基于 UML 状态图的面向对象软件测试用例生成

江 曼 王天青 潘金贵

(南京大学计算机软件新技术国家重点实验室 南京大学计算机科学与技术系 南京 210093)

摘 要 本文提出了一种基于 UML 状态图的测试用例生成方法,生成较少但有效的测试用例,便于实现自动化测试。测试用例从状态图中的转换(转换路径)中产生,一个用例代表了一条转换路径。对深度优先算法进行改进后,给出了从 UML 模型视图的状态图中获得测试用例的算法,该算法从状态图中的初始状态到终止状态进行遍历,可以得到所有的转换路径,根据循环复杂度来得到状态图的基本路径的最大数量,即测试用例的最小数量。

关键词 UML,测试用例,测试用例生成

UML State Diagram-based Test Case Generation of OO Software

JIANG Man WANG Tian-Qing PAN Jin-Gui

(State Key Lab for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093)

Abstract This paper introduces the development of software testing technology, puts forward a method to obtain test cases from UML State Diagram. Test cases will be generated from the transitions (transition path) of the State Diagram. The method above is absolutely based on UML and can generate less but more effective test cases, which can be automated easily. Then we improve the depth-first-search algorithm and bring forward an algorithm to obtain test cases from UML State Diagrams, which travel the State Diagram from the initial state to the ending states and find all the transition paths. And we can get the maximum of basic path based on cycle complexity, namely the minimum of test case.

Keywords UML, Test case, Test case generation

1 背景

软件测试技术经历了从人工向自动化发展并不断成熟的过程。测试自动化能够更好地利用资源,使测试具有一致性、可重复性和复用性。测试自动化研究包括:自动生成测试用例(脚本)、自动运行测试脚本和自动分析测试结果[1]。其中,自动测试用例生成是最重要的环节。

面向对象技术的引入为软件测试的发展带来了新的契机和挑战。面向对象系统包括对象和类,一个对象由一组定义其状态的属性和一组定义其行为的操作组成。封装、继承、多态是面向对象最重要的三个特性。封装隐藏了类的数据结构和实现细节,对于测试需要的内部信息,测试人员往往不得而知;继承允许子类复用父类,引入了重测试问题;多态和动态绑定引入了不可判定问题,很难静态地确定在给定的测试和用例中哪个方法被激活。在面向对象软件测试中,基本可测试单元不再是子程序,而是类或对象。同时必须修改集成测试策略,使用集成类来创建系统与整个开发方法相联系。在对每个类进行了单元测试后还需对类簇进行测试,逐步完成整个系统的测试[2]。

UML 由于其定义良好、易于表达、功能强大,融入了软件工程领域的新思想、新方法和新技术,不仅可以支持面向对象的分析与设计,还应用于从需求分析开始的软件开发的全过程。目前,UML 已被广泛应用到描述系统的静态结构和动态行为中,因此自然成为了测试用例生成的有效来源。

本文提出了从 UML 模型视图的状态图中获取测试用例的方法,该方法从状态图中的初始状态到终止状态进行遍历,可以得到所有的转换路径,并根据循环复杂度来获得状态图基本路径的最大数量,即测试用例的最小数量。

2 基于 UML 的状态图测试用例生成

UML 提供了一系列的视图,用来描述待开发系统的不同方面,状态图就是其中之一。基于状态的测试非常适合于面向对象软件。面向对象系统常常把正确行为的责任分配到几个类、簇、一个子系统和一个系统上,状态机可以为行为建立模型,基于状态的测试也可以用于任何范围(类、簇、构件、子系统、整个系统)。

2.1 类状态自动机

状态图(State Diagram)用来描述一个特定对象的所有可能状态及引起状态转移的事件。UML 状态图基于有限状态自动机,它使用 Harel 状态图元素,通常用来表示一个对象的行为。

所有对象都具有状态(State),一个对象的状态是它所有成员属性的值和成员对象状态的组合。对象的动态性通过状态间的转换来建模。一个转换(Transition)由源状态(Source State)、目标状态(Target State)、事件(Event)、监护条件(Guard)和一系列的动作(Action)组成。

UML 转换分为 5 种类型:高层转换(high-level transitions)、混合转换(compound transitions)、内部转换(internal transitions)、完成转换(completion transitions)和激活转换(enabled transitions)[3]。我们的研究只集中于激活转换,一

江 曼 硕士研究生,研究方向为软件测试、软件工程;王天青 硕士研究生,研究方向为软件工程、软件过程改进等;潘金贵 教授、博士生导师,从事中间件、Agent 技术及多媒体远程教学系统等的研究。

个激活转换被一个事件触发，它的源状态是一个活动状态。当从源状态到目标状态至少存在一条完整的路径的时候，一个激活状态被触发。

一个简单的例子[4]，类 Engine：

```
Class Engine {
int speed;//速度
boolean keyOn;//钥匙是否插入
Engine();//构造函数
~Engine();//析构函数
void Start(int S);//开车
void Stop();//停止
int getSpeed();//获得速度
setSpeed(int S);//设置速度
setKeyOn(boolean in);//设置钥匙插入
}
```

车有两个状态：OFF(停止状态)和 ON(运行状态)。

OFF：speed = 0 and KeyOn = false ON：0≤speed≤110 and KeyOn = true

它的状态转换图如图 1 所示。

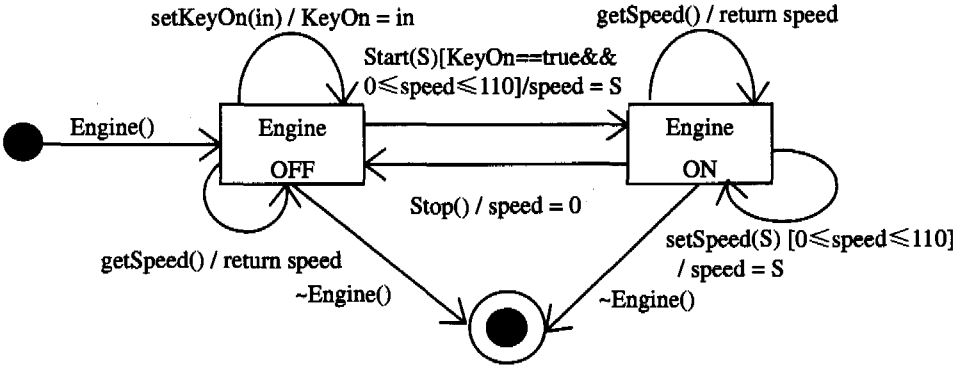

图 1　类 Engine 的状态转换图

状态图可视化地描述了一个有限状态自动机，我们可以从一个状态转换图中得到一个 CSM[4]。

定义 1 CSM(Class State Machine)　类 C 的一个类状态自动机 M =(*V*, *F*, *P*, *S*, *T*)，其中

V 是类 C 实例变量的有限集。

F 是类 C 成员方法的有限集。

P 是类 C 可变(Mutator)成员方法参数的有限集。可变成员方法指的是修改了对象状态的方法。

S 是状态的有限集，$S=\{s|s=(\text{pred})\}$，其中 pred 是 *V* 中实例变量的一个谓词表达式。

T 是转换的有限集，$T=\{t|t=(\text{source, target, fn, guard, action})\}$，其中

source，target∈*S*，是转换开始前和结束后的状态。

fn∈*F*，是当监护条件计算为真时，触发 t 的成员函数。

guard 是 *V* 中实例变量和 *F* 中成员方法的参数的一个谓词表达式。

action 是 *V* 中实例变量和 *F* 中成员方法参数的一系列计算。

对于 Engine 而言，它的类状态自动机的各个部分如下：

S={S0, Sf, OFF, ON}

V={int speed, boolean keyOn}

F={Engine(), ~Engine(), setKeyOn (boolean in), Start(S), Stop(), setSpeed (int S)}

P={setKeyOn:in, Start:S, setSpeed:S}

T={ti|1≤i≤9}

t1={S0, OFF, Engine (), true, {speed=0, keyOn=false}}

t2={ OFF, OFF, getSpeed (), true, {return speed}}

t3={ OFF, OFF, setKeyOn (in), true, {keyOn = in}}

t4={ OFF, ON, Start(S), KeyOn = = true and 0≤speed≤110, {speed = S}}

t5={OFF, Sf, ~Engine(), true, {}}

t6={ON, ON, getSpeed(), true, {return speed}}

t7={ON, ON, setSpeed(S), 0≤speed≤110, {speed = S }}

t8={ON, OFF, Stop(), true, {speed = 0}}

t9={ ON, Sf, ~Engine(), true, {}}

我们可以从图 1 得到如图 2 所示的 Engine 的类状态转换自动机。

2.2　覆盖标准

我们希望实现了状态机模型的软件能够进行正确的状态转换，即在某个状态接受了合法的事件(状态图中定义的事件)，执行正确的动作后能跳转到正确的结果状态，同时要拒绝非法的事件(状态图中没有定义的事件)。但是大量的错误仍然存在，文[5]给出了可能出现的错误。为了能够找出错误是否存在，我们必须从状态图中得出足够的测试用例。状态图描述的转换，是我们测试的目标。我们从状态图中获得的测试用例就是为了检查对于状态图中的每个转换，其对应的实现是否也存在这样的转换，同时两者是否具有一致性(包括一致的源状态、动作、结果状态等)。

因此我们需要一定的覆盖级别来生成测试用例。文[6]给出了 4 个级别：①转换覆盖；②全谓词覆盖；③转换对覆盖；④完整序列覆盖。

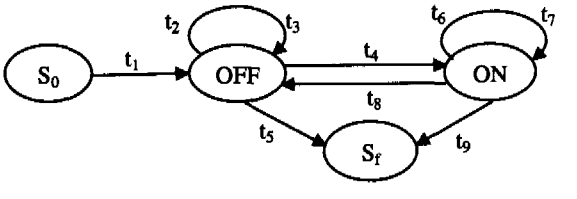

图 2　类 Engine 的状态转换自动机

定义 2　转换覆盖(Transition Coverage)　测试用例集必须覆盖状态图中的每一个转换。

定义 3 全谓词覆盖(Full Transition Coverage) 对于每个转换的监护条件,测试用例集必须覆盖监护条件的每个有影响的分支。所谓有影响的分支指的是该分支(c)的取值影响监护条件(Guard)的取值,即当 c 分别为 true 和 false 的时候,Guard 的取值正好相反。易见,当全谓词覆盖级别达到的时候,转换覆盖级别也已达到。

以上两个测试级别都是对转换单独地进行测试,而在实际测试中,这往往是不够的,还需要以下两种覆盖级别。

定义 4 转换对覆盖(Transition-pair Coverage) 对于状态图中每一对相邻的转换,测试用例集中必须包含覆盖这一转换对的测试用例。

定义 5 完整序列覆盖(Complete Sequence) 必须通过选择需要进入的状态来定义状态图中有意义转换序列。

一个完整的序列是指一系列状态转换,这些状态转换形成了对系统的一个完整的实际使用。在大多数实际的应用中,选择全部完整的可能序列的数量太多,因此需要通过某些手段在不降低测试可信度的情况下,来减少测试用例。

3 用例生成算法实现

可以看到,对于每个用例(Use Case,相当于一个完整的实际使用),能够从状态图中找到一条从初始状态开始到终止状态的状态转换路径。也就是说,一个用例代表了一条转换路径,因此测试只要能够找到状态图中所有的转换路径生成测试用例,就可以达到有效测试。我们对完整序列覆盖进行改进,就得到完整转换路径覆盖。

定义 6 转换路径(Transition Path) TP=($t1$, $t2$,…, tn),其中:

$ti \in T, 1 \leqslant i \leqslant n$;

$t1$ 的源状态为初始状态,tn 的目标状态为终止状态;

ti, $ti+1 \in T, 1 \leqslant i \leqslant n-1$, ti. target = $ti+1$. source;

$ti \in T, 1 < i \leqslant n$, ti. source≠初始状态;

$ti \in T, 1 \leqslant i < n$, ti. target≠终止状态;

定义 7 完整转换路径覆盖(Complete Transition Path Coverage) 令 TPS 为状态图中转换路径集合,TC 为测试用例集,对于任意 $tp \in TPS$,必须至少有一个 $t \in TC$,t 运行时能够覆盖这一路径。

对深度优先算法进行改进,从状态图中的初始状态到终止状态进行遍历,可以得到所有的转换路径(对于回路,我们限制它在转换路径中至多出现一次)。可以根据循环复杂度(cycle complexity)来得到活动图的基本路径的最大数量,这就是测试用例的最小数量。

该算法需要满足:

1)检测出所有的无效状态和无效转换。

2)驱动转换发生,由一个事件触发。

3)找出监护条件,判断前后之间的逻辑关系是否有矛盾冲突。

4)找到所有的转换路径。

算法描述如下:

```
输入:Transition。从 XML 文件中读入转换,并初始化;
输出:TransPath。所有的转换路径;
public class Method;//触发方法
public class Guard;//监护条件
public class Action;//动作
public class Transition{//转换
    StatusPoint source,target;
    Method method;
    Guard guard;
    Action action;
}
public class StatusPoint{//状态点
    int num;//编号
    boolean endPoint = false;//是否为终止状态
    int[] inEdge;//入边
    int[] outEdge;//出边
}
public class TransPath{//找出所有的转换路径
  boolean [] visited;//用于记录是否已经访问
  StatusPoint start;//开始状态
  StatusPoint [] finish;//结束状态集
  Transition [] transition;//转换集
  StatusPoint [] spArray;//状态集
  List Result;//保存所有转换路径
/ * * 构造函数,用来初始化状态图 * /
public TransPath(…){
//step 1  从 XML 文件中读入转换,初始化 transition;
transNum = ……//读入转换数目
   statusPointNum = ……//读入状态数目
   transition = new Transition [transNum];
          …//初始化每一个转换;
//step 2  从转换中得到所有的状态
    spArray = new StatusPoint[statusPointNum];
                //找出所有状态,读入初始状态
for (int k = 0; k<statusPointNumber; k++){
    //新建状态点
    StatusPoint sp = new StaticPoint();
    //赋值给状态点的编号
    sp. num = k;
    spArray [k]=sp;
    for(int i=0;i <transNum;i++){
    //查找 Transition 中的 source
        StatusPoint first = transition [i]. source;
        //将转换的原状态赋值给初始状态
        if(first. num == k)
        outEdge. add( transition [i]. target. num);
    / * 如果等于 i;则把 ArrayList tran 的值赋 e 为 l;如果 i 等于
    StatusPointNumber,则 endPoint 为 true * /
        if(first. num == 0)
        start = first;
          else if(i == StatusPointNum)
              endPoint = true;
              ……
        }
    //step 3  初始化访问状态
    visited = new Boolean[statusPointNum];
     for(int i = 0;i < statusPointNum; i++){
       visited [i] = false;
     }
    //step 4  调用深度优先算法
    DFS( 0, List path);
    ……//检测所有的转换路径,删除所有的无效转换和无效状态;
}
 / * * 深度优先算法 DFS * /
 public void DFS (int v, List path){
 //当前节点访问设为真
 visited [spArray[v]. outEdge[current]] = true;
        if (spArray[v]. endPoint == true){
        Result. add(path);//保存一条转换路径
            return;
 }
 //将状态 v 转换到的下一个状态的编号赋给 w;
        w=spArray [v]. outEdge[current];
            ……
        / * 检测是否满足监护条件的测试数据,并修改 Attribute 属
        性值,return * /
        While (w ! = null){
        If(! Visited[spArray[w]. tran[current]])
          path. add(v);
        //将当前转换添加到路径中
        DFS(w,path);//递归访问
 visit[w. tran[current]]=false;
 //访问结束后清楚访问状态
 path. remove(v);
 //访问结束后将当前转换从路径中删除
 w=GetNextNeighbor(v,w);
    }
  }
}
```

该算法采用面向对象的 Java 语言加以实现,具有可移植性、安全性和稳定性。测试结果也令人十分满意,达到了最初设定的要求。

使用该算法可以得到以下的转换路径:

p1={t1,t5},p2={t1,t2,t5},p3={t1,t3,t5}
p4={t1,t4,t9},p5={t1,t4,t6,t9},p6={t1,t4,t7,t9}
p7={t1,t2,t4,t9},p8={t1,t2,t4,t6,t9},p9={t1,t2,t4,t7,t9}

(下转第 290 页)

i201，在创建时初始化了 item i002 的基本信息，例如标题、分类、所属者、底价等。假如 item i201 是属于达芬奇的画，而 A 与 C 都对画感兴趣，但基于内容的订阅方式可以让 A 只订阅齐白石的画，而 C 只订阅达芬奇的画，所以 US 服务器创建了 i002 后只向 France 服务器发送创建通知。

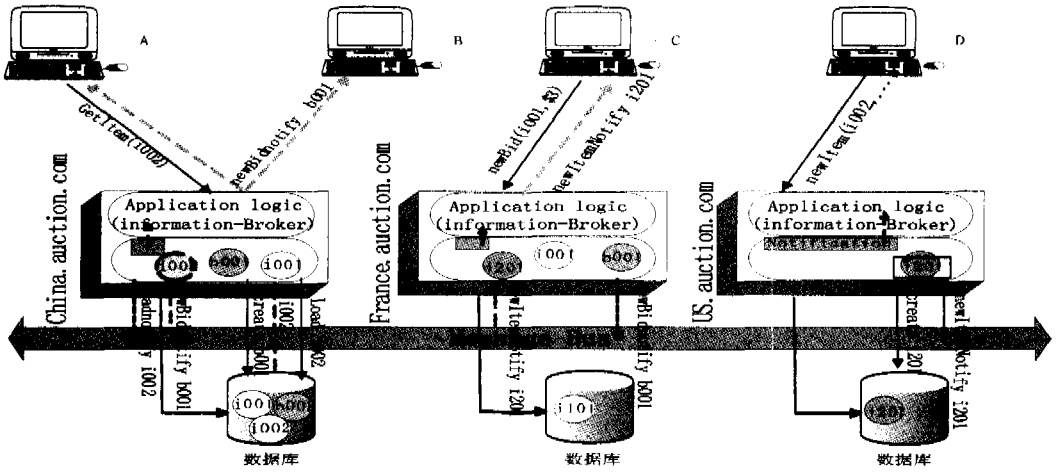

图 4　扩展持久状态服务运行范例

总结　本扩展模型在原持久状态服务模型的基础上加上了缓存管理以及基于内容订阅模式的消息中间件。通过加上缓存管理，可大大加快扩展的持久状态服务访问持久对象的速度；通过加入基于内容订阅模式的消息中间件，可实现分布式信息系统中服务器间的持久对象共享，而且在消息中间件上发布与订阅时使用 XML 来描述持久对象的状态信息可提高系统的灵活性和可扩展性。

参 考 文 献

1　OMG. Persistent State Service Specification. Object Management Group. http://www.omg.org/cgi-bin/doc? ptc/00-12-07, January 2000

2　Liebig C, Cilia M, Betz M, et al. A publish/subscribe CORBA Persistent State Service Prototype. In: Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms, 2000. 231～255

3　Garcia-Banuelos L, Duong P, Nedelec T, et al. A Component-based Infrastructure for Customized Persistent Object Management. In: Database and Expert Systems Applications, Proc. 14th Intl. Workshop on, 2003. 536～541

4　曾一，徐珂. 基于组件架构的对象持久层框架设计. 计算机科学，2005(12)

5　刘振涛，熊璋，欧阳元新. 基于内容的订阅方法及其在 MQSeries 中的实现. 计算机工程，2004，30(16)

(上接第 292 页)

p10={t1,t3,t4,t9}, p11={t1,t3,t4,t6,t9}, p12={t1,t3,t4,t7,t9},
p13={t1,t4,t8,t5}, p14={t1,t4,t6,t8,t5}, p15={t1,t4,t7,t8,t5}
p16={t1,t2,t4,t8,t5}, p17={t1,t2,t4,t6,t8,t5}, p18={t1,t2,t4,t7,t8,t5}
p19={t1,t3,t4,t8,t5}, p16={t1,t3,t4,t6,t8,t5}, p17={t1,t3,t4,t7,t8,t5}

我们可以从每一条转换路径得到一个测试用例，该测试用例用来检查目标程序是否存在这样一条路径。

总结和进一步的工作　本文提出的基于 UML 模型的测试用例的生成方法利用 UML 状态图，为测试用例生成提供信息。该方法的特点是：①完全基于 UML；②生成较少的测试用例；③便于实现自动化。

对基于 UML 生成测试用例的研究仍存在很多不足。目前对基于 UML 生成测试用例的研究，主要是以单个模型图作为研究对象；基于设计描述的形式化测试准则还较少；对 UML 模型图做进一步精化或扩展其语义后，使用常规方法进行测试用例生成的情况较多，直接使用 UML 模型图提供的信息的方法较少；在具体的方法研究中，假设的前提和约束太多，还不能够达到实用的程度。因此，对基于 UML 生成测试用例的深入研究将是我们未来工作的重点。主要有以下几个目标：

第一，我们可以将状态图作为多个交互类集成测试的基础，在状态图上定义测试标准，生成的测试规格说明书必须满足这些准则并在数据库中保存状态转换描述、控制、数据流等信息。

第二，深入研究在生成针对某一层次测试的测试用例时，综合利用待测试系统的各种模型图的测试用例生成方法。

第三，提出一种与面向对象软件开发过程集成的测试过程，设计出能够实现面向对象软件测试活动中测试用例生成、测试执行和结果评价这三个过程的软件自动化测试框架。

参 考 文 献

1　Mosley D J, Posey B A. 软件测试自动化. 邓波，黄丽娟，曹青春，等译. 北京：机械工业出版社，2003

2　Barbey S, Strohmeier A. The problematics of testing object-oriented software. In: SQM'94 Second Conference on Software Quality Management, 1994, 2. 411～426

3　Object Management Group. UML Specification V1.5, 2003. http://www.omg.org/cgi-bin/apps/doc? formal/03-03-01.pdf

4　Gallagher L, Offutt J. Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details: [Technical Report]. CMU/ISE-TR-04-04, 2004

5　Binder R. 面向对象系统的测试. 华庆一，王斌君，陈莉，译. 北京：人民邮电出版社，2001

6　Offutt J, Abdurazik A. Generating Tests from UML Specifications. In: Proceedings of the 2nd International on Unified Modelling Language. Beyond the Standard(UML'99), 1999