基于非结构化控制流的进化测试方法

江 胜 卢炎生

(华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 结构性测试中,进化测试是一种高效的自动生成高质量测试用例的技术。然而,当程序中出现非结构化特征时,在面向节点的测试标准下,其效率极其低下甚至不及随机测试生成方法。在考虑循环体内部出现跳转(goto、return)语句的情况下,提出了一种适应度计算方法。该方法在结合传统进化测试适应度计算的基础上,充分考虑了循环次数对于进化搜索的影响。实验结果表明,本适应度函数可以很好地引导进化搜索,并以较小的代价生成测试用例。

关键词 进化测试,非结构化控制流,循环,适应度,进化搜索

中图法分类号 TP301 **文献标识码** A

Method of Evolutionary Testing Based on Unstructured Control Flow

JIANG Sheng LU Yan-sheng

(School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract Evolutionary testing is a highly effective technique for automatically generating high quality test data, which is used for structural testing. However, under the criterion of Node-oriented, testing of unstructured programs is inefficient and leading the technique degenerates to random testing. In this paper, with regard to the unstructured programs that contain arbitrary jump statement inner a loop body, a method of fitness calculation based on traditional approach was proposed, in which the impact of the number of iteration for evolutionary search was adequately considered. The experiments were then presented and the results show that the fitness function could effectively guide evolutionary search to find required test data at low cost.

Keywords Evolutionary testing, Unstructured control flow, Loop, Fitness, Evolutionary search

1 引言

传统的软件测试往往花费大量的人力物力,其代价通常占整个软件开发成本的50%左右[1]。通过软件测试的自动化可以显著地降低软件开发成本,而自动化测试面临的最主要问题就是测试用例的自动生成。

进化算法基于对自然基因的处理和达尔文生物进化理论,它代表了一类自适应的搜索方法[2]。最常见的进化算法是遗传算法。进化测试(Evolutionary Testing)是一种应用进化算法的软件工程技术,近年来该方法已多次被重复证明是自动化测试生成的有效途径[5,11,13-16]。进化测试的主要思想在于将生成测试用例的问题转化为一个进化搜索问题,图1展示了应用进化算法生成测试用例的一般步骤。进化测试时,将一串独立的编码作为一个个体,针对个体编码的解码即作为一个程序输入,并使用该输入调用实际程序运行。而适应度函数用来计算个体的适应度,适应度用以表示个体接近测试用例间的程度,测试过程依据其值对每个候选个体进行评价。在种群的进化过程中,选择、交叉、变异等进化算子得以应用于个体,采用适者生存的原则,直至找到最优个体或达到最大种群数。一般地,被测程序的输入空间则组成了该测试数据的搜索空间。

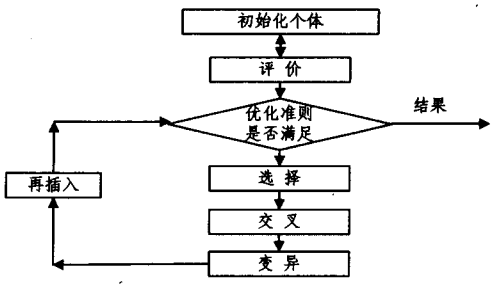

图1 进化算法用于测试用例生成结构简图

进化测试作为一种高效的结构性测试方法,应用中仍面临很多问题。在面向节点的测试准则下对非结构化程序进行测试就是其中之一[11]。当程序中引入任意跳转语句(Arbitrary Jump Statement)后,比如goto(此外作为goto语句的衍生还包括return,break和continue等),程序的控制流呈现非结构化形式[8]。任意跳转语句的使用使程序的控制流变得复杂和不明确,尤其在循环体内部出现时,循环的边界难以确定,数据依赖和控制依赖分析变得更加困难,整个程序的控制

到稿日期:2008-10-08 返修日期:2008-12-25 本文受"十一五"部委预研项目(513150601)资助。

江 胜(1976-),男,讲师,主要研究方向为软件测试等,E-mail:jwt@mail.hust.edu.cn;卢炎生(1949-),男,教授,博士生导师,主要研究方向为软件测试、特种数据库、数据挖掘。

流呈现“意大利面条”(SpaghettiCode)形态[7,8]。由于进化搜索的适应度函数依赖于数据和控制依赖分析,于是非结构化控制流带来进化测试的效率低下[6,12]。因此必须针对此类应用程序设计合理的适应度函数,为进化搜索提供有利于搜索进展的导向。

本文第2节介绍背景知识及相关问题;第3节进行定义,并提出一种适应度函数计算方法以及根据进化测试需要对程序进行插桩和转换的算法;第4节给出实验及其结果分析;最后进行总结。

2 背景及相关问题

2.1 进化测试

结构性测试广泛用于工业实践和大多数的软件开发标准,传统结构性测试可采用的标准有语句覆盖、分支覆盖和条件覆盖等[1]。将进化算法用于结构性测试,就是在一定的测试覆盖标准下,使用进化搜索技术生成测试用例。

在以面向节点的测试覆盖标准进行进化测试时,其个体适应度的计算通常包含两个部分,即接近层次(Approximation Level)和分支距离(Branch Distance)[11],如式(2)所示。Approximation Level 表示输入个体的执行路径与目的节点间距离的一个度量,其计算牵涉到关键分支(Critical Branch),关键分支指的是导致目的节点无法达到的分支[4]。图2是一个带有关键分支的控制流程图示例,其中节点6是目的节点,其关键分支包括节点1的T分支和节点4的F分支。节点5的F分支同样导致在循环体内无法到达节点6,也是一个关键分支。Approximation Level 的值为当前节点与目的节点间关键分支的个数减去1。例如,节点5的F分支 Approximation Level 大小为 $1-1=0$,节点4的F分支 Approximation Level 为 $2-1=1$。

当进化测试驱动程序执行至某节点并按某一关键分支执行时,此时适应度的第二部分即计算 Branch Distance。Branch Distance 的值描述了对当前分支而言,当前个体与满足要求输入间(由分支谓词决定)的接近程度。例如节点5,假设其分支条件为 $x==y$,当其F分支得以执行,则节点5关于此分支条件的 Branch Distance 为 $d=|x-y|$。当进化搜索满足 $d=0$,即 fitness=0 时,节点5的T分支得以驱动执行,达到测试生成目的。

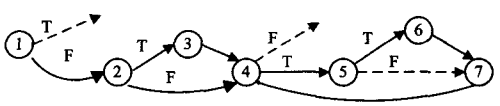

图2 一个带有关键分支的控制流程图示例

$$\text{normalize}(d)=1-1.001^{-d} \quad (1)$$

$$fitness=approximation\ level+normalize(d) \quad (2)$$

式(1)中,$d=$ cost(P),并为一个非负值,P 为分支条件,cost(P)表示满足 P 的 Branch Distance。反之,若$\neg P$需要满足以覆盖相应分支,则 $d=$cost($\neg P$)。normalize(d)是一个最小化过程,并将 d 规约至[0,1)。针对不同分支谓词 d 的计算参见文献[10]。而 Approximation Level 则牵涉关键分支的计算[4]。一般地,在优化过程中用于进化测试的适应度也是一个最小化过程,直至找到测试用例,或者达到最大种群数而启发式搜索过程失败。

在进化测试中,适应度函数的构造基于被测应用程序的数据和控制依赖分析[11]。一个好的适应度函数应该满足两方面的要求。首先,在一定的测试标准下适应度函数应提高生成测试用例的机会,其次适应度函数应较好地引导进化搜索,并以较少的代价生成测试用例[14]。

2.2 包含非结构化控制流的进化测试问题

自从文献[11]提出用于进化测试的适应度函数后,大量研究工作使用此方法进行适应度的计算并取得良好的效果。但在考虑循环体内部包含跳转语句,而使程序的控制流呈现非结构化特征时,还没有相关的研究出现。考虑图3所示的两个例程,由于循环体内均包含跳转语句,当 goto 或 return 语句得以执行时,位于循环体后的目的节点都将无法到达。对于例程1所示带循环体的非结构化控制流,在文献[6,12]中均有相应例程。此外在商用应用程序中,也存在大量实例。如 Linux2.6.14 源码 kernel 部分(约占总代码 0.7%),经统计,包含非结构化控制流(循环体内含有 goto 和 return 语句)的函数共有 107 处(for 循环 75 次,while 循环 32 次)。

例程1

```
#define ARRAYSIZE 10
int LoopFunc (char a[ARRAYSIZE] ){
    int rt;
① for(int i=0;i< ARRAYSIZE;i++){
②   rt=i;
③   if(a[i]!=0)
④     goto out;
    }
⑤ printf("Find!");//目的节点
⑥ return -1;
  out:
⑦ printf("Jumped to stop. i=%d\n",rt);
⑧ return rt;
  }
```

例程2

```
#define ARRAYSIZE 10
int Find(char a[ARRAYSIZE],
        char b[ARRAYSIZE]){
  int i = 0;
  bool flag=true;
  while(flag){
    if(a[i]>b[i])
      return i;
    i++;
    flag=(i<ARRAYSIZE);
  }
  printf("Find!");//目的节点
  return -1;
}
```

图3 两个包含非结构化控制流的简单例程

例程1中,函数 LoopFunc 中循环体内部含有 goto 语句,目的节点位于循环体后,函数输入为数组 a。循环中对于某个 i,若 $a[i]!=0$,则程序流程转至节点7,不能到达目的节点。现考虑测试用例如表1所列,用例1和用例2分别在 $i=3$ 和 $i=4$ 时使得条件 $a[i]!=0$ 为真,此时启动进化搜索。然而按照传统方法 $fitness_1=fitness_2$,即搜索没有取得进展。但对于测试生成而言,用例2明显比用例1更接近我们需要的测试用例。也就是说,在 goto 语句没有被执行时,循环次数越高。用例的质量就越高,反之,循环次数越低,就越难达到测试生成的目的[12,13,15]。情况更糟的是,用例3比用例1更优,反而 $fitness_1<fitness_3$。由此,在对包含 goto 语句(或例程2中的 return 语句)的循环体进行进化测试时,传统方法显然不能较好地引导进化搜索向利于测试生成的方向发展,而需要针对此类应用程序改进其适应度计算方法。

从表1可以看出,传统方法下 Approximation Level 计算失效,而只有用于局部优化的 Branch Distance 影响适应度。

表1 传统方法下用于例程1的测试用例及其适应度计算

	用例	进化搜索触发时机	接近层次	分支距离	适应度
1	a{0,0,0,1,1,1,1,1,1,1}	i=3	0	1	0.001
2	a{0,0,0,0,1,1,1,1,1,1}	i=4	0	1	0.001
3	a{0,0,0,0,3,1,1,1,1,1}	i=4	0	3	0.003

根据文献[9]中关于后支配和控制依赖的描述,若例程1中去掉节点4,即图4中“Jump”边将不存在,则此时目的节点5并不控制依赖于节点1(因为节点5后支配于节点1);而当“Jump”边存在时,由节点1存在一条边到达终止节点E而不

经过节点5,即节点5不再后支配于节点1。此外,节点5后支配于分支〈1,5〉,所以节点5控制依赖于节点1。

例程1的传统进化测试效率低下,其原因就在于目的节点控制依赖于循环体,而这种控制依赖关系在适应度计算中没有得到充分考虑。鉴于以上关于非结构化控制流的进化测试问题,传统方法中适应度不能很好地引导进化搜索。参照文献[12,13]提出的思想:适应度计算需要考虑循环次数带来的影响,即产生循环次数高的用例质量就越高,反之其质量就越低,并提出了一种新的适应度计算方法。

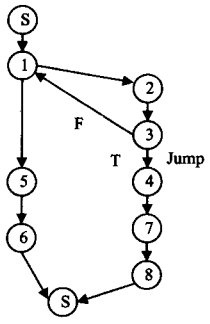

图4 例程1的控制流程图(CFG)

3 非结构化程序的进化测试

3.1 定义

为讨论方便,本文中结构化和非结构化程序定义如下。

定义1 构化程序(Structured Program) 一个结构化程序使用如下的代码结构组成:

1)顺序执行语句;

2)条件语句(如if…else…,case等);

3)循环;

4)结构化的子程序调用。

定义2 非结构化程序(Unstructured Program) 一个非结构化程序是循环体内部使用了如下代码结构:

1)goto语句(排除使用goto实现定义1中前3种结构,goto为向后跳转至循环体外);

2)return语句。

依据定义2,非结构化控制流即非结构化程序的控制流程。非结构化控制流虽然在很多情况下可以减少重复代码、避免深层嵌套、提供好的异常处理方法,甚至可以为编码提供更好的灵活性,但同时存在着程序的可读性差、程序控制结构变得复杂、难于测试和维护等弊病[10]。在很多有关非结构化程序的研究中,往往采用程序转换的方法,从而提高程序的可读性。或为了达到编译器优化等目的,例如文献[7]中提到的使用break语句和附加变量来替换goto语句的使用。本文研究的目的在于通过对非结构化程序的转换和插桩,从有利于测试生成的角度对源程序进行改造,提高其易测性。

根据文献[4]中关于Critical Branch的定义,从表2中可以看出每个用例的Approximation Level均为0,即体现控制依赖的Approximation Level并未对适应度产生影响,仅仅只有用于局部优化的Branch Distance决定适应度,从而导致传统方法失效。我们针对此类非结构化程序提出了一个度量方法,即循环距离(Loop Distance)来描述测试用例与目的节点间的"距离"。

定义3 循环距离(Loop Distance) 循环距离描述经过循环达到目的节点的接近程度。假定循环L为while(P){…},n为执行测试时的当前循环次数,则循环距离是一个关于P和n的函数:LoopDistance(P,n),记为ℓ。其计算方法如下:

$$\ell=\alpha\times \text{Branch\_Dist}(\rightarrow P)/n \tag{3}$$

$$\text{fitness}=\text{LoopDistance}(P,n)+\text{normalize}(d) \tag{4}$$

式(3)中,α为系数,Branch_Dist($\rightarrow P$)表示循环谓词的Branch Distance,其值即式(1)中的d。ℓ的值反映到达目的节点的接近程度。当P成立时,循环次数越高,ℓ越小;当$\rightarrow P$成立时,Branch_Dist($\rightarrow P$)=0,ℓ=0。程序代码中L可能不是一个条件循环,如for循环,则可以采用一定方法将其转换为while(P){…}形式。由式(3),改进的进化测试适应度计算为式(4)。该式中,LoopDistance(P,n)是对当前个体与目的节点间接近程度的度量,而normalize(d)则针对跳转语句的分支谓词进行局部优化。如例程1中,当节点2的true分支得以执行,则启动进化搜索直至找到可行解,而Loop Disatance(P,n)用于该进化搜索,并为其提供搜索的导向。

Tracey等人试图直接以循环谓词的Branch Distance表示测试用例经过循环到达目的节点的接近程度[10]。然而,当循环谓词不能提供任何进展时,如例程2中循环谓词为一个布尔变量,根据Tracey在文中针对Branch Distance的定义,此时该方法亦不能引导进化搜索向利于测试生成的方向发展。

通过引入循环距离的概念,本文给出的方法在适应度计算中充分考虑循环次数带来的影响,并引导进化搜索取得较好的进展,提高其测试生成效率和效果。

3.2 非结构化程序转换和插桩算法

进化测试依赖于被测程序的执行。当一个偏离目的节点的分支被触发,则启动进化搜索,评价该个体的适应度,直至找到测试用例或到达最大种群数。进化测试中往往需要对程序进行必要的插桩,以便控制进化搜索流程[11]。为讨论方便和不失一般性,假定循环体内出现一个goto(或return)语句,循环体为有限次循环,不考虑嵌套循环。

算法1 针对非结构化程序的进化测试转换和插桩算法

输入:待测函数F

输出:插桩和转换后的F'

Step1 在F中添加float类型变量*fitness*和int类型变量*counter*于循环体前,并分别初始化其值为0;

Step2 在循环体首部插桩*counter*=*counter*+1;

Step3 若循环为计数型循环(如for循环),则转换该循环为while(P)形式;

Step4 在跳转语句(return、goto)前插桩*fitness*=$f(x)$;

Step5 在循环体末尾插桩*fitness*=$g(x)$;

Step6 输出F',算法终止。

$$f(x)=\textit{LoopDistance}(P,\textit{counter})+\textit{BranchDistance}(\rightarrow C) \tag{5}$$

$$g(x)=\textit{LoopDistance}(P,\textit{counter}) \tag{6}$$

算法1中,第1步对进化搜索需要的变量进行插桩,即*fitness*和*counter*,其中*fitness*用于适应度计算,*counter*变量用于在循环体首部记录当前循环次数。

第2步将*counter*在每次循环开始时进行增量计算,以便用于*fitness*计算。

第3步对循环进行转换。若循环为条件循环(如while

(*P*)或 do…while(*P*)),则保持不变;若为计数型循环(如 for 循环),则转换为 while(*P*)形式。转换后可利用式(3)计算 *Loop Distance*。

第 4 步对任意跳转语句前插桩 $fitness=f(x)$。$f(x)$的计算参见式(5),其中 C 为跳转语句对应的分支条件。例程 1 中 C 为 $a[i]!=0$。*BranchDistance*($\neg C$)表示偏离当前分支的 *Branch Distance*,使用式(1)后的一个量化。如例程 1 中,当条件 $a[i]!=0$ 成立,则启动进化搜索,在局部搜索中 *normalize*(d)为一个最小化过程,同时 *LoopDistance*(P,*counter*)也随着循环次数的递增逐渐减小,因此 *fitness* 的值将获得良好的下降趋势,在全局上有利于搜索进展。

第 5 步在循环末尾插桩 $fitness=g(x)$。当$\neg P$ 成立,即有 $fitness=0$,则表示测试用例找到或可以达到目的节点,进化测试过程将停止。

第 6 步输出 F',程序转换和插桩算法终止。

算法 1 实现对非结构化程序的转换和插桩。表 3 右部即为例程 1 经转换和插桩后的程序代码。类似很多文献中提到的易测性转换(Testability Transformation)[5],算法 1 的目的是提高非结构化程序的易测性,便于依据程序结构构造合理的适应度函数。

4 实验及结果分析

使用 Matlab7.0 实现了一个进化测试系统 ETS1.0,该系统参照文献[2,11]中提到的编码、算子设计和算法实现。ETS1.0 运行环境为 Windows2003,Intel(R) Core(TM)2 Duo E65502.33GHz 台式机。实验中设定交叉概率为 0.2,变异概率为 0.05,用于式(4)的常量系数 $\alpha=5$。ETS1.0 针对本文描述的非结构化循环体进化测试,设定进化算法终止条件为适应度为 0(传统方法为找到测试用例)或者达到最大种群数 1000。试验针对例程 1(输入空间为[0,255])进行,该例程为一个数组搜索程序,对输入的数组进行判断。若没有非零元素,则返回 0;若存在,返回第一个非零元素的数组序号。

实验分别使用传统进化测试方法和改进方法进行。传统方法下,适应度计算采用式(2),即仅考虑 Branch Distance (Approximation Levels 均为 0);而后者使用式(4)。用于实验的两个不同例程版本如图 5 所示。每种方法分别执行 8 次,实验结果为进化搜索中适应度评价时适应度比较的次数。

传统方法

```
#define ARRAYSIZE 10
int LoopFunc (char a[ARRAYSIZE] ){
  double fitness=0;
  int rt;
  for(int i=0;i< ARRAYSIZE;i++){
    rt=i;
    if(a[i]!=0){
      fitness=Branch_Dist(¬(a[i]!=0));
      goto out;}
  }
  printf("Find!");//目的节点
  return -1;
out:
  printf("Jumped to stop. i=%d\n",rt);
  return rt;
}
```

新方法

```
#define ARRAYSIZE 10
int LoopFunc (char a[ARRAYSIZE] ){
  double fitness=0;
  int counter=0;
  int rt;
  int i=0;
  while(i< ARRAYSIZE){
    counter=counter+1;
    rt=i;
    if(a[i]!=0){
      fitness=f(x);
      goto out;}
    i++;
    fitness=g(x);
  }
  printf("Find!");//目的节点
  return -1;
out:
  printf("Jumped to stop. i=%d\n",rt);
  return rt;
}
```

图 5 例程 1 用于不同实验转换和插桩后的代码

表 2 展示了使用传统方法的进化测试结果。从该结果可以看出,由于其适应度函数只考虑局部优化,结果显示其进化搜索随机性较大,并导致其效率低下,以致出现类似表 1 中所示用例 1 和用例 2 的情况(适应度相等)。而这样的结果显然不能引导进化搜索朝利于测试生成的目的发展。并且当循环次数较大时,进化搜索达到最大种群数而搜索失败。

表 3 为使用新方法的进化测试结果。由于改进了其适应度函数,使得进化搜索在全局上获得较好进展,其效率得到很大提高,并成功生成测试用例。

表 2 例程 1 传统方法进化测试适应度比较次数

数组大小	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8
2	142	452	252	375	50	736	254	482
5	16854	6235	5542	6254	5426	2368	4578	8652
10	29236	28865	15584	28632	16786	16354	failed	failed
20	failed	failed	failed	failed	failed	failed	failed	failed
30	failed	failed	failed	failed	failed	failed	failed	failed

表 3 例程 1 新方法进化测试适应度比较次数

数组大小	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8
2	165	218	188	168	187	135	114	165
5	553	522	664	755	653	684	695	542
10	1065	1366	1166	1432	1542	1256	1687	1234
20	3995	3943	4046	4265	3564	3965	4015	4235
30	6116	7354	6278	8682	7542	7562	8452	7653

结束语 包含循环体的非结构化程序给面向节点的进化测试带来新的挑战,第一次提出该问题的是文献[12]。Wegener 提出问题的原型并分析指出适应度计算必须考虑循环次数带来的影响,但并未提供具体的适应度计算方法。本文是该文献一个有益的补充。针对包含循环体结构程序的结构性测试,Jones 首次提出适应度的计算需要考虑循环次数带来的影响[13],即循环次数高,则适应度应越优,但并未就非结构化控制流做出论述。Hierons 使用易测性转换(Testability Transformation)对包含多重退出(Mutliple-level exit)的非结构化程序进行重新构造[6],并使得转换后程序与原程序保持分支覆盖准则等价性,从而提高类似进化搜索等结构性测试方法的实用性。此外,Liu Xiyang 针对含有跳转语句的循环体内部出现 flag 变量时[15],通过对目的节点和循环体间数据依赖的分析,提出一种适应度计算规则。该方法详细讨论了循环体内部出现 break 和 continue 语句情况下如何构造适应度函数,以提高进化测试效率。

本文结合此前文献的研究基础,较全面分析了循环体内包含任意跳转语句(return,goto)的非结构化程序在面向目的进化测试中的问题。指出传统方法的失效原因在于忽略目的节点与循环体间控制依赖关系,提出用循环距离(Loop Distance)来弥补传统进化测试适应度计算方法的不足,并给出新的适应度函数。同时根据进化测试过程和适应度计算的需要,提出一个非结构化程序转换和插桩算法。实验结果表明我们的方法提高了进化搜索效率,并以较小代价成功地找到测试用例。

文中主要针对目的节点与循环体间存在的控制依赖关系进行讨论。当它们之间还存在数据依赖关系时(如文献[16]中的 flag 问题),本方法还需要进一步完善。其次,式(3)中使用的常量 α 在一定程度上体现了 Loop Distance 和 Branch Distance 在适应度计算中的权重比,对于不同的程序输入域

(下转第 181 页)

角色类之间的分离，因为角色类中的很多行为属性有可能是分散在多个应用程序类中的。本文中的方法使用角色建模设计模式时不仅仅只是把角色作为连接应用程序类和角色类之间关系的一个类，而是将设计模式中的各种元素都作为角色来定义，包括行为角色和属性角色以及关系角色都作为单独的类，这样将建模设计模式的元素从元层来考虑，从而实现应用程序类和角色类的彻底分离，达到设计模式不仅设计思想可以重用，代码也可以重用的目的。文献[10]没有对设计模式的演化及一致性验证问题进行探讨。文献[12]对模式的演化进行了研究，但没有给出形式化模型，也没有对演化后的一致性进行验证。也有一些对设计模式的精确建模进行研究的文献，如文献[13]。但这些研究都是从对 UML 扩充的角度来进行的。

参 考 文 献

[1] Gamma, et al. 设计模式——可复用的面向对象软件的基础[M]. 北京：机械工业出版社，2005

[2] Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ[J]. ACM SIGPLAN Notices, 2002, 37(11): 161-173

[3] 沈毅，缪淮扣，文志诚，等. 一个 Z 的证明责任产生器[J]. 上海大学学报：自然科学版，2005(11)

[4] 文志诚. 面向对象软件的形式验证技术[D]. 2006

[5] Meisels I, Saaltink M. The Z/EVES 2.0 reference manual[R]. TR-99-5493-03e. ORA Canada, Canada, 1999

[6] France R, Kim D-K, Sudipto G, et al. A UML-Based Pattern Specification Technique[J]. IEEE Transactions on Software Engineering, 2004, 30(3): 193-206

[7] Lauder A, Kent S. Precise Visual Specification of Design Patterns[C]// Proc. of ECOOP' 98, LNCS 1445. Springer-Verlag, 1998: 114-134

[8] OMG. UML2. 0superstructurespecification[OL]. http://www.omg.org/uml/

[9] Riehle D. Describing and Composing Patterns Using Role Diagrams[C]// Proc. of the Ubilab Conference' 96. 1996: 137-152

[10] Kim S K, Carrington D. Using integrated metamodeling to define OO design patterns with object-Z and UML[C]// Proc. of the 11th Asia-Pacific Software Engineering Conf. (APSEC 2004). Los Alamitos: IEEE Computer Society, 2004: 257-264

[11] 何成万，何克清. 基于角色的设计模式建模和实现方法[J]. 软件学报，2006(4)

[12] Dong J, Yang S, Zhang K. A Model Transformation Approach for Design Pattern Evolutions[C]// Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems. 2006: 80-92

[13] Jing Dong. UML Extensions for Design Pattern Compositions [J]. Journal of Object Technology, 1(5): 149-161. http://www.jot.fm/issues/issue_2002_11/article3

(上接第 152 页)

甚至于很大的输入空间，如何设定 α 值是提高进化搜索效率的一个重要因素。此外，当循环次数很大时，如何结合针对循环的优化方法[9]对进化搜索过程进行优化以保证测试的效率和效果；当循环体内包含多个 return 语句和循环存在嵌套时，如何改进现有方法。所有这些都会是后续研究的重点。

参 考 文 献

[1] Beizer B. Software Testing Techniques[M](2nd edition). Van Nostrand Reinhold, 1990

[2] Harman M. The automatic generation of software test data using genetic algorithms[D]. Pontyprid, Wales, Great Britain: University of Glamorgan, 1996

[3] Clark J, Dolado J J, Harman M, et al. Reformulating software engineering as a search problem[J]. IEEE Proceedings Software, 2003, 150(3): 161-175

[4] Korel B. Dynamic method for software test data generation[J]. Software Testing, Veri_cation and Reliability, 1992, 2(4): 203-213

[5] Harman M, Baresel A, Binkley D, et al. Testability Transformation-Program Transformation to Improve Testability[M]. Formal Methods and Testing, Lecture Notes in Computer Science, Springer-Verlag, 2008, 4949: 320-344

[6] Hierons R, Harman M, Fox C. Branch-coverage testability transformation for unstructured programs[J]. The Computer Journal, 2005, 48(4): 421-436

[7] Ramshaw L. Eliminating goto's while preserving program structure[J]. J. ACM, 1988, 35: 893-920

[8] Agrawal H. On Slicing Programs with Jump Statements[C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, 1994

[9] Ferrante J, Ottenstein K J. The Program Dependence Graph and Its Use in Optimization[J]. ACM Transactions on Programming Languages and Systems, 1987, 9: 319-349

[10] Tracey N, Clark J, Mander K, et al. An automated frame framework for structural test-data generation[C]// Proceedings of the International Conference on Automated Software Engineering. Hawaii, USA, 1998

[11] Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing[J]. Information and Software Technology, 2001, 43(14): 841-854

[12] Wegener J. Overview of Evolutionary Testing[C]// IEEE Seminal Workshop. Toronto, May 2001

[13] Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms[J]. Software Engineering Journal, 1996, 11(5): 299-306

[14] Baresel A, Sthamer H, Schmidt M. Fitness function design to improve evolutionary structural testing[C]// Proceedings of the Genetic an Evolutionary Computation Conference (GECCO 2002). New York, USA, 2002

[15] Liu Xiyang, Lei Ning, Liu Hehui, et al. Evolutionary testing of unstructured programs in the presence of flag problems[C]// Asia-Pacific Software Engineering Conference. TaiPei, TaiWan, 2005

[16] 艾丽蓉，赵庆兰，刘西洋，等. 面向 Java 语言地进化测试中分支依赖图的构建[J]. 计算机科学，2006, 33(7): 249-252, 285