维普资讯 http://www.cqvip.com

面向 Java 语言的进化测试中分支依赖图的构建*)

艾丽蓉[1] 赵庆兰[1] 刘西洋[2] 刘 洋[2] 刘鹤辉[2]

(西北工业大学计算机学院 西安 710072)[1] (西安电子科技大学软件工程研究所 西安 710071)[2]

摘 要 在面向对象程序的进化测试中,面向对象程序本身的状态特性使得一个分支的到达需要一定长度方法调用序列的执行,而封装特性则使得在一个对象外部无法直接调用其 private 方法或者 protected 方法。针对这两个问题,采用基于蚁群算法和遗传算法的动态搜索与静态分析相结合的方式提供了一种有效的思路。本文针对这种思路中的静态分析过程提出分支依赖图的概念,并设计、实现了针对 Java 语言的方法内分支依赖图以及类的分支依赖图的构建算法。

关键词 进化测试,状态问题,分支依赖

Construction of Branch Dependence Graph in Evolutionary Testing for Java

AI Li-Rong[1] ZHAO Qing-Lan[1] LIU Xi-Yang[2] LIU Yang[2] LIU He-Hui[2]

(Computer Department, Northwestern Poly-technical University, Xi'an 710072)[1]

(Software Engineering Institute, Xi'dian University, Xi'an 710071)[2]

Abstract In evolutionary testing, the execution of a sequence of method invocations is required to cover a given branch for the state characteristic of object-oriented programs, and private/protected methods can't be invocated directly because of the encapsulation. An effective solution to the both problems is to combine static analysis with dynamic search based on ant colony algorithm and genetic algorithm. Aiming at the static analysis in this approach, in this paper, the conception of Branch Dependence Graph(BDG) is proposed, and algorithm for constructing the BDG for Java is given.

Keywords Evolutionary testing, State problem, Branch dependence

1 引言

传统上,软件测试数据的生成通常是通过软件测试人员由手工构造的,这是一项非常繁重而且开销很高的劳动,并且消耗一半以上的软件开发和维护费用,因此测试数据的自动生成一直以来都是软件工程中的研究热点和难点。进化测试(Evolutionary Testing)[1]是近年来刚刚兴起的一种有效的测试数据自动生成技术,并已经被成功地应用于很多工程代码的测试数据生成中[2~6]。进化测试的基本思路就是将在不同覆盖准则下的测试数据生成问题转化成一个全局优化问题[1]。当为一个特定的测试目标搜索测试输入时,通过将被测程序的输入参数进行编码,并用每一串独立编码代表被测程序输入参数的具体取值,就可以通过对合适编码的搜索而进行测试数据的搜索,而被测程序的输入空间则组成了该测试数据的搜索空间。

在白盒测试中,结构测试(structure testing)是目前使用最普遍的测试类别,它要求根据一定的覆盖准则,如分支覆盖、节点覆盖、路径覆盖等为程序生成一个测试数据集。但由于软件本身的非线性特征,有些分支往往很难被覆盖到。然而,这些分支通常情况下却是隐含致命软件错误的角落分支,对这些分支,进化测试能够非常有效地为其自动生成测试数据。但是,由于面向对象测试中状态特性和封装特性的存在,单纯的进化搜索往往会受到阻碍而很难或者无法找到所需的测试数据。通常情况下,对于类的 public 状态变量可以通过对该变量的直接赋值进行改变,但是对于 private 或者 protected 变量的改变只能通过相应的方法调用序列来触发,这就是通常所说的状态问题。一般情况下,一个对象的 public 方法往往是较多的,但对一个存在状态依赖的目标分支,并不是所有的方法调用都能够改变其相应状态变量的状态,以影响这个特定分支的执行。更为严重的是,由于面向对象程序的封装特性,对其 private 方法或者 protected 方法无法直接进行测试,而只能通过分析 public 方法里的方法调用接口来进行测试。

针对面向对象程序进化测试中的这两个问题,本文作者在文[7]中提出了一种采用静态分析与基于蚁群算法和遗传算法的动态搜索相结合的方式对面向对象程序进行测试的思路,并设计了 APF(Ant Path Finder)算法。其整体思路如图 1 所示。

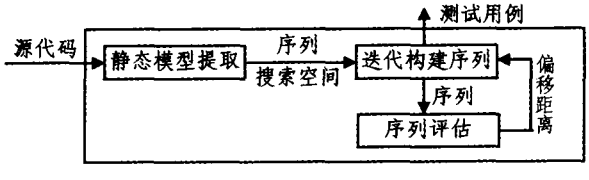

图 1 状态问题的整体解决方案

实现步骤如下:

*)"十五"国防预先研究项目"可信软件工程技术"(413150501)资助。**艾丽蓉** 硕士生导师,研究方向为软件工程、信息安全与智能处理;**赵庆兰** 硕士研究生,研究方向为进化测试;**刘西洋** 硕士生导师,研究方向为进化测试、模型检验;**刘 洋** 硕士研究生,研究方向为进化测试;**刘鹤辉** 硕士研究生,研究方向为进化测试。

维普资讯 http://www.cqvip.com

1)静态模型抽象(Static Model Abstraction)。动态搜索的目的是在众多能到达测试目标的调用序列中找到一个最短或者近似最短的调用序列,静态模型的抽象就是要为动态搜索提供一个尽可能小的搜索空间。静态分析阶段除了要得到源代码的控制流信息和数据依赖信息,还要对这些信息进行进一步的抽取,抽取出能够影响测试目标的覆盖或者对方法调用序列搜索有帮助的静态信息,求解影响状态分支的分支依赖关系并得到与 private/protected 方法状态相关的公共接口。

2)迭代构建序列(Iterative Construction of Sequence)。应用搜索算法,例如蚁群算法,反复迭代构建能覆盖给定测试目标的方法序列,直到找到最短或者近似最短的序列。

3)序列评价(Sequence Evaluation)。使用搜索算法,例如遗传算法,对于在第 2 步中构建的一个序列寻找一个评估参数,然后看测试目标是否已经被这个序列覆盖,通过执行这个序列返回当前序列和第 2 步的可行序列的分支距离。

由图 1 可以看出,在 APF 算法中,其核心思想是静态分析与动态搜索的相互结合。静态分析是动态搜索的基础,并为动态搜索提供了强有力的导向作用。对于找到方法的触发调用语句,可以通过采用求解方法调用链(Call Chain)的方法来解决[8]。对于状态问题,在静态分析过程中本文引入了分支依赖图。作为进化测试的第一步,在静态模型抽象时需要得到分支间的依赖关系,构建类的分支依赖图。

2 形式化定义

为了方便描述分支依赖图的构建,这里先给出与分支相关的形式化定义。

2.1 分支

简单地说,程序里具有相同执行条件的语句的集合构成了一个分支。例如,对图 2 中 Foo 类的所有分支进行编号,可以得到其分支号所对应的语句号。类 Foo 的分支号与语句号的对应表格如表 1 所示。

分支定义变量的集合:$def(n)=\{v|v$ 是分支 n 所改变了的变量的集合$\}$;分支引用变量的集合:$use(n)=\{v|v$ 是分支 n 所引用的变量的集合$\}$;分支执行条件的引用变量集合:$con_use(n)=\{v|v$ 是分支 n 的执行条件里所引用的变量$\}$。当且仅当这些变量满足一定的条件时,分支才能执行。当 $v\in con_use(n)$时,称分支 n 控制依赖于变量 v。

```
class Foo{
  private:
    int stat_qsize,nubukets;
  pulic:
    int qsize;
  public:
1:   Foo(){
2:     stat_qsize=0;
3:     nubukets=4;
4:     qsize=100;
     }
5:   void adjust (){
6:     if (stat_qsize>qsize)
7:       resize (nubukets*2);
     …//其余代码省略
  }
8:   void sub(int x){
9:     if (x<4)
10:        stat_qsize--;
       else
11:      stat_qsize=2;
     …
   }
12:  void add (int x){
13:    int temp=qsize;
14:    if (x<temp);
       {
15:      state_qsize++;
16:      temp++;
       }
17:    if (temp)
18:      //test goall;
       …
     }
19:  void resetMax(int x){
20:    qsize=x;
     …
   }
  private;
21: void resize(int nbuck){
22:    if (nbuck<4)
23:      nubukets=4;
       …
     }
…
}
```

图 2 类 Foo 的 Java 代码

表 1 类 Foo 的分支表

方法	分支	语句	方法	分支	语句
foo	n1	2,3,4	add	n7	13
adjust	n2	空		n8	15,16
	n3	7		n9	18
sub	n4	空	resetMax	n10	20
	n5	10	resize	n11	空
	n6	11		n12	23

2.2 分支的分类

从分支与变量的控制依赖关系上讲,分支可以分为以下 4 种:

直接状态分支,当且仅当(存在 v 是状态变量 $\wedge v\in con_use(n)$);

间接状态分支,当且仅当(存在 $v1$ 不是状态变量 $\wedge v1\in con_use(n))\wedge$(存在状态变量 $v2$ 使得 $v1$ 数据依赖于状态变量 $v2$)$\wedge$(不存在变量 $v3$ 是状态变量使得 $v3\in con_use(n)$);

复合状态分支,即是间接状态分支又是直接状态分支的分支;

非状态分支,即不属于上述三类分支的分支。

图 2 中分支 $n1$ 属于非状态分支,$n3$ 是直接状态分支,$n8$ 是间接状态分支。

利用进化算法进行测试时,对于非状态分支,控制依赖条件里的局部变量与状态变量没有关系,可以通过改变方法调用的参数或者直接给局部变量赋值来测试,不属于动态搜索的范围;但是对于状态分支的控制依赖条件,特别是 protected 或者 private 状态变量,无法对其进行赋值,只有搜索改变这些状态变量的分支,即状态分支所依赖的分支。如果依赖的分支是状态分支,则继续搜索,直到遇到非状态分支为止。这种分类表现在分支的存储结构上,在动态搜索过程中可以以分支的类属来决定搜索的深度。

2.3 分支之间的依赖关系的定义

2.3.1 *分支间的控制数据依赖关系*(Control Data Dependence)

对于两个分支 $n1$ 和 $n2$,

def2.3.1 v 是一变量,若以下两个条件满足,则称 $n1$ 关于变量 v 控制数据依赖于 $n2$,记为 CDD($n1$,$n2$),

(1)$s1$ 是分支 $n1$ 的控制执行条件,$s2$ 属于分支 $n2$;

(2)当 $n1$、$n2$ 属于不同的方法时,$v\in use(s1)$并且 $v\in def(s2)$(即 $v\in con_use(n1)$并且 $v\in def(n2)$);或者,当 $n1$、

维普资讯 http://www.cqvip.com

$n2$ 在同一方法内时，$s1$ 关于变量 v 数据依赖于 $s2$。

2.3.2 分支间的引用数据依赖关系(Use Data Dependence)

def2.3.2 v 是一变量，若以下两个条件满足，则称 $n1$ 关于变量 v 引用数据依赖于 $n2$，记为 UDD($n1$,$n2$)，

(1)存在 $s1$ 属于分支 $n1$，$s2$ 属于分支 $n2$；

(2)当 $n1$、$n2$ 属于不同的方法时，$v \in use(s1)$ 并且 $v \in def(s2)$(即 $v \in use(n1)$ 并且 $v \in def(n2)$)；或者，当 $n1$、$n2$ 在同一方法内时，$s1$ 关于变量 v 数据依赖于 $s2$。

2.3.3 分支间的传递控制数据依赖关系(Passed Control Data Dependence)

def2.3.3 当满足以下两个条件时，$n1$ 通过 n 传递控制数据依赖于 $n2$，记为 PCDD($n1$,$n2$,n)。

(1)$n1$ 控制依赖于 n；(2)n 引用依赖于 $n2$。

值得注意的是，因为方法内的程序是过程化的，在定义求解方法内的分支间的依赖关系要考虑数据流信息。如果一个分支的语句数据依赖于另外一个分支的语句，则两个分支肯定存在依赖关系；但是求解方法间的依赖关系时，不必考虑数据流和控制流信息。

def2.3.4 方法 m 执行依赖于分支 n，记为 ED(m,n)，当且仅当分支 n 中存在语句调用了方法 m。如果方法 m 执行依赖于 n，则方法 m 里的所有分支都执行依赖于 n。

在图 2 中，分支 $n3$ 和分支 $n1$ 存在控制数据依赖关系 CDD($n3$,$n1$)；$n7$ 和 $n1$、$n10$ 存在引用数据依赖关系 UDD($n7$,$n1$)、UDD($n7$,$n10$)；$n8$ 通过 $n7$ 传递控制数据依赖于 $n1$，记为 PCDD($n8$,$n1$,$n7$)；方法 resize 和分支 $n3$ 存在执行依赖关系 ED(resize(),$n3$)。

3 分支依赖图(BDG)

如引言中所述，一个状态分支的到达不但需要经过一定长度的 public 方法序列的调用，而且必须执行这些 public 方法中对该状态分支的状态变量进行改变的分支。例如在图 2 例子中，把语句 7 作为测试目标时，这就是一个存在状态依赖问题的分支。目标分支的到达依赖于状态变量 stat_qsize 达到一定的状态值。通过静态分析，我们可以找到所有改变这个状态变量的方法分支，比如 foo()、sub()、add()方法里的分支。于是，在进行序列搜索时，也就不必调用所有的 public 方法，大大压缩了动态搜索的空间。另外，由于封装问题，对 resize()内的语句是无法通过直接调用 resize 方法进行覆盖的，只能通过调用 public 方法 adjust()来触发其执行。因此，不同分支之间不但存在一定的控制依赖关系，而且存在一定的数据依赖以及调用关系。为更形象地描述分支间的依赖关系，本文定义并构建了分支依赖图。

3.1 方法的分支依赖图(MBDG)

def 3.1.1 面向对象程序中，方法内的分支依赖图(MBDG)是一个有向图，它可以用一个二元组表示(S,E)。S 是节点集，每个图有个入口节点 entry，$S=\{entry\} \cup N$，其中 N 是方法所覆盖的所有分支。边集 E 是节点间的依赖关系，entry 与所有的节点相连。如果 $s1$,$s2$ 是图中分支节点时，则边〈$s1$,$s2$〉表示 CDD($s1$,$s2$)，或者 UDD($s1$,$s2$)或者 PCDD($s1$,$s2$,n)。图 3(a)是图 2 中的类 Foo 的方法 add 的分支依赖图。

3.2 类分支依赖图(CBDG)

def3.2.1 面向对象程序中，类的分支依赖图(CBDG)是一个有向图，它可以用一个二元组表示(S,E)，S 是节点集，每个图有个入口节点 entry，$S=\{entry\} \cup M \cup N$，$M$ 是类内的方法的集合，N 是类内覆盖的分支的集合。边集 E 是节点间的依赖关系，entry 与所有的方法节点的 entry 入口点相连。当 $s1$,$s2$ 是两个分支节点时，则边〈$s1$,$s2$〉表示 CDD($s1$,$s2$)，或者 UDD($s1$,$s2$)或者 PCDD($s1$,$s2$,n)；当 $s1$ 是方法，$s2$ 是此方法外的分支时，则边〈$s1$,$s2$〉表示 ED($s1$,$s2$)。图 3(b)是图 2 中 Foo 类的分支依赖图。

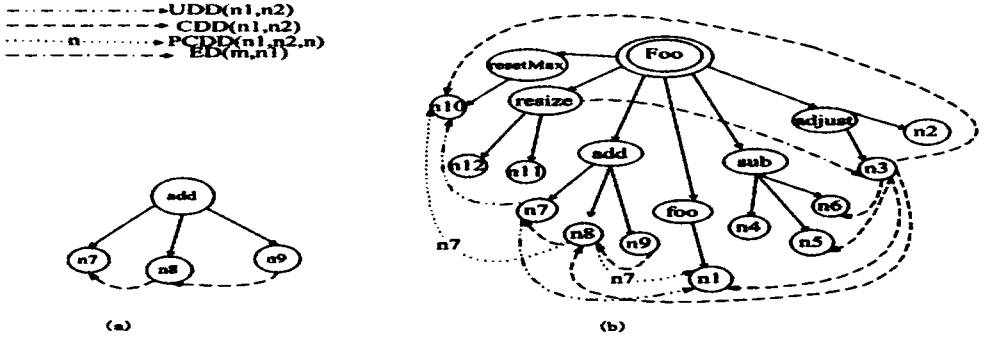

图 3 例子图 2 中方法 add 的分支依赖图和类 Foo 的分支依赖图

3.3 分支依赖图的构建算法

3.3.1 程序静态信息的获取

静态分析的对象是源程序，初步的静态分析可以从源程序中提取与控制流、数据流、接口和表达式等相关的信息，这些信息可以通过求解源代码的程序依赖图[9]得到，这些分析已经是十分成熟的技术[10]。程序依赖图中包含的是语句级的信息，定义的是语句间的控制依赖和数据依赖关系，针对不同的应用，比如错误检测、程序切片或者程序测试，需要进一步地提取和抽象静态信息。本文静态分析需要更抽象、更简洁并能反映分支依赖关系的静态信息，也就是分支依赖图。在构建分支依赖图时，首先对源程序的抽象语法树进行深度优先遍历，提取基本的静态信息，然后对提取的静态信息进行再加工，把语句间的依赖关系转化为分支间的依赖关系。

通过扫描分析 Java 程序的抽象语法树，抽取静态信息，得到方法内的数据依赖图 DDG 以及类的状态变量集 STATE(C)、类分支集合 N(C)、类的方法集合 M(C)、每个方法的分支集合 N(M)以及每个分支 n 的 def(n)、use(n)、con_use(n)集合；通过调用链分析记录方法调用点，这样对 private 或者 protected 方法内的分支进行测试时，只需搜索它们的调用触发点，以这些触发点作为测试目标。这些静态信息保存到了 xml 文件，在构建分支依赖图的时候，以这些存到 XML 文件的静态信息作为输入，这样就减少了对源文件的干预，尽

维普资讯 http://www.cqvip.com

可能减少错误的出现。

3.3.2 方法内的分支依赖图的构造算法

在本文第2部分已经提到方法内的分支依赖关系是要考虑数据流，因此在求解方法内的分支依赖图的时候，我们要以包含数据依赖信息的数据依赖图DDG和包含控制流信息的分支节点子信息N(M)为输入，这些输入在第一遍扫描抽象语法树已经得到，具体算法如算法3.3.2所示。设方法内的分支数为m，语句数为n，则本算法的时间复杂度为$O(m+n+m*m)$，在最坏的情况下，当m接近于n时，其复杂度为$O(n^2)$级别，而一般情况下，m远小于n时，时间复杂度为线性级别$O(n)$。

算法3.3.2

输入：M的数据依赖图DDG以及方法内的分支节点信息N(M)

输出：方法M内的分支依赖图MBDG

步骤：

初始化MBDG的边集E={}、节点集S={}；建立方法入口定点entry，S=S∪{entry}；

```
while 分支 n ∈N(M){
    S=S∪{n};
    生成入口顶点到分支的边〈entry,n〉,E=E∪{〈entry,n〉};
    初始化标识 n 为非状态分支;
}end while
while(扫描 DDG 的每一条数据依赖边〈si,sj〉){
  if((si∈n1)∧(sj∈n2)∧(n1 不等于 n2)
     生成一条引用数据依赖边 UDD〈n1,n2〉,E=E∪{UDD〈n1,
     n2〉};
  if((si 是 n1 的执行控制条件)∧(sj∈n2))
       生成一条控制数据依赖边 CDD〈n1,n2〉,E=E∪{CDD〈n1,
       n2〉};
       if(si 的 use 集合里面含有状态变量){
            if(si 的 use 集合里面含有非状态变量)标识 n1 是复杂
            状态分支;
            else 标识 n1 是直接状态分支;
       }end if
       else if(sj 的 use 集合里有状态变量) 标识 n1 是间接状态分
       支;
}end while
while 分支 n∈N(M){
    if(存在一条控制数据依赖边 CDD〈n1,n2〉){
       while 分支 n ∈N(M){
          if(存在一条引用数据依赖边 CDD〈n2,n3〉)
             生成一条传递控制依赖边 PCDD〈n1,n3,n2〉,E=E∪
             { PCDD〈n1,n3,n2〉};
          }end while
    }end while
```

算法3.3.2结束。

3.3.3 类的分支依赖图算法

求解类的分支依赖关系时，对于类的方法内的分支依赖关系，可以直接复制过来。对于不同方法内的分支依赖关系可以本文第2部分的分支依赖关系的定义来求解，算法如算法3.3.3所示。本算法以3.3.2节得到的类的每个方法的方法依赖图作为输入，扫描每个方法以及每个方法内的每个分支。设类内含有n个方法，每个方法内平均含有m个分支，算法的时间复杂度为$O(n^2*m^2)$。通常情况下，n远小于m，因此算法的复杂度约为$O(m^2)$级别。

算法3.3.3

输入：类C内所有方法的分支依赖图MBDGs

输出：类C的分支依赖图CBDG

步骤：

初始化CBDG的边集E={}、节点集S={}；

建立类的入口顶点entry；S=S∪{entry}；

```
while( m∈M(C)){ 把方法 m 的分支依赖图拷贝到 CBDG; }
end while
while(( m1∈M(C)){
    while (n1 是 m1 的 MBDG 的节点集中的分支节点){
        while(( m2∈M(C)){
            while(n2 是 m2 的 MBDG 的节点集中的分支节
            点){
            if(存在变量 v∈con_use(n1)∧v∈def(n2)){
               增加一条 n1 到 n2 的控制数据依赖边 CDD
               (n1,n2),E=E∪{ CDD(n1,n2)};
               if(存在 UDD(n2,n3)边){
                  增加一条 n1 通过 n2 到 n3 的传递控制数
                  据依赖边 PCDD(n1,n3,n2);
                  E=E∪{ PCDD(n1,n3,n2)};
               }end if
            }end if
            if( 存在变量 v∈use(n1)∧v∈def(n2)){
               增加一条 n1 到 n2 的引用数据依赖边 UDD
               (n1,n2);
               if(存在 CDD(n3,n1)边){
                  增加一条 n3 通过 n1 到 n2 的传递控制数
                  据依赖边 PCDD(n3,n2,n1);
                  E=E∪{ PCDD(n3,n2,n1)};
               }end if
            }end if
        if(n2 是方法 m1 的调用触发点)
          生成一条从 m1 到 n2 的执行依赖边 e=〈m1,n2〉,
          E=E∪{e};
        }end while
    }end while
}end while
```

算法3.3.3结束。

4 类的继承和方法的重载

继承是面向对象的一个重要的属性。子类一方面要继承父类的状态变量和方法，另一方面又有自己的状态变量和方法。这在构建分支依赖图的时候，是不能忽略的问题。在分析子类前，首先分析父类，保存父类的分支依赖图，然后分析子类时把父类的分支依赖信息复制到子类，并增加子类本身的状态变量和方法信息。如果子类的方法中有分支修改父类的成员变量、引用到父类的成员变量或者用到父类定义的方法时，则根据依赖的定义在子类和父类的分支之间或者成员变量和分支间建立依赖关系。

对于重载方法，子类重载了父类的方法时，删除父类的对应的方法分支依赖图信息以及和该方法有关的依赖信息，添加子类的新定义的方法分支依赖信息，并和父类以及子类的其他分支建立依赖关系。

5 类间交互

类间交互的主要语法现象就是一个类的状态变量或者局部变量都可以是其他类的实例化对象。当状态变量是一个类的对象时，在求解含有类的交互的分支依赖图时，首先把实例化的对象看成普通的变量进行分析，认为所有对该对象的成员变量或者成员函数的操作都可能改变该对象的状态。然后再对分支依赖图进行剪枝，把分支依赖图简化，得到类的分支依赖图。

结束语 进化测试是一种有效的测试数据自动生成技术，其核心思想之一为动态搜索与静态模型指导的相互结合。本文主要针对其静态模型抽取部分，设计并实现了方法分支和类分支依赖图的构建算法。通过开源工具ANTLR生成的Java词法和语法分析器得到Java源程序的抽象语法树，在抽象语法树上利用本文算法抽取静态模型，得到动态搜索所需空间。至于在模型之上的动态搜索过程，在本文作者相关文章[5~7]中做了详细介绍，本文不再赘述。

相对于由语句组成的程序依赖图或者系统依赖图，本文所实现的由分支组成的分支依赖图能够以指数数量级压缩搜索空间，同时能够对private/protected方法进行拉链，暴露其相应的公共接口，降低了动态搜索的盲目性。在实际应用中，

（下转第285页）

维普资讯 http://www.cqvip.com

第一步，RCCS间的语义映射。设平均有$\frac{R}{a}$个节点在另一棵RCCS上有相关节点，那么对于每个相关节点还要检测其子树，设节点子树节点个数平均为$\frac{R}{b}$个，对于两个节点的子树上的节点都要判断其是否相关，所以对于相关节点的检测时间(由算法的递归关系知最坏遍历子树上所有节点)得：$\frac{R}{a}\cdot\left(\frac{R}{b}\cdot\frac{R}{b}\right)\cdot h^2$；而对于非相关节点只需要遍历目标RCCS上所有节点，查字典判断其是否同义即可，因此对于所有源RCCS节点的检测时间为$\frac{R}{a}\cdot\left(\frac{R}{b}\cdot\frac{R}{b}\right)\cdot h^2+R(1-\frac{1}{a})\cdot R\cdot h^2$。

第二步，将req转换为新req。对每个req查找映射表，映射表最大长度为$\frac{R}{a}$(每个相关节点均可语义映射)，所以时间为：$\frac{R}{c}\cdot\frac{R}{a}$。

第三步，对新req和msu进行匹配，即分析算法①的时间复杂度。对每个req节点(遍历msu节点)所用时间为：$\frac{R}{c}\cdot 2R$；得$T_1(R)=\frac{R}{c}\cdot\frac{R}{c}\cdot 2R=O(R^3)$。

设有n组req和msu进行匹配：$T_5(R,h,n)=\frac{R}{a}\cdot\left(\frac{R}{b}\cdot\frac{R}{b}\right)\cdot h^2+R\left(1-\frac{1}{a}\right)\cdot R\cdot h^2+n\cdot\left(\frac{R}{c}\cdot\frac{R}{a}+\frac{R}{c}\cdot\frac{R}{c}\cdot 2R\right)=O(R^3\cdot h^2+n\cdot R^3)$

一般来说，字典数据是非常大的，h比R的数量级要高。如果对这种互操作的需求很高，对一个req可以找到若干msu需要与之进行匹配，那么n的增长速度也会非常快。将R看作常数，得：

$T_4(n,h)=O(n\cdot h)$；$T_5(n,h)=O(\max(h,n))$

显然，算法⑤的时间效率要高得多。

5 讨论

本文引入ISO16100国际标准对软件描述的建模方法，研究了基于这种建模方法进行需求匹配的机制。按照标准建模方法和需求分析，分别讨论了简单匹配(req和msu建模于同一RCCS)和语义匹配(req和msu建模于不同RCCS)两种情况。

简单匹配参考同一模板，可以基于ID进行匹配。

对于语义匹配的情况，最简单的就是算法②，它无需参考模板，仅需要查字典，是脱离于标准建模方法也可以进行匹配的一个特例。另外，本文还提供了两种需要参考字典中同义词项的匹配机制：

- 一种是基于能力描述间的节点的功能比较，这种匹配机制基于req的节点展开，在对req和msu的节点匹配时，分别参考不同的模板(RCCS)。
- 另一种是将req转换为新(同义)的能力描述，然后将其与msu进行简单匹配，以达到符合要求的语义匹配。

简单匹配和语义匹配都支持单元匹配级别，对每个req节点都能找到精确匹配、弱匹配和无匹配。其中，精确匹配扩展一般的匹配于功能分解，它不仅可以找出对该需求功能的实现，而且可以找出实现了该功能的更高一层(即集成了该功能)的功能模块；弱匹配可以提示将req此节点进行需求的功能分解(依RCCS)：先找出匹配部分功能的软件，然后开发剩余的功能模块，以提高集成效率。

算法⑤的第2)步在无相关映射表时新建映射表，但若每次注册新RCCS时均建立与其相关的映射表，且在没有映射存在(即映射表为空)时数据库中相关项的uri置空，这样算法⑤就可以结束于映射表uri为空的情况。但是这会面临新的挑战：在RCCS不断增多的情况下，是否需要将新注册的RCCS与现有所有RCCS进行映射？还是可以挑选可能映射的RCCS进行映射，而对于彼此根本不可能存在同功能节点的RCCS直接置空？

虽然需要考虑对遗留系统的兼容，但标准最终的目标是统一，比如对于同节点的功能定义、名称定义的广泛统一。这就要求我们在找出不同的模板(RCCS)后对模板进行规格化和统一性的调整，而将遗留系统融入到新系统，或将不同的系统定义统一，将所有注册的msu按新系统进行相应调整和重新注册(自动化)，这都是今后需要考虑的问题。

参考文献

1 ISO 16100：Industrial Automation Systems and Integration——Manufacturing Software Capability Profiling for Interoperability
Part 1：Framework
Part 2：Profiling Methodology
Part 3：，Interface protocol and Template
Part 4：Conformance test methods，criteria and reports

2 于玮烨，王茜. 基于能力模型的服务匹配机制的研究与实现. 2005年数据库年会，内蒙古自治区，2005

(上接第252页)

这种分支依赖图不仅能够用于进化测试，而且能应用于传统的基于分支覆盖的软件测试以及程序调试和程序理解。

参考文献

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

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

3 Jones B，Sthamer H H，Eyres D. Automatic structural testing using genetic algorithms. The Software Engineering Journal，1996，11：299～306

4 Michael C，McGraw G，Schatz M. Generating software test data by evolution. IEEE Transactions on Software Engineering，2001，27(12)：1085～1110

5 Liu X，Liu H，Wang B. Unified fitness function calculation rule for flag conditions to improve evolutionary testing. The 20th IEEE/ACM International Conference on Automated Software Engineering. Long Beach，California，USA，Nov. 2005

6 Liu X，Liu H，Wang B. Unifying the fitness function calculation for flag conditions. The 6th Metaheuristics International Conference(MIC'05). Vienna，Austria，Aug. 2005

7 Liu X，Wang B，Liu H. Evolutionary Testing in the context of Object-Oriented Programs. In：Proceeding of the 8th Intl. conf. for Young Computer Scientists (ICYCS' 05). Beijing，China，Sept. 2005

8 Rountev A，Kagan S，Gibas M. Static and Dynamic Analysis of call chains in Java. In：Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis，Jul 2004

9 Ottenstein K，Ottenstein L. The program dependence graph in a software development environment. In：Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments，1984. 177～184

10 Zhao J. Applying program dependence analysis to Java software. In：Proc. Workshop on Software Engineering and Database Systems，Dec. 1998. 162～169