梅，于磊，张兴隆，等. 面向对象软件测试的蜕变关系构造方法[J]. 计算机应用，2015，35(10)：2990-2994.

[14] OFFUTT A J, LEE A, ROHERMEL G, et al. An experimental determination of sufficient mutant operators[J]. ACM Transactions on Software Engineering & Methodology, 2000, 5(2): 99-118.