面向对象的PROLOG程序测试工具的研究与实现*

王 雷 侯广德

(哈尔滨工程大学计算机与信息科学系,哈尔滨 150001)

摘要 本文首先对Prolog程序中的错误进行了系统的分类,并对各类错误提出了相应的测试算法.接着介绍了一个面向对象的Prolog程序测试工具的设计与实现.

关键词 测试工具,Prolog,面向对象.

随着归结原理在1965年的提出,国内外计算机界对逻辑程序设计和Prolog语言的研究产生了浓厚的兴趣.Prolog语言有以下几个特点:Prolog既是一种程序设计语言,又是一个逻辑系统[1],由于允许使用一阶逻辑元语言的形式描述子句,Prolog表示能力不弱于一阶谓演算系统.Prolog语言的算法和逻辑统一,数据和程序结构统一,执行控制是由合一和回溯自动实现.Prolog语法简明扼要,它只有三类基本语句——事实,规则及询问.

Prolog语言与传统的过程式语言明显的区别和本身的特点决定了Prolog执行的不确定性,因此更需要测试工具保证程序的正确性.但目前国内外还没有一个完善的Prolog程序的测试系统.本文针对Prolog语言的特点,对于程序的不终止性、求解正确性、规则库的检测方法与技术进行了研究,并用C++语言采用面向对象的方法实现了一个测试工具.

1 Prolog程序的错误分类

我们首先根据程序的正确性,将Prolog程序的错误分为两大类:不终止与求解错误.下面介绍几个概念.

定义 1.1. 逻辑程序P的实际含义M(P),是由P可推出的单位基目标的集合.程序P的预期含义M也是一组基目标.

定义 1.2. 终止域D是指基于程序应当成功的在它之上终止的域.

从定义中可看出程序的实际含义M(P)是终止域D的一个子集,否则程序将不终止.

如果M(P)包含在预期含义M中,则对于M,程序P是正确的,如果预期含义M包含于M(P)中,则对于M,程序P是完备的.

* 本文1994-04-01收到,1994-08-13定稿

作者王雷,1969年生,博士生,主要研究领域为软件工程,人工智能.侯广德,1936年生,副教授,主要研究领域为软件工程方法,工具与环境.

本文通讯联系人:侯广德,哈尔滨150001,哈尔滨工程大学计算机与信息科学系

给定一个预期含义和一个预期域,一个纯Prolog程序可能出现两类错误:不终止,求解错误.不终止可以分为4种情况:机制限制,模式不恰当,特殊情况,初值不恰当.求解错误可以分为2种情况:回送某个错误解,无法回送某个真解.

通过对Prolog错误分类,可以将错误作为类,使面向对象的方法与测试工具结合起来.下面还要结合例子,详细地分析产生错误的原因,并给出测试方法.

2　测试技术与方法

上面我们对Prolog程序的错误做了一个概括的分类.这里根据产生错误的不同原因,结合例子给出每类错误的测试方法.

2.1　不终止类错误

不终止类错误主要是指程序中存在死循环.根据我们的经验和所见到的资料可将不终止类错误分为以下4种情况.

(1)机制限制类.

出于求解速度的原因,Prolog采用了深度优先、顺序合一和自动回溯的搜索策略,这引起了不完备性.一旦程序员忽略了这种搜索策略的限制,他们所编写的一些程序中的逻辑功能将无法实现,并有死循环产生.我们称这类错误为机制限制类.下面是一些例子:

a.传递关系:

例1:r1:ab(a,b).
　　r2:ab(b,c).
　　r3:ab(c,d).
　　r4:ab(X,Y):—ab(X,Z),ab(Z,Y).

在此例中,求解目标为ab(X,Y)时,在搜索树中存在一个死循环:ab(d,_)匹配ab(d,_),其中_表示变量没有被约束.这就是由搜索总是从第一个规则先开始所造成的.

b.对称关系:

例2:r1:brother("Harpo","Grou").
　　r2:brother(X,Y):—brother(Y,X).

例2在求解目标为brother(X,Y)时,存在一个死循环:brother(X,Y)调用brother(Y,X),其中X,Y没有约束.这主要是由于Prolog要自动回溯求所有可能解所造成的.

c.等价关系:

例3:r1:ab(a).
　　r2:ab(X):—ba(X).
　　r3:ba(X):—ab(X).

例3中存在一个ab(X)调用ba(X),ba(X)再调用ab(X)的死循环.

(2)模式不当类.

我们首先给出Prolog中"模式目标"的定义.

定义2.1.一个模式目标是一个谓词符号加上对各参量是约束还是自由的情况的指示而构成.一个谓词带有的几个参量约束与否就是谓词的模式.

Prolog的谓词调用方式,可按其参量的约束情况分类.由于逻辑程序的可逆性,Prolog的输入、输出方式很灵活.在Prolog中有个假定,一个谓词将参量赋值或任其自由后,所对

应的某个目标执行后如果为真，自由参量必然约束一个确定的值. 由于这些原因，在程序中容易出现非法的目标模式. 例如：

例 4：r1：append([],Ys,Ys).

r2：append([X|Xs],Ys,[X|Zs]):－append(Xs,Ys,Zs).

这是一个拼接两个表的程序. 如果 append 前面的两个参量是约束的，那么第 3 个参量将输出前面的两个参量的拼接，即 append([a],[b,c],X)，则 X＝[a,b,c]. 若 append 第 3 个参量是约束的，而前两个是自由的，即 append(X,Y,[a,b,c])，将会得到下面的结果.

X＝[],Y＝[a,b,c];

X＝[a],Y＝[b,c];

X＝[a,b],Y＝[c];

X＝[a,b,c],Y＝[];

这正说明了 Prolog 程序的可逆性. 但 append 不是对所有模式都正确，当第 1 和第 3 个参量为自由，第 2 个为约束时，append 将产生一个死循环.

(3)特殊类错误

这类由递归调用引起的死循环比较隐蔽. 多数是由于判断条件考虑不周全，不满足某些特殊情况而引起的，我们称之为特殊类错误.

例 5：r1：isort([X|Xs],Ys]):－isort(Xs,Zs),insrt(X,Zs,Ys).

r2：isort([],[]).

r3：insert(X,[Y|Ys],[X,Y|Ys]):－X＜Y.

r4：insert(X,[Y|Ys],[Y|Zs]):－X≤Y,insert(Y,[X|Ys],Zs).

r5：insert(X,[],[X]).

这实际是一个插入排序的程序. 但是程序没有考虑两元素相等时的处理，因此当目标为 isort([2,2],Xs)时，在 r4 中存在一个死循环，详见图 1.

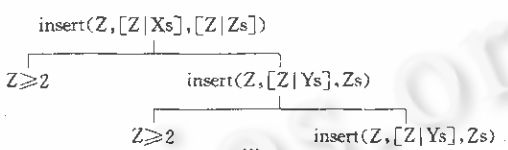

图1 insert的执行过程

(4)初值不当类

若在程序中没有设置对变量初值的检查，那么不恰当的赋值将引起死循环. 我们称之为初值不当类错误.

例 6：r1：fact(1,1)

r2：fact(N,R):－N_1＝N－1,fact(N_1,R_1),R＝R＊N.

这是一个求阶乘的程序，当求解目标为 fact(0,X)时，产生死循环. 调用过程如下：

fact(0,X)

fact(－1,X)

fact(－2,X)

...

跟踪上面的例子我们将发现它们的一些共同点，在 Turbo Prolog 中自由变量用“_”代替. 例1中有 ab(d,_)调用 ab(d,_)；例2，brother(_,_)调用 brother(_,_)；例5中 insert(2,

[2|_],_)调用insert(2,[2|_],_),其他例子中也有类似情况.Prolog程序的执行是由一个目标调用子目标的递归调用过程.我们通过上面的分析可以看出,在这一系列目标$G_1,G_2,\cdots,G_n$中,如果存在死循环,必有一个目标G调用了自身,且G的约束变量不变或远离界值.这样我们通过对目标比较便可以发现死循环.

但是如果每个目标都和其他目标比较,这个数目很大.对于一个n阶的路径需要比较的次数是$(n^2-n)/2$.另外Prolog激活规则方式很多,存在路径数目很大,因此这种算法的时间花费太大.

为了解决上述问题,本系统采用了一种目标比较和递归深度相结合的算法,算法基于这样的思想:死循环一定在最长的路径上.首先设一个最大递归深度D和一个较小的递归深度d,当递归深度超过d时,保留最长的路径,进行比较.如果有死循环则停机,否则将d扩大,继续运行,当递归深度超过D时,停止并保留最长路径供用户判断.这个算法每次只比较一条路径,大大缩小了系统的开销.

2.2　求解错误类

关于求解的正确性问题有两种错误情况:一是返回一个错误解;二是正确解被遗漏.我们要回顾一下前面给出的定义,M(P)为Prolog程序P的实际含义,M为程序执行的预期结果,下面分析产生错误的原因及测试方法.

(1)返回错误解

返回一个错误解的原因:仅当程序有一个假子句时,它才可能送回一个错误解.当子句C包含一个实例,其体在M中为真而头在M中为假.这样的实例称为C的反例.包含反例的子句就是出错的原因.

测试算法:首先程序员给出一组测试用例,在测试覆盖率的指导下,发现错误解.然后基于一个错误解的证明树,求得一个子句的反例;按后序遍历证明树,通过询问检查证明树中每个节点是否为真,如果发现一个假节点,则以假节点为头,以假节点的子节点的连接为体的子句就是程序中的一个反例.

(2)正确解被遗漏

为了弄清正确解被遗漏的原因,我们给出覆盖这个概念的定义.

定义2.1.　如果一个子句有一个实例,它的头是目标A的一个实例,而体在预期含义M中,我们就说这个子句关于M覆盖目标A.

可以证明,如果程序P关于预期含义M有一个遗漏解,那么M中存在一个不为P的任何子句所覆盖的目标A.

诊断一个遗漏更加重了决断者(程序员)的负担,他不但要知道目标是否有解,而且在解存在时,还必须提供它.利用这样的判断,寻找一个未被覆盖的目标的算法如下:

通过一组测试用例寻找一个遗漏解,算法从这个初始的遗漏解开始.对于每一个和初始遗漏解匹配的子句,通过程序员检查该子句体是否有一个实例在M中.如果不存在这样的子句,该目标就没有被覆盖,算法结束;否则,算法就在子句体中找出一个失败的目标.至少应当有一个失败的目标,不然的话程序就解出了子句体,从而也就解出了目标.算法被递归地应用于这一目标.

2.3　测试用例生成

测试用例生成是测试工具的一个重要组成部分. 我们利用 Prolog 程序的不确定性,产生测试用例,以实现自动测试. 产生——测试方法就是对 Prolog 不确定性的一个应用. 在这个方法中,我们用一个程序产生测试用例,一个程序去测试. Prolog 程序编写的一个简单的产生——测试程序一般具有两个目标的连接,其中一个作为产生器,而另一个作为测试器,如下面子句所示:

```
find(x):-generate(x),test(x).
```

这样可以通过自动回溯机制,对所有 generate 所产生的用例进行测试.

系统采用自动记录测试用例方法. 减少了程序员的重复输入,又便于分析测试结果.

有时输入很多用例,但却不一定能查出许多错误. 这是因为可能大部分测试用例都是在一条路径上的重复. 因此我们提供测试覆盖检测来指导用例生成.

测试覆盖检测就是给出以前测试用例覆盖的谓词及其成功还是失败,使程序员了解哪个谓词没有被测试到,在以后的测试用例设计中,要考虑对这些谓词的测试.

3 系统的设计与实现

本 Prolog 程序测试系统总体结构是采用面向对象的方法设计的,面向对象方法的优点来自其方法的特点. 其主要特点是封装性、对象之间消息传递以及继承性. 前两个特点导致类(模块)之间的联系大大减弱,使面向对象系统中的类较之传统方法中的模块具有更高的独立性;最后一个特点简化了代码的设计与冗余. 本系统定义每一种错误为一对象,将具有相同性质的对象划分为错误类. 具体的一些测试算法和附加功能是采用 C++ 与 Prolog 混合编程来实现.

系统主要功能如下:对 Prolog 程序中四类不终止问题进行测试;对 Prolog 程序中二类求解错误问题进行测试;提供了编辑功能,用户可在系统内编辑被测程序;提供了跟踪功能;提供了交互和批处理两种测试用例输入方式;自动对测试覆盖进行分析.

本系统目前的处理对象(被测程序)是 Turbo Prolog 语言书写的无语法错误的程序. 使用本系统进行测试时,不用编写前驱和后继模块,对被测程序中包括的所有目标均可进行单独测试,而不需要程序员做额外的编程工作.

系统的用户接口采用了菜单与窗口方式,上面的功能的选择都可以在菜单中进行. 用户在系统的指导下,输入被测程序、测试用例,系统根据测试算法生成测试报告,并以文件的方式存储便于以后存档. 系统还通过测试覆盖率指导用户生成测试用例.

系统的总体结构如图2所示. 系统分为类库和总控类两部分. 类库包括不终止类和求解错误类两大类. 用户自己需要编写测试工具时,可以从类库中直接提出基类. 这正是"软件部件化"思想的体现.

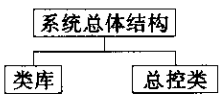

图2 系统总体结构

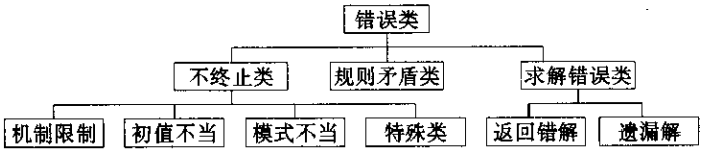

图3 类结构

系统是采用面向对象方法设计的.在系统中看到的只是通信着的对象.类的设计采用了一种定货单的方式.但是最终对象之间的关系和系统的功能必须被实现.这些都是通过总控类来实现的.

下面简略地介绍各个类的实现以及总控类的结构.

(1)类库

系统将所有的错误类放在类库之中.如果用户想自己编写测试工具,可以将类库的头文件嵌入他所编写的程序中便可以使用这些错误类的测试功能.这使系统具有良好的可重用性.类库中的类结构如图3所示.

系统以错误类作为根类,为了便于扩充错误类定义为一个抽象类.抽象类是供别的类作为基类使用的类.在C++中,抽象类应含有纯虚拟方法.不能够为一个抽象类类型生成任何实例对象.然而,指向抽象类对象的指针和引用是合法的.

抽象类为多态的应用提供了方便.错误类是这样定义的:

```
class error
{
 public:
   vritual void report();  //报告生成
   vritual void cover();  //测试覆盖
   char * profile(char * tfile);  //文件预处理
 };
```

说明:其中virtual关键字指明了虚函数,虚函数依赖于调用它们的对象的原始类,它们须有一个隐含的对象,并且必须是一个类的初始成员.

在不终止类中,除了包含两个虚函数外,还包含有一个由4个子类共享的测试算法.不终类定义如下:

```
class loop:public error
{
 public:
   vritual void report();
   vritual void cover();
   void test(char *tfile);  //测试算法
 };
```

不终止类的子类包含两个虚函数的具体实现,下面给出其中一个子类的结构:

```
class sloop:public loop
{
 public:
   sloop(char * n);  //构造函数
   ~sloop();  //析构函数
   void report();
   void cover();
 };
```

求解错误类及其子类结构和不终止类相似,这里就不重复给出了.

(2)系统的总控类

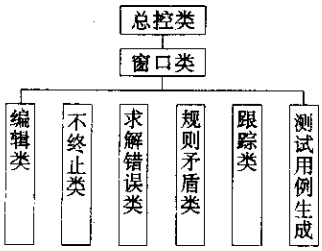

图4 系统模型

系统的设计是在类一级进行的，而运行是在实例对象上进行的. 系统的一次运行就是对类的一次例化. 从功能的角度上看，就是对 Prolog 程序的错误进行一次测试. 系统通过总控类来激活，形成一个互相发消息的运动着对象集.

系统的对象关系模型如图4.

说明：总控类首先激活窗口类. 窗口类是用户和系统之间通信的接口. 通过窗口类用户可以选择所测试的错误类. 之后系统分析用户的选择并形成消息发送给相应的对象，因此总控类还是各对象之间通过发消息进行交流的中心.

总控类的类结构如下：

```
class control
{
 public :
    pdrive();  //激活系统
    perror(error * p);  //处理错误类消息
    pedit();  //处理编辑类消息
    ptrace();  //处理跟踪类消息
    pcase();  //处理用例生成类消息
 };
```

对不同的错误类应有不同的测试报告和测试算法，而在总控类中，除了构造对象以外，其余部分都统一了. 这是由于有了指向错误类的指针 P. 当用户选择了不同的错误类时，P 便指向程序中动态构造的错误对象. 这时程序便通过指针 P 找到了应该执行的那个对象的操作. 这不仅使程序的源代码简明清楚，而且当类增加时，源代码也不用做较大的变动. 这正体现了系统易于维护、易于扩充的性质.

(3)元解释程序

在众多的软件测试方法中，动态测试具有一定优越性，是较为成功的一种测试方法. 但它也存在一些缺点，首先动态测试使程序运行要花费系统资源，代价比较昂贵. 另外在程序动态运行的瞬间去发现错误是很困难的. 这需要控制被测程序的运行. 在运行过程中，不仅要知道当前的目标、参量内容，而且还要根据一些异常信息对错误定位、判断.

为了解决上述问题，我们采用了元解释程序的方法，将测试部分分成3个层次，它们的关系如图5. 元解释程序就是把其它程序作为数据来处理的程序，它可以分析、变换以及模拟其它程序. 由于 Prolog 中程序与数据的等价性：两者都是 Prolog 的项，所以便于编制元解释程序. 便于编写元解释程序是 Prolog 程序设计语言具备的一个很大的优点，它使建立一个一体化的程序设计环境成为可能，并提供了解语言计算进程的途径.

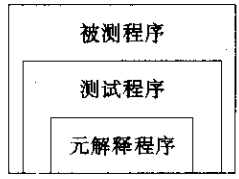

图5 测试部分结构

从图5中，可以看出测试系统数据分3个层次：被测程序、测试程序和元解释程序. 三者用户视图都是 Prolog 程序，但在系统中他们是不一样的. 被测程序在测试程序中，作为输入的

数据.而测试程序又作为元解释程序的输入数据.在测试程序这个中间环节中,需要对被测程序进行比较、匹配等一些操作,但在元解释程序中,存储的被测程序都是些静态项,这之间不能比较,需要引入一个转换机制,将静态项变为一般Prolog项,这样才可能比较.我们采用一个 cmp_str 转换器解决了这个问题.

4　系统的评价

面向对象的 Prolog 测试工具是在286上,采用 Borland C++ 2.0和 Turbo Prolog 混合编程实现的.系统的源代码近200K.系统采用动态的方法对 Prolog 程序中的不终止性和求解错误进行了测试.

在实际测试中,系统快速地测试出被测程序中的不终止错误;在程序员的参与下,测试出被测程序中的求解错误.并对错误进行了精确的定位,取得了良好的效果.

系统主要有以下特点:系统采用的面向对象方法实现.整个系统建立在类库基础上,类比传统的模块有更高的独立性,使系统具有良好的可重用性,是软件部件化思想的集中体现.由于封装、继承等性质减少代码的设计与冗余,使系统易于扩充与维护.系统采用动态的方法测试 Prolog 程序中的错误,这虽然增加了系统的开销,但使测试更加彻底,并且提高了发现错误的概率.系统采用递归深度与目标比较相结合的不终止测试方法.目前无论从时间,还是从内存的角度看,算法都是很优越的.系统采用了谓词覆盖准则进行测试检测,并指导用例生成,提高了发现错误的概率,减少不必要的重复测试.系统采用了产生——测试技术来生成用例,减少了程序员的工作量,并且为以后 Prolog 程序测试用例的自动生成铺平了道路.

参考文献

1　黄明等.一个 Prolog 查错机制——矛盾检测的设计与实现.第四次全国软件工程会议论文集,北京,1991.

RESEARCH AND IMPLEMENTATION OF OBJECT—ORIENTED TESTING TOOL FOR PROLOG PROGRAM

Wang Lei　Hou Guangde

(*Department of Computer and Information Science, Harbin Engineering University, Harbin* 150001)

Abstract　In this paper, the errors in Prolog program aresystematically classed, and the corresponding testing algorithm is presented. Then design and implementation of object—oriented testing tool for Prolog program is introduced.

Key words　Testing tool, Prolog, object—oriented.