基于 UML 的系统测试用例自动生成①

胡俊豪，何 春，宗竹林

(电子科技大学 电子科学技术研究院，成都 610054)

摘 要：提出了一种新颖的基于 UML 图自动生成系统测试用例的方法。此方法适用于所有 UML 建模的系统。用创新性地结合图论知识，对用例图和序列图进行抽象，定义了相应的测试覆盖准则以生成测试用例。通过两级遍历，自动生成系统测试向量，并且达到所提的系统测试覆盖率的要求。能够覆盖所有用例、用例依赖性、以及交互中所有的消息序列。实现最大化的覆盖范围。

关键词：系统测试；UML；用例图；序列图；用例自动生成；图论

Automatic System Testing Test Case Generation Based on UML

HU Jun-Hao, HE Chun, ZONG Zhu-Lin

(School of Electronic Engineering, University of Electronic Science and Technology, Chengdu 610054, China)

Abstract: This paper presents a novel approach of generating system testing automatic test cases from UML design diagrams. This method can be applied to any system modeled by UML. Associating Graph Theory innovatively, we abstract use case and sequence diagram and define coverage criteria to generate test cases in our test case generation scheme. Through two levels traverse, generate test cases automatically and satisfy the coverage criteria stated in this paper. The test cases thus generated are suitable for system testing and to detect operational, use case dependency, interaction and scenario faults. The coverage reaches at its maximum level, covering all use cases, use case dependency, and each sequence of message path of interaction.

Keywords: system testing; UML(unified modeling language); use case diagram; sequence diagram; test case automatically generation; graph theory

近年来，UML (unified modeling language)作为软件建模的标准，并得到了研究者和实践者极大的重视。随着系统的复杂化，UML 在系统测试中得到了极大的关注。

本文提出了基于 UML 模型，一种新颖的自动生成系统测试测试用例的方法。这里使用用例图和序列图作为生成测试用例的信息来源。

创新地结合图论知识对 UML 模型进行抽象，采用 2 级遍历，生成测试用例满足如下两个覆盖原则：

覆盖标准 1：所有用例和用例依赖关系标准： 假设测试集 tSet 和用例图 G1, tSet 必须使每个用例均发生并且使每个代表依赖性的路径执行至少一次。

覆盖标准 2：所有顺序图消息路径序列覆盖标准：假设测试集 tSet 和顺序图 G2，tSet 必须使每个消息序列路径执行至少一次。

此方法生成的测试集能够覆盖操作错误、用例依赖性错误、初始化错误以及交互错误。

1 引言

系统测试的目的就是要确保整个综合系统处于无错误的状态。系统测试通常被认为是所有测试类型中最为错综复杂的类型。这归结于系统测试的是一个完全综合的系统，此系统通常大型、复杂并且基于状态机，通常超出了手动测试的范围[1]。

近年来，UML (unified modeling language)作为软件建模的标准，并得到了研究者和实践者极大的重视。UML 模型在设计测试用例方面的重要性已经得到很好的验证[2-5]。Briand 和 Labiche[2]，描述了 TOTEM

① 收稿时间:2010-05-22;收到修改稿时间:2010-06-30

(Testing Object orienTed systEms with the unified Modeling language) 系统测试方法。系统测试需求得自于早期的 UML 分析模型，比如用例图和序列图。他们通过在活动图中附加人为干涉来获得用例之间的顺序关系。基于这些顺序依赖关系，生成有效的用例序列来生成测试用例。他们这种方法实质上是一个半自动的场景覆盖方法，因为需要预先设定初始状态和测试指导方案。

Monalisa Sarma 和 Rajib Mall[1]，通过使用 UML 的用例，序列和类(sequence and class)层次的状态图模型，来生成一个包含多种情节(scenarios)的序列的集合，该集合能够满足充分的系统状态覆盖率。所生成的测试集，由包含应用用例情景的执行序列构成。

本文提出了基于 UML 模型，一种新颖的自动生成系统测试测试用例的方法。这里使用用例图和序列图作为测试用例生成的信息来源。此方法生成的测试集能够覆盖操作错误、用例依赖性错误、初始化错误以及交互错误。

在提出的方法中，使用 UML 建模的时候我们结合图论的相关知识，对系统的用例图模型和序列图模型进行抽象，这有利于算法进一步的实现。

我们首先将用例图 UCD 和顺序图 SD 进行抽象；然后通过抽象出来的模型，进行用例图和顺序图两个级别意义上的遍历来自动生成测试例子，这里我们定义了两个覆盖率准则。

2 基本概念介绍

UML 作为一种对软件系统进行规约、构造、可视化和文档化的语言，融合了 Booth 方法、OMT 方法和 OOSE 方法的核心概念，取其精华、去其繁杂，形成了一个统一的、公用的、具有广泛适用性的建模语言。

其中用例图描述了系统提供的一个功能单元。序列图显示一个具体用例或者用例一部分的详细流程。

3 模型抽象

本文在常规 UML 模型基础上，运用图论知识，对序列图和状态图进行了抽象，方便了随后算法的具体实现。

3.1 抽象用例图

我们运用图论知识实现对用例图的抽象化,得到的图命名为 G1。具体方法如下：

首先，我们考虑 G1 的结点集合 $Node_{G1}$。

G1 的结点由 UCD 里的所有用例 $UC=\{U_1,U_2,...,U_i\}$ 和参与者 $A=\{A_1,A_2,...,A_j\}$ 的有限结点集合 $Node_{G1}=UC\cup A$。每个用例映射为 G1 的中间结点。所有参与者可以映射到 G1 为开始结点集合 $Snode_{G1}\subset\{A_1,A_2,...,A_j\}$ 或最终结点集合 $Fnode_{G1}\subset\{A_1,A_2,...,A_j\}$，或者两者皆是。开始结点集合表示这些参与者扮演着数据源的角色，而最终结点的集合表示这些参与者扮演着数据汇聚的角色。

其次，我们考虑 G1 的边集合 E_{G1}。

G1 的边 $E_{G1}=AU\cup UU$,体现了用例之间的依赖关系。其中 $AU=\{A\times U\}\cup\{U\times A\}$ 是操作者和用例之间联系所组成的集合，体现参与者与用例之间的消息交互；$UU=\{U\times U\}$ 表示两个用例之间的相互依赖关系。

最后 G1 结点中存储的数据 $Data_{G1}$。

$Data_{G1}=Data_{AU}\cup Data_{UA}\cup \mathrm{Pr}e_U\cup Pos_U$。其中，$Data_{AU}$ 表示从参与者到用例的数据，$Data_{UA}$ 表示从用例到参与者的数据，$\mathrm{Pr}e_U$ 表示用例的前置条件，Pos_U 表示用例的后置条件。

前置条件是一个条件列表。描述了执行用例之前系统必须满足的条件。这些条件必须在使用用例之前得到满足。前置条件在使用之前，已经由用例进行过测试。如果条件不满足，则用例不会被执行。

后置条件将在用例成功完成后得到满足，它提供了系统的部分描述。即在前置条件满足后，用例做了什么，以及用例结束时，系统处于什么状态。我们并不知道用例中止后系统处于什么状态。因此必须确保在用例结束时，系统处于一个稳定的状态。

3.2 抽象序列图

我们运用图论知识实现对用例图的抽象化,得到的图命名为 G2。具体方法如下：

首先，我们考虑 G2 的结点集合 $Node_{G2}$。

G2 的结点由 SD 里的所有场景和场景执行后的状态组成的有限结点集 $Node_{G2}=S_{con}\cup S$。其中 S_{con} 为场景执行后的状态，$S=\{S_1,S_2,...,S_j\}$ 代表一个序列图的所有场景。开始结点 $Fnode_{G2}\in S_{con}$，表示操作开始的状态。最终结点集合 $Fnode_{G2}\subset S_{con}$，表示操作结束的状态。

其次，我们考虑 G2 的边集合 E_{G2}。

G2 的边 E_{G2} 表示两个状态之间迁移时，经过的所

有边的集合。

最后 G2 结点中存储的数据 $Data_{G2}$。$Data_{G2} = O_{Attr} \cup P_{En} \cup G_{pre} \cup Attr_{Domain} \cup Pos_{Event}$。其中，$O_{Attr}$ 表示此状态相应对象的属性，P_{En} 表示入口参数，G_{pre} 表示监护条件的谓词，此信息从类图中获取。$Attr_{Domain}$ 表示在此状态下对象所有属性的取值范围，此信息可以从与给定设计相关的数据词典中获取。Pos_{Event} 表示一个事件发生后的理想结果。

需要指出的是，G2 只能有一个开始结点，但依赖于不同的操作场景，可以有一个或多个终止结点。

显然 G2 里的每个结点都被映射为两个对象 O_i 和 O_j 之间通过消息 m_k 的一个带或不带监护条件的交互。与此相关的信息需要在 SDG 相应的结点中进行存储。

4 测试覆盖标准

每个用例的执行均是多个对象协力完成的动作和。但是一个用例要成功执行，这些对象在用例开始时，必须在确定的期望状态。换句话说，一个用例的要按照规格执行并产生正确结果，必须存在正确的上下文。此外，针对于外界输入和系统状态的一些特定组合，每个用例指定了一个响应集合。如果某个用例没有服从期望的输入-输出关系，系统会导致操作错误。同样值得注意的是用例之间通常具有依赖关系。换句话说，为了完成一个任务，一些用例的执行是其他用例可以执行的必要条件。在这种情况下，一个用例可能产生一些中间的结果，来使得后来的用例能够成功执行。例如，在线购买系统中，在处理订单用例之前必须先建立订单。当一个用例在不满足必须的依赖条件的前提下开始执行，将产生错误。因此，一个测试集需要检测出上面提及到得错误。我们遵循下面陈述的覆盖标准 1 来获得测试集。

顺序图描述在一个操作过程中，不同对象之间各种可能存在的交互。在一个交互过程中可能发生几个错误，比如对一个消息的错误响应，正确的消息传递到错误的对象或者错误的消息传递到正确的对象，消息发起伴随着不适当的或者不正确的参数，消息传递到还未初始化的对象，不正确的或者丢失的输出等等。此外，一个顺序图描述了多个操作场景。在顺序图中，每一个场景对应于不同的由消息路径组成的序列。对于一个特定的操作场景，由于不正确的状态赋值或非正常终止等原因，消息序列可能不会遵循期望的路径。因此，测试集必须检测当一个对象调用另一对象的程序时，是否紧跟着正确的消息序列以完成操作。显然，由于 G2 覆盖了从开始结点到最终结点的所有路径，将最终覆盖所有交互和消息序列路径。我们遵循上面陈述的覆盖标准 2 来获得测试集。

因此我们提出如下两个覆盖原则来生成测试案例。

覆盖标准 1：所有用例和用例依赖关系标准： 假设测试集 tSet 和用例图 G1, tSet 必须使每个用例均发生并且使每个代表依赖性的路径执行至少一次。

覆盖标准 2：所有顺序图消息路径序列覆盖标准：假设测试集 tSet 和顺序图 G2，tSet 必须使每个消息序列路径执行至少一次。

5 生成测试用例

用例图中的每个用例都和个或多个序列图相对应。通过上述抽象后 G1 表示整个待测系统的用例图抽象结果，G1 的每个中间结点 Ui 都和一个或多个 G2 相对应，我们将其进行逻辑上的连接，及将 G2 嵌入到 G1 的中间结点中，构成了两个层次上的关系，从而形成一个新的 G1。

我们现在通过 G1 来生成测试例子。算法在两个级别上遍历 STG。遍历从 G1 开始，我们标志此遍历为一级遍历。此遍历访问所有的用例，并且生成测试案例用于检测初始化错误。二级遍历从相应的 G2 中已遍历的用例结点开始，生成用于检测操作错误的测试案例。最后,我们列举 G1 中所有的路径来确定所有用例的依赖关系，并生成用于检测用例依赖性错误的测试案例。

下面说明的算法满足覆盖标准 1 和覆盖标准 2。我们用伪码来表示算法的具体实现过程。

```
Algorithm
Input:
G1:运用图论知识对用例图的抽象。
G2:运用图论知识对序列图的抽象。
Output:
测试集 tSet
Begin
tSet = Φ//定义一个测试集
P = EnumerateAllPaths(G1)//列举 G1 中的所有路径
For each path pi∈P do
//遍历每条路径，生成针对于用例依赖性的测试集
UD = FindAllUseCaseDependency(G1)
//找到每条路径的所有用例依赖关系
    For each UDi∈UD do
        For eachUj ∈UDi do
```

```
                Pre =GetPreCondition (Uj)
                    //取得用例的前置条件
                Pos = GetPostCondition(Uj)
                    //取得用例的后置条件
                t = SelectTestCase{ Pre, Pos }
                    //生成针对于用例依赖性的测试集
                tSet ←tSet ∪t
            EndFor
    EndFor
EndFor
For each use case Ui ∈G1do
  //生成测试集 T1,使得所有用例至少发生一次
  O = FindObjects(Ui ) //找到和用例相关的所有对象
  For each oi ∈O Do
      Input= IdentifyInputDomain(oi)
          //确定每个对象的输入范围
      Output = IdentifyOutputDomain(oi)
          //确定每个对象的输出范围
      S=GetSystemstate();//获取系统状态的特定组合
    Pre= GetPreCondition(Ui) //取得用例的前置条件
    Pos =GetPostCondition(Ui) //取得用例的后置条件
     T1 = SelectTestCase(Input,Output,S,Pre, Pos)
      //生成测试集 T1,使得所有用例至少发生一次
      tSet ←tSet∪T1
  EndFor
 Link = Ui →G2 //建立每个用例和对应序列图的连接
  P = EnumerateAllPaths(G2) //列举 G1 中的所有路径
  For each path Pi ∈P do
      Node=FindNode(Pi)//寻找每条路径的所有结点
 Snode=FindStartNode(Pi) //寻找每条路径的开始结点
      preCi = FindPreCond (Node)
          //寻找开始结点的前置状态
      Ti ←Φ//定义一个测试集
       For each node nj ∈Node do
            Ej= FindEvent ( nj )
                //寻找每个结点相关的事件
            Rj =FindDesiredResult(Ej)
                //获得每个事件执行后的理想结果
       EndFor
       Fnode=FindFinalNode(Pi)
            //寻找每条路径的终止结点
       preC= FindPosCond (Fnode)
            //寻找开始结点的后置状态
       t={preCi, E,R, Gpre, postC}//合成测试集
        Ti = Ti∪t
      EndFor
      T1 ←T1 ∪Ti
  EndFor
  tSet = tSet ∪T1
EndFor
Stop
```

6 算法复杂度分析

本算法主要是两级遍历，首先对 G1 进行遍历，其次对 G1 对应的每个 G2 进行遍历。这里我们对 G1 采用深度优先搜索(DFS),实质上是寻找每个结点未被访问邻接点的过程，时间复杂度为 O(e)，其中 e 为图 G1 中边的条数。对 G2，我们采取改进的 DFS，寻找每条边依附结点的未访问邻接边，时间复杂度为 O(n)，其中 n 为图 G2 中边的条数。所以整个算法的时间复杂度对为：T= O(e)+N×O(n)，其中 N 为 G1 的结点数。

7 结论

本文提出了一种新颖的基于 UML 模型自动生成系统测试测试用例的方法。这里使用用例图和序列图作为测试用例生成的信息来源。此方法生成的测试用例集使得每个用例均发生并且使每个代表依赖性的路径执行至少一次，并且每个消息序列路径执行至少一次。测试集能够覆盖操作错误、用例依赖性错误、初始化错误以及交互错误。对算法进一步的优化将是下一步的主要工作。

参考文献

1 Sarma M, Mall R. System Testing using UML Models. Test Symposium, 2007, ATS07. 2007.

2 Briand L, Labiche Y, A UML-Based Approach to System Testing. Journal of Software and Systems Modeling, Springer Verlag, 2002,1:10－42.

3 Riebisch M, Philippow I, Gotze M. UML-Based Statistical Test Case Generation. Proc. of ECOOP 2003. Springer Verlag, LNCS 2591, 2003. 394－411.

4 Briand LC, Labiche Y, Cui J. Automated Support for Deriving Test Requirements from UML Statecharts. Journal of Software and System Modeling, 2005,4(4):399－423.

5 Hartmann J, Vieira M, Foster H, Ruder A. A UML-based Approach to System Testing. Journal of Innovations System Software Engineering, 2005,1:12－24.