一种基于动态依赖关系的类集成测试方法

张艳梅 姜淑娟 张红昌
(中国矿业大学计算机科学与技术学院 江苏 徐州 221116)

摘 要 类间集成测试是面向对象软件测试的重要组成部分,合适的测试顺序能够极大地节省测试成本. 类间依赖关系构成环路的情况下,需要删除某些依赖关系以消除环路,同时需要引进测试桩. 忽略类间动态依赖关系导致测试桩的数目不足,难以完成测试. 文中提出一种基于动态依赖关系的类集成测试方法. 首先分析了类之间的静态依赖和动态依赖关系;然后在保证测试桩的数目尽可能少的前提下,给出了边的删除规则以及消除由静态依赖关系和动态依赖关系形成的环路的算法,在此基础上,进一步提出测试顺序分配策略和算法;最后针对提出的方法开发了基于测试级的类测试序列自动生成工具——TLOG. 实验结果表明:该方法较其它方法需要较少的测试桩,测试效率有明显提高.

关键词 集成测试;测试顺序;测试桩;动态依赖;测试级
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01075

An Approach for Class Integration Testing Based on Dynamic Dependency Relations

ZHANG Yan-Mei JIANG Shu-Juan ZHANG Hong-Chang
(*School of Computer Science and Technology, China University of Mining and Technology, Xuzhou Jiangsu* 221116)

Abstract Class integration testing is an important part in object-oriented software testing. An appropriate test order for software testing can reduce test cost. In the presence of cyclic dependency calls, the test order problem can be solved by removing relationships to break cycles and then create test stubs. There are insufficient test stubs to complete testing if ignoring dynamic dependency relations. In this paper, an approach for class integration testing based on dynamic dependency is proposed. First, inter-class static and dynamic dependencies are analyzed. Then, under the premise of minimizing the number of test stubs, rules of edge deletion are given, in addition, an algorithm of eliminating the cycles that formed by static and dynamic dependency is presented. Furthermore, an integration testing order strategy and an algorithm are given. Finally, the whole approach is implemented in a test levels order generator——TLOG. The experiment results show that the method requires less test stubs than others and improves the test efficiency obviously.

Keywords integration testing; test order; test stubs; dynamic dependency; test levels

1 引 言

面向对象程序中,类间测试顺序的确定是类间集成测试中最重要的问题[1]. 不同的类测试序列,需要不同的测试代价,所以,尽可能减少测试代价是确定类间测试顺序的重要目标. 如果类的静态依赖关系中不存在环路,类间测试顺序问题可以通过简单

收稿日期:2010-10-19;最终修改稿收到日期:2011-05-03. 本课题得到国家自然科学基金(60970032)、教育部科学技术研究重点项目(108063)、江苏省自然科学基金(BK2008124)、江苏省"青蓝工程"、江苏省研究生培养创新工程项目(CX10B_157Z)资助. **张艳梅**,女,1982 年生,博士研究生,研究方向为软件分析与测试、异常处理等. E-mail: ymzhang@cumt.edu.cn. **姜淑娟**,女,1966 年生,博士,教授,博士生导师,主要从事程序设计语言、编译技术、软件工程等方面的教学与研究工作. E-mail: shjjiang@cumt.edu.cn. **张红昌**,男,1989 年生,硕士研究生,主要研究方向为软件分析与测试等.

的逆向拓扑排序来解决,而对于存在环路的情况,测试人员必须从对象关系图中删除某些依赖关系,以打破其中的环路. 在删除依赖关系时,需要引进测试桩. 测试桩指的是当 $C1$ 依赖 $C2$,增量集成过程中,$C1$ 集成时,但 $C2$ 尚未被集成,用来模拟 $C2$ 的服务组件.

节省测试成本是选择测试顺序的一个重要目标,而在面向对象程序的测试中,开发测试桩是一项成本很高的工作,这是因为根据测试桩的定义可知,测试桩需要模拟的是待测试对象所依赖的类的服务组件,而由于一个类的对象行为依赖于对象当前状态,并且一个行为需要涉及到多个对象,因此要准确地模拟一个类的对象行为需要理解所有与之相关的类的对象行为,所以节省开发测试桩的开销是确定类间测试顺序的一个重要任务. 减少开发测试桩成本的方法主要有两类:一类是最小化所需测试桩的数目,另一类是最小化测试桩的复杂度. 测试桩复杂度是用来衡量构造一个测试桩的难易程度. 测试桩的复杂度不易控制,因为:(1)测试桩的复杂度标准难以制定,Kraft 等人[2]提出了一套测试桩的复杂性标准,但根据案例研究表明,他们的标准在某些情况下并不可行;(2)测试桩的复杂度在某种程度上取决于开发人员的编程能力,由于面向对象编程具有极大的灵活性,不同开发背景的程序员开发同一个类的测试桩,完全有可能生成复杂度相差很大的测试桩;(3)Tai 等人[3]将测试顺序应用到集成测试中,他们证明先构造类的测试顺序,然后按照类的测试顺序来进行集成测试的方法,可以在一定程度上减少测试桩的数量. 因此,我们采用的是最小化测试桩数目的方法.

现有类间测试顺序研究方法大多只限于静态分析[1,4-6],而类间的动态依赖关系比较普遍,动态依赖关系同样会导致类关系图中环的形成,对类间测试序列产生影响,在删除动态依赖关系时,需要引进动态依赖关系测试桩. 忽略类间动态依赖关系将导致测试桩的数目不足,使得测试不充分.

本文主要研究基于动态依赖关系的类测试顺序的确定方法,解决忽略类间动态依赖关系所导致的测试桩的数目不足而难以完成测试的问题,通过花费尽可能少的测试桩找到一个最佳的测试顺序. 需要解决两个问题:静态依赖关系和动态依赖关系构成的环路中边的删除规则以及类测试顺序分配策略.

针对要解决的问题,为测试提供足够的测试桩,同时遵循满足构造的测试桩最少的原则,本文首先提出一组由静态依赖关系和动态依赖关系构成的环路中边的删除规则,通过删除弱联系边,解决已有方法中断开继承、组合和聚集等强联系关系增加测试成本的问题;然后在消除环路的基础上给出基于测试级的测试顺序分配策略.

2 类间的依赖关系

面向对象程序类间的依赖关系主要包括两类:一类是静态依赖关系,一类是动态依赖关系. 静态依赖关系指的是整个程序代码的静态结构中反映出来的类与类之间的关系. 动态依赖关系是指类在程序运行期间形成的一种依赖关系.

2.1 类间的静态依赖关系

静态依赖关系指的是整个程序代码的静态结构中反映出来的类与类之间的关系. 面向对象程序中,类间的静态关系主要有继承关系、聚集关系和关联关系.

(1)继承指的是子类可以共享父类(或祖先类)属性和操作. 父类中定义了其所有子类的公共属性和操作,在子类中除了定义自己特有的属性和操作外,还可以对父类(或祖先类)中的操作重新定义其实现方法. 如果类 A 继承类 B,则类 A 称为类 B 的子类,并且类 A 能继承类 B 的成员(数据成员和成员方法).

(2)如果类 A 聚集类 B,则说明类 A 的数据成员具有一个或多个类 B 的实例.

(3)关联表示类的实例之间存在的某种关系,A 的实例映射到 B 的实例,这种关系叫关联. 如果类 A 关联类 B,则说明类 A 的成员方法使用了类 B 的实例(如,类 A 的成员方法包含 B 类型的参数或成员方法内部包含 B 类型的局部变量).

对于基于图论的方法,类簇以及它们之间的关系可以抽象为对象关系图(ORD)[1],ORD 用一个有向图 $G(V,E)$ 表示,其中,V 是由一组代表类的节点构成的集合,E 是由一组代表类间关系的有向边构成的集合. 图 1 显示了一个典型的 ORD.

图 1 是对象关系图(ORD)示例. 图中节点表示程序中的类,实心有向边表示类之间的继承、聚集和关联关系. 对于任何类 $C1$ 到 $C2$ 之间的有向边:

(1)标有 I 的边表示 $C1$ 是 $C2$ 的子类;

(2)标有 Ag 的边表示 $C1$ 是 $C2$ 的一个聚集类($C1$ 包含一个或多个 $C2$ 对象);

(3)标有 As 的边表示 $C1$ 与 $C2$ 关联.

类之间的继承、聚集和关联关系使得在整个 ORD 中存在着依赖关系.准确地说,ORD 描述的是类之间的静态依赖关系.我们可以依据文献[7]中对类之间静态依赖关系的定义来进行判断.

定义 1.　对于一个 ORD 中的类 X 和 Y,X 静态依赖 Y,当且仅当在 ORD 中存在一条从 X 到 Y 的有向路径.令 Rs 表示 ORD 中一条有向边的二元关系:$Rs=\{(C1,C2)|$在 ORD 中存在一条从 $C1$ 到 $C2$ 边$\}$,则类 X 静态依赖的类的集合 $S(X)=\{Y|\langle X,Y\rangle\in Rs^+\}$,其中,$Rs^+$ 表示 Rs 的传递闭包.

例如,图 1 所示为一个面向对象程序 P 中的 5 个类{教师,学生,课程,本科生,成绩}构成的 ORD.其中,本科生是学生的一个子类,学生和课程是成绩的聚集类以及学生关联课程、课程关联教师等.

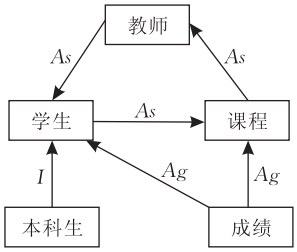


图 1　对象关系图(ORD)示例

对于面向对象集成测试中的每一个类,可用一个类的集合 $S(X)$ 对类的静态依赖关系进行描述.$S(X)$ 表示 X 在编译阶段静态依赖的类的集合.

2.2　类之间的动态依赖关系

动态依赖关系是指类在程序运行时期形成的一种依赖关系.若类 A 继承于类 B,且重写了类 B 的虚方法;类 B 是类 C 的服务类,即 C 要么和 B 相关联要么是 B 的聚集类,且调用了类 B 中被类 A 重写的虚方法,则在程序运行时期,C 和 B 的子类 A 动态绑定,类 C 动态依赖于类 A.如果类 A 继承于类 B,类 B 是类 C 的服务类,即 C 要么和 B 相关联要么是 B 的聚集类,则在程序运行时期,C 可能会和 B 的子类 A 动态绑定,类 C 可能会动态依赖于类 A.本文是在 ORD 的基础上进行类间分析的,因此我们将可能存在的动态依赖关系都标记在 ORD 中.下面给出动态依赖关系的定义.

定义 2[7].　类间的动态依赖关系用 Rd 表示,令 $Rd=\{\langle C1,C3\rangle|$存在一个类 $C2$,它是 $C3$ 的父类,且 $C2$ 被 $C1$ 调用$\}$,则类 X 动态依赖的类的集合 $D(X)=\{Z|\langle X,Z\rangle\in Rd^+\}$,其中,$Rd^+$ 表示 Rd 的传递闭包.

我们可以从类之间的静态依赖关系得出动态依赖关系,例如:图 2 中,教师关联学生,学生是本科生的父类,在执行阶段,教师可能动态依赖于本科生.动态依赖边 Dy 用虚线有向边表示.

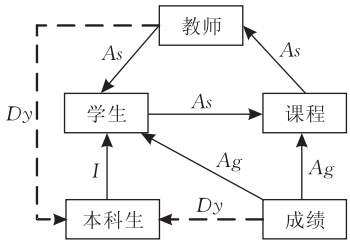


图 2　扩展对象关系图(EORD)示例

推论 1.　由定义 1 和定义 2 我们可以推导出类 X 在编译阶段静态依赖和在执行阶段动态依赖的类的集合 $SD(X)=\{Ck|\langle X,Ck\rangle\in(Rs^+\cup Rd^+)\}$.

证明.　由 $SD(X)$的定义可知,$SD(X)=S(X)\cup D(X)$.由 $S(X)$和 $D(X)$的定义可知,$S(X)=\{Y|\langle X,Y\rangle\in Rs^+\}$,$D(X)=\{Z|\langle X,Z\rangle\in Rd^+\}$,则 $SD(X)=\{Y\cup Z|\langle X,Y\rangle\in Rs^+,\langle X,Z\rangle\in Rd^+\}$成立,令 $Ck=Y\cup Z$.由于二元关系可以用笛卡尔积来表示,因此,X 与 Y 的二元关系 $X\to Y$ 可以记为 $X\times Y\in Rs^+$,X 与 Z 的二元关系 $X\to Z$ 可以记为 $X\times Z\in Rd^+$.因此,X 与 Ck 的二元关系 $X\to Ck$(即 $X\to(Y\cup Z)$)可以记为 $X\times Ck=X\times(Y\cup Z)=(X\times Y)\cup(X\times Z)=Rs^+\cup Rd^+$.因此,$SD(X)=\{Ck|\langle X,Ck\rangle\in(Rs^+\cup Rd^+)\}$.

定义 3.　扩展的对象关系图(EORD).在对象关系图 ORD 中增加表示动态依赖关系的边,用标有 Dy 的虚线边表示,可得到扩展的对象关系图,记为 $EORD=\{V,E\}$,其中,V 为表示程序中的各个类节点,$E=\{E_I\cup E_{Ag}\cup E_{As}\cup E_{Dy}\}$,$E_I$、$E_{Ag}$、$E_{As}$ 分别表示类间继承边、聚集边和关联边.

图 1 的 ORD 中没有包含动态依赖信息,图 2 是扩展后的对象关系图(EORD),其中动态依赖用虚线箭头表示,如类教师与类本科生之间存在动态依赖.

图 1 中存在一个环路:教师→学生→课程→教师.但是,由于动态绑定特性,当测试类教师时,类本科生中的方法可能被执行.也就是说,在程序执行阶段,类教师可能依赖类本科生,类成绩可能依赖类本科生.所以,正确的依赖如图 2 所示.图 2 中存在两个环路,使原来的 ORD 的环路增加.为了确定类测试序,需要消除在图 1 和图 2 两种情况下的环路.由于动态依赖边的引入对环路产生影响,因此消除

环路时所删除的边也需要随之变化.

3 消除环路算法

对于存在环路的 EORD,需要消除环路进而给出类间测试序列.因此,确定类间测试顺序的核心问题就是打破环路.Kung 等人[1]、Kraft 等人[2]、Briand 等人[8]都认为给不同类型的边创建测试桩的难易程度的高低和代价 $Cost(C)$ 均存在如下关系:C(关联边)=C(动态依赖边)<C(聚集边)≪C(继承边).因此,他们称继承关系和聚集关系为强联系关系,动态依赖关系与关联关系均为弱联系关系.为了减少测试代价,本文消除环路时避免删除强联系边;同时,由于本文的解决方案旨在降低测试桩的数目,因此,我们在打破环路的过程中遵循删除涉及环路最多的关联边或者动态依赖边的规则.其中关键问题是首先需要识别出 EORD 中由类以及它们之间的依赖关系形成的强连通分量($SCCs$),然后查找每一个子强连通分量($SCCi$)中所有的环路,统计 $SCCi$ 中每条动态依赖边和关联边所涉及的环路数目,进而将一个有环图消除环路成为一个无环图.

3.1 EORD 中的环路

对于 EORD 中由类以及它们之间的依赖关系形成的 $SCCs$,我们可以通过 Tarjan 等人[9]的算法进行识别.本节以一个 EORD 为例介绍子强连通分量($SCCi$)中每条动态依赖边和关联边所涉及的环路数目的计算方法.每一个 $SCCi$ 中的环路查找方法将在 4.3 节的环路生成模块中再作具体介绍.

假设 $CP(P,Q)$ 表示边 $P\rightarrow Q$ 涉及的环路数目.如图 3 中,$CP(G,C)=2$,环路分别为 $G\rightarrow C\rightarrow D$,$G\rightarrow C\rightarrow F$.

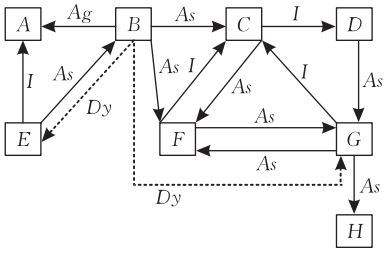


图 3 扩展对象关系图(EORD)示例

在图 3 中,不考虑动态依赖关系的情况下,类 C、D、F、G 构成一个 $SCC\{C,D,F,G\}$,其中各关联边的环路数目:$CP(C,F)=2$,$CP(D,G)=2$,$CP(F,G)=2$,$CP(G,F)=2$.考虑动态依赖关系后,$B\rightarrow E$ 使 ORD 构成了 $SCC\{B,E\}$,增加了环路 $B\rightarrow E\rightarrow B$.因此,对于 $SCC\{B,E\}$,$CP(B,E)=1$,$CP(E,B)=1$.

3.2 消除 EORD 中环路的方法

3.2.1 边的删除规则

对于存在环路的 EORD,首先需要消除环路.删除哪些边消除环路将直接影响到构造测试桩的数量.为了解决已有的方法忽略动态依赖关系的问题,为测试提供足够的测试桩,需要考虑动态依赖边对打破环路的影响.同时为了满足构造的测试桩最少,我们应该遵循删除最少的边打破尽量多的环路的原则.下面给出相关的删除规则.

定理 1. B 是 C 的父类,又是 A 的服务类,即 A 要么和 B 相关联要么是 B 的聚集类.如果 A 在 B 和 C 之前进行测试.如果 B 是非抽象类,不需要为 C 构造测试桩,只需为 B 构造测试桩.

证明. 已知 B 是 C 的父类,又是 A 的服务类,根据动态依赖关系的定义可知,在程序运行时期,A 可能会和 B 的子类 C 动态绑定,即 A 可能会动态依赖于 C,因此,当测试 A 时,需要依赖 B 和 C.如果 B 是非抽象类,B 能实例化,它作为服务类的作用不需要通过其子类 C 实例化来提供,在这种情况下,不需要为 C 构造测试桩,因此,B 是非抽象类时,只需为 B 构造测试桩.

推论 2. 假设 λ 是无环的 ORD,节点 A,B,$C\in\lambda$,B 是 C 的父类,又是 A 的服务类,且 $A\rightarrow B$ 是关联边,动态依赖边 $A\rightarrow C$ 使得 EORD 产生了环路 $A\rightarrow C\rightarrow A$,边 $A\rightarrow C$ 和边 $C\rightarrow A$ 没有与其它节点构成新的环路,为了构造尽可能少的测试桩,如果 B 是非抽象类,且 $C\rightarrow A$ 是非关联边,则直接删除动态依赖边 $A\rightarrow C$;当 $C\rightarrow A$ 是关联边,则可以删除边 $A\rightarrow C$ 或边 $C\rightarrow A$.

证明. 由定理 1 我们可以推断,如果 B 是非抽象类,删除关联边 $A\rightarrow B$ 时,动态依赖边 $A\rightarrow C$ 随之消失,由于动态依赖边 $A\rightarrow C$ 使得 EORD 产生了环路 $A\rightarrow C\rightarrow A$,则此时已经消除了该环路.如果 A 在 B 和 C 之前进行测试,需要为 B 和 C 构造测试桩.事实上,直接删除边 $A\rightarrow C$ 可以消除环路,只需要为 C 构造测试桩,此时,我们选择直接删除边 $A\rightarrow C$.如果 $C\rightarrow A$ 是关联边,也可以通过删除边 $C\rightarrow A$(由于关联边为弱联系边)来消除环路,否则,只能删除边 $A\rightarrow C$.

推论 3. 假设 λ 是无环的 ORD,节点 A,B,$C\in\lambda$.B 是 C 的父类,又是 A 的服务类,且 $A\rightarrow B$ 是聚集边,同时动态依赖边 $A\rightarrow C$ 使得 EORD 产生了环路 $A\rightarrow C\rightarrow A$,边 $A\rightarrow C$ 和边 $C\rightarrow A$ 没有与其它节点

构成新的环路.为了构造尽可能少的测试桩,如果 $C\rightarrow A$ 是关联边,则可以删除边 $C\rightarrow A$ 或者 $A\rightarrow C$,相应地为 A 或 C 构造测试桩;否则,只能删除边 $A\rightarrow C$,为 C 构造测试桩.

证明.　已知 B 是 C 的父类,又是 A 的服务类,根据动态依赖关系的定义可知,则在程序运行时期,A 可能会动态依赖于 C.由于动态依赖边 $A\rightarrow C$ 使得 EORD 产生了环路 $A\rightarrow C\rightarrow A$,可以删除边 $C\rightarrow A$ 或者 $A\rightarrow C$ 或者 $A\rightarrow B$,已知 $A\rightarrow B$ 是聚集边,将其排除(Kraft 等人[2]已经证明聚集是强联系边,本文避免删除该类型的边);因此,当 $C\rightarrow A$ 是关联边,则可以删除边 $C\rightarrow A$,为 A 构造测试桩,测试顺序为 BCA,或者删除 $A\rightarrow C$,测试顺序为 BAC,为 C 构造测试桩;否则,只能删除边 $A\rightarrow C$,为 C 构造测试桩.

例如,由于考虑了动态关系,图 2 中生成了一个新的环路:教师→本科生→学生→课程→教师.根据推论 1,删除关联边教师→学生,可以打破该环路.此外,删除关联边教师→学生还可以打破图中的另一个环路:教师→学生→课程→教师.满足了删除最少的边打破尽量多的环路原则.因此,我们选择删除动态依赖边教师→学生.

3.2.2　EORD 中的环路消除算法

如前面所提到的,消除环路是确定类间测试序的关键步骤.根据 3.2.1 节提出的边的删除规则,我们给出相应的环路消除算法:首先统计动态依赖边和关联边所涉及的环路数目(行 1～5),然后判断涉及环路数目最多的边的类型:如果涉及环路数目最多的边为动态依赖边,或者动态依赖边和关联边涉及相同的环路个数(包括由一条动态依赖边和一条关联边构成一个最简单的环路的情况),判断导致动态依赖边形成的边的类型(行 6～8):如果是关联边,删除动态依赖边;如果是聚集边,也删除动态依赖边(行 9～14).如果两个类之间同时存在一条关联边和一条动态依赖边(同向),且它们涉及的环路数目相同且最多,则同时删除这两条边(行 15～17);如果两个类之间同时存在一条聚集边(或继承边)和一条动态依赖边(同向),且它们涉及的环路数目相同且最多,则删除环路中的关联边(行 18～20)(由于 Kung 等人[1]已经证明 ORD 中的任意一个 SCC 中至少包含一条关联边).如果涉及环路数目最多的边为一条关联边,则删除该关联边(行 21～23).如果多条关联边(或动态依赖边)的环路数目相同且最多,则删除其中任意一条关联边(或动态依赖边)(行 24～26).环路消除算法如算法 1 所示.

算法 1.　$EORD(V,E)$的环路消除算法.

输入:$EORD(V,E)$

输出:无环的 EORD

```
Begin
 1. find all SCCs in EORD;
 2. for(each SCCi(Vscci;Escci )∈SCCs) do
 3.   search cycles(…);
 4.   for (each association edge As and dynamic depend-
        ency edge Dy∈Escci) do
 5.    calculate the number of cycles that involve As and Dy;
 6.    while(totalCycle!=0)do
 7.      if Dy has the most cycles or Dy and As have the
          same number of cycles(include the situation
          of Dy and As constitute a simple cycle of two
          classes) do
 8.        judge if Dy is lead by another association edge As';
 9.        if lead by As' then
10.          remove the dynamic dependency edge Dy;
11.        else lead by aggregation edge Ag
12.          remove the dynamic dependency edge Dy or
               the association edge As;
13.        endif
14.      endif
15.      if two classes have an As and a Dy in the same
           direction and have the most cycles do
16.          remove the As and Dy simultaneously;
17.      endif
18.      if two classes have an Ag(or I) and a Dy in the
           same direction do
19.          remove the other association edge of the cycle;
20.      endif
21.      if an association edge As has the most cycles then
22.        remove the association edge As;
23.      endif
24.      if more than one association(dynamic dependen-
           cy)edges have the same number of cycles then
25.        remove any association(dynamic dependency)edge;
26.      endif
27.        totalCycle=totalCycle-number of cycles broken;
28.        recalculate the number of cycles for remaining
             association edges and dynamic dependence
             edges;
29.    endwhile
30.  endfor
31. endfor
End
```

下面把图 3 中示例应用到该算法中,对算法的具体步骤作如下说明:

执行第 2～3 行能够在 $SCC\{C,D,F,G\}$ 中找出 5 个循环,如表 1 所示.

表 1 $SCC\{C,D,F,G\}$中环路

No.	Cycles
1	$C\rightarrow F\rightarrow C$
2	$C\rightarrow F\rightarrow G\rightarrow C$
3	$C\rightarrow D\rightarrow G\rightarrow C$
4	$F\rightarrow G\rightarrow F$
5	$C\rightarrow D\rightarrow G\rightarrow F\rightarrow C$

表 2 $SCC\{C,D,F,G\}$中各关联边涉及的环路

No.	Edges	Cycles involved	NC
1	$C\rightarrow F$	{1,2}	2
2	$D\rightarrow G$	{3,5}	2
3	$F\rightarrow G$	{2,4}	2
4	$G\rightarrow F$	{4,5}	2

根据算法 1 中的第 4～5 行计算 $SCC\{C,D,F,G\}$中各条关联边和动态依赖边涉及的环路数目，结果如表 2 所示. 其中，NC 表示环路的数目. 由算法 6～17行可以得出边 1:$C\rightarrow F$，边 2:$D\rightarrow G$，边 3:$F\rightarrow G$ 和边 4:$G\rightarrow F$ 涉及环路数目相同，均为 2，我们任意删除其中的一条关联边，此处，假如删除关联边 2，$D\rightarrow G$，可以打破两个环路{3,5}，剩余环路 1,2,4.

接下来，应用算法的第 28 行对剩余的边重新计算涉及环路数目，可以得出两条拥有最多环路数目的关联边分别为 $C\rightarrow F$ 和 $F\rightarrow G$，且删除这两条边中任意一条打破的环路数目相同，所以可以任意选择一条边，这里，我们选择删除边 $C\rightarrow F$，打破环路 1，2，此时，只剩下环路 4，该环路由边 $F\rightarrow G$ 和 $G\rightarrow F$ 构成，这两条边为具有相同环路的关联边，可以任意删除其中的一条边，我们选择删除 $F\rightarrow G$，则消除了 $SCC\{C,D,F,G\}$中环路.

对于 $SCC\{B,E\}$，根据算法 1，需要删除边 $E\rightarrow B$ 打破环路. 此时，EORD 成为了无环图，如图 4 所示.

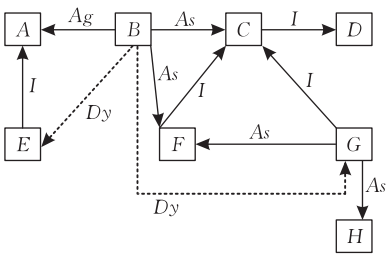


图 4 消除环路后扩展的对象关系图(EORD)

打破 EORD 中所有环路需要删除 $D\rightarrow G$，$C\rightarrow F$，$F\rightarrow G$，$E\rightarrow B$ 这 4 条边，分别为这 4 条边的源类 G、F 和 B 各自创建 1 个测试桩，共需要 3 个测试桩.

4 测试顺序分配策略

在程序的执行过程中，消除 EORD 中的环路以后，程序中仍存在动态依赖关系，但这些动态依赖关系并没有使得 EORD 构成环路，即无环的 EORD 中既包含静态依赖关系又包含动态依赖关系. 由于动态依赖关系在程序运行时期才会存在，同时，传统的逆向拓扑排序只能在程序静态状态下确定类测试顺序，因此不能通过简单的逆向拓扑排序为存在两种依赖关系类簇确定测试顺序. 因此，为了解决无环的 EORD 中的动态依赖关系带来的新的测试问题，我们给出基于测试级的测试顺序分配策略.

4.1 测试级的定义

测试级是指根据静态依赖和动态依赖关系为被测类分配的两个测试级别：静态测试级和动态测试级. 其中，静态测试级是指不运行被测程序本身，对程序进行静态分析，仅通过分析或检查源程序中的待测目标类来检查其正确性. 静态测试的结果可用于进一步查错，并为测试用例选取提供指导. 动态测试级是指需要通过运行被测程序，来检查程序的正确性，该方法由三部分组成：构造测试实例、执行程序、分析程序的输出结果.

一个测试级 T 可以表示为一个三元组 $T=(T.need,T.all,T.type)$[7]. 其中，$T.need$ 为被测试类的集合；$T.all$ 为被测类与被测类所依赖的类构成的并集；$T.type$ 为测试的类型. 其中，测试类型包括两种，一种是静态测试，用 S 表示；另一种是动态测试，用 Dy 表示.

下面具体介绍如何为消除环路后的 EORD 的每一个类确定测试级，即如何划分被测类的测试级别.

(1) 为了程序的精确测试，对于 EORD 中的每一个类 X，为每个类定义一个静态测试级 $T=(\{X\},\{X\}\cup S(X),S)$；

(2) 对满足 $D(X)\neq\varnothing$ 的类 X，定义一个动态测试级 $T=(T.need,T.all,Dy)=(\{X\},\{X\}\cup D(X),Dy)$.

表 3 所示为图 4 中无环 EORD 的 $S(X)$ 和 $D(X)$，首先为每一个类各自定义一个静态测试级，其中类 B 满足 $D(X)\neq\varnothing$，那么为 B 定义动态测试

表 3 图 4 中 EORD 的 $S(X)$和 $D(X)$

类 X	$S(X)$	$D(X)$
A	$\varnothing$	$\varnothing$
B	{A,C,D,F}	{A,C,D,E,F,G,H}
C	$D\}$	$\varnothing$
D	$\varnothing$	$\varnothing$
E	{A }	$\varnothing$
F	{C,D}	$\varnothing$
G	{C,D,F,H}	$\varnothing$
H	$\varnothing$	$\varnothing$

级，因此类 B 有两个测试级. 按上述方式，可定义图 4 中 EORD 的所有测试级如表 4 所示.

表 4　图 4 中 EORD 的测试级

T.need	*T.all*	*T.type*
$\{A\}$	$\{A\}$	S
$\{B\}$	$\{A,B,C,D,F\}$	S
$\{C\}$	$\{C,D\}$	S
$\{D\}$	$\{D\}$	S
$\{E\}$	$\{A,E\}$	S
$\{F\}$	$\{C,D,F\}$	S
$\{G\}$	$\{C,D,F,G,H\}$	S
$\{H\}$	$\{H\}$	S
$\{B\}$	$\{A,B,C,D,E,F,G,H\}$	Dy

4.2　测试级顺序分配策略

每一个类的测试级确定之后，需要为它们分配一定的测试顺序. 类测试顺序由类的测试依赖性决定[10]. 测试依赖性表示一个类依赖于其它类的程度. 为了确定类测试顺序，本文从类测试依赖性入手，提出两个测试依赖性定理作为确定类测试顺序的依据.

定理 2.　当类 A 是类 B 的一个子类，或者类 A 是类 B 的一个聚合类，或者类 A 是类 B 的一个关联类，则在集成测试时，类 A 依赖于类 B，类 A 在类 B 之后进行测试.

证明.　类 A 继承于类 B 时，类 A 会继承类 B 的部分属性，从而类 A 依赖于类 B. 若类 B 中被类 A 继承的成员发生变化，或被类 A 继承的成员有直接或间接影响的成员发生变化时，将会影响类 A 的行为.

当类 A 是类 B 的一个聚合类，由于聚合是整体和个体的关系，即若干个类 B 聚合成一个类 A，则类 B 的改变必然会影响类 A，因此类 A 依赖于类 B.

当类 A 是类 B 的一个关联类，则类 A 能够访问类 B 的数据成员，或者类 A 传递一个消息到类 B. 因此，如果类 B 的数据成员发生变化，或者当类 B 接收类 A 发送的消息，并且类 B 的成员函数发生变化，则类 B 对消息的响应会有变化，返回给类 A 的结果也会发生变化. 因此，类 A 依赖于类 B.

因此，类 A 在类 B 之后进行测试. 得出定理 2 的结论.

定理 3[11].　假设 3 个类 A、B、C，当 A 是 B 的一个子类，B 是 C 的一个服务类，即 C 关联于 B（或者依赖于 B）或者 C 是 B 的一个聚合类，则在考虑多态性的情况下，集成测试时，C 依赖于 B，也依赖于 B 的子类 A. 测试顺序为 B，A，C.

证明.　A 是 B 的一个子类，B 是 C 的一个服务类，由测试依赖性定理 2 得出，A 在 B 之后测试，C 在 B 之后测试. 对于 A、C 的测试依赖性关系有如下情况：

（1）若 C 关联于 B（或者依赖于 B）. 如果 C 需要访问 B 中的某个数据成员，而 A 继承 B 的该数据成员. 由于继承和多态性，在程序运行时 C 可能访问到的实际上是 A 中的该数据成员. 因此 C 依赖于 A. 但由于从 A 到 C 之间没有任何依赖关系，因此，A 不依赖于 C.

（2）若 C 是 B 的一个聚合类，由于 A 是 B 的子类，而 C 是 B 的聚合类，因此 C 也可以作为 A 的聚合类. 该情况下 C 依赖于 A.

（3）如果 C 传递一个消息到 B，而 B 负责具体处理消息. 由于继承和多态性，继承了 B 类处理方法的 A 也能够处理该消息，C 也能够将消息传递到 A，由 A 负责具体处理. 该情况下 C 依赖于 A.

由上述分析得出，测试顺序为 B，A，C. 得出定理 3 的结论.

因此，对于消除环路以后的 ORD 或者 EORD，根据定理 2 和定理 3，测试级顺序需要满足：首先考虑静态依赖，因为动态依赖只是潜在的运行期间的动态绑定依赖关系，弱于静态依赖关系，即在测试一个类之前，该类所依赖的所有类都已被测试，且在对一个类进行动态测试之前，已完成了对所有类的静态测试. 这样，满足测试类所需要的类都已经被测试，可以保证测试桩的数量尽可能少. 根据这一要求，应该遵循以下测试级顺序分配规则.

（1）若类 $C1$ 和类 $C2$ 有两个不同的静态测试级 T_{C1} 和 T_{C2}，且 T_{C1} 静态依赖 T_{C2}，则 $C2$ 的静态测试级 T_{C2} 先于 $C1$ 的静态测试级 T_{C1}.

（2）若类 $C1$ 和类 $C2$ 有两个不同的动态测试级 T_{C1} 和 T_{C2}，且 T_{C1} 动态依赖 T_{C2}，则 T_{C2} 先于 T_{C1}.

（3）若类 C 有 2 个测试级，静态测试级 $Ts=(\{X\},X\cup S(X))$ 和动态测试级 $Td=(X,X\cup SD(X))$，其中，$X\in T.all$，则 Ts 先于 Td.

（4）若类 $C1$ 的一个动态测试级是 $T_{C1}=(T_{C1}.need,T_{C1}.all)$，并且 $C2\in T_{C1}.all$，$C1\neq C2$，则对 $C2$ 的静态测试级 $T_{C2}=(\{C2\},C2\cup S(C2))$ 先于 T_{C1}.

无环的 EORD 中类测试顺序分配算法如算法 2 所示.

算法 2.　无环的 EORD 中类测试顺序分配算法.

输入：无环的 EORD

输出：无环的 EORD 中类测试顺序分配算法

```
Begin
1. for each class X of acyclic EORD do
2.   find X′ static dependency set X_s, add X_s to S(X);
3.   SetStaTstLvl(X,S(X)); //为 X 建立静态测试级
4.   find X′ dynamic dependency set X_d, add X_d to D(X);
5.   SD(X)=(S(X)∪D(X))^+
6.    if S(X)⊂SD(X) then
7.      SetDynTstLvl(X,D(X));
                          //为 X 建立动态测试级
8.    endif
9.  endfor
10.    determine total number of test level N;
11.    order the test level N according to integration
         test order rule;
End
```

4.3 测试序列生成器 TLOG

我们根据上述基于测试级的类集成测试方法设计并实现了一个工具——TLOG(Test Level Order Generator),其功能结构如图 5 所示.

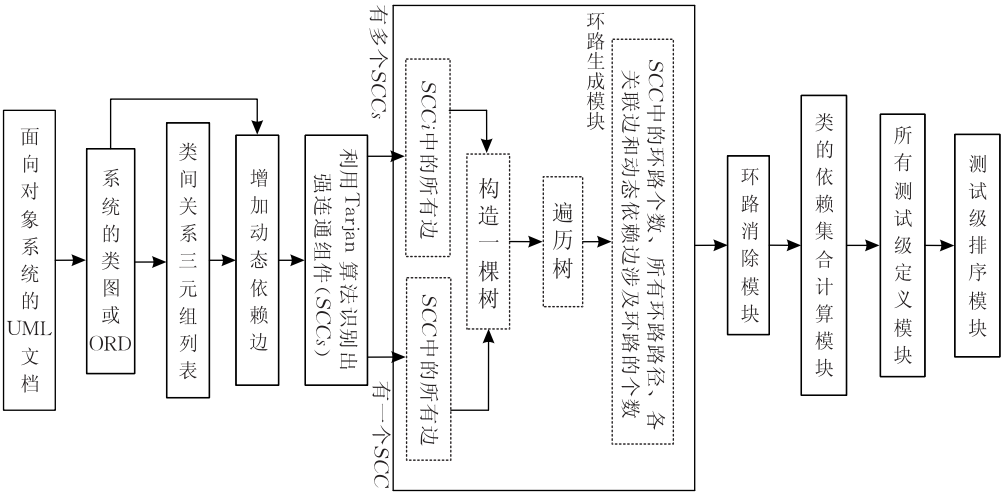


图 5 TLOG 功能结构图

该工具的输入信息是一个描述面向对象系统中类间关系的三元组列表,其中的三元组列表可以由面向对象系统的统一建模语言(UML)设计文档中的类图获得.下面介绍 TLOG 的几个主要功能模块.

(1)环路生成模块.如图 5 中虚线框所示,其输入信息是 UML 设计文档中的类图或 ORD 所包含的 *SCCs* 中的各条边,查找 *SCCs* 中所有的环路时,首先将每一个 *SCCi* 中的边以树的形式组织在一起,总共构造 i 棵树,每棵树的特点是叶节点与根节点相同,然后对所构造的树进行前序遍历,进而找出所有环路,最后自动输出 *SCCi* 中的环路数目及所有环路路径、*SCCi* 中各关联边和动态依赖边涉及的环路的数目.

(2)环路消除模块.首先对输入的三元组描述信息所表示的类图或 ORD,增加动态依赖关系,然后采用 Tarjan 等人[9]的算法,依次识别 *SCCi*,最后按照 3.2.2 节中的算法 1,通过断开一条或多条关联边或者动态依赖边,将每一个子 *SCCi* 的环路断开,直到消除所有环路.

(3)测试级排序模块:由 4.1 节的方法定义测试级以后,根据测试级的顺序分配的四条规则以及 4.2 节中的算法 2,得到基于测试级的类集成测试顺序.

TLOG 能够自动生成基于动态依赖关系的类集成测试序列,可减少测试的工作量.

5 实 验

在这一节中,我们将前面介绍的方法应用到一组基准程序中,通过实验验证本文方法的有效性.

5.1 实验描述和过程

在实验中,我们采用了在许多软件测试研究中被作为基准程序来使用的 3 个系统:(1) ANT 系统;(2) DNS 系统;(3) BCEL 系统.其中,ANT[5] 系统(http://jakarta.apache.org)包含 25 个类,83 条依赖边和 654 个依赖环路.DNS[5] 系统(http://www.xbill.org/dnsjava/)可以提供网络域名服务,包含 61 个类,276 条依赖边,其中,10 个类构成 16 个环路.BCEL[5] 系统(http://jakarta.apache.org/bcel/index.html)包含 45 个类,294 条依赖边,其中,41 个类构成 416091 个环路.实例的详细信息如

表 5 所示. 其中 2～8 列为仅考虑类间静态依赖关系时系统的统计信息，当考虑动态依赖关系时，类间关系更加错综，类间测试过程也变得复杂很多，表中最后一列是我们考虑动态依赖关系时统计的环路数目. 这 3 个系统的类图可以参照 Briand 等人的文献[5-6]. 对象关系图(ORD)的表示方法类似于类图. 类图比 ORD 多了 use 和 composition 关系，但由于 use 关系和 association 关系同为弱联系关系，因此在计算类间动态依赖关系时，可将 use 视为 association 关系，同理，composition 关系视为 aggregation 关系.

表 5　系统的详细信息

系统名	类的数目	use 边数	*As* 和 *Ag* 边数	composition 边数	inheritance 边数	静态边构成环路数	代码行数	所有边构成环路数
ANT	25	54	16	2	11	654	4093	8894
DNS	61	211	23	12	30	16	6710	16
BCEL	45	18	226	4	46	416091	3033	626826

5.2　实验过程

5.2.1　ANT 系统

为了说明 TLOG 工具的工作原理，我们给出实验的具体过程. 首先以 ANT 系统为例进行分析. 该系统的类图可以参见文献[5]，其中所包含的类如表 6 所示.

根据本文算法 1，当多条关联边的环路数目相同，删除其中任意一条关联边，导致测试顺序的不确定性，同时为了与已有方法[6,12]进行比较，我们在实验时运行 100 次. 表 7 首先给出我们的方法打破静态依赖关系构成的环路过程(此时假设不考虑动态依赖).

表 7 说明采用本文方法，如果只考虑类间静态依赖关系，打破环路共删除10条边. 因为需要为构成这 10 条边的源类创建测试桩，因此，打破 ORD 中所有环路需要为类 18，24，16，20，22 构造测试桩，总共需要构造 5 个测试桩.

表 6　ANT 系统

类编号	类	类编号	类
1	AntClassLoader	14	NoBannerLogger
2	BuildEvent	15	PathTokenizer
3	BuildException	16	Project
4	BuildListener	17	ProjectComponent
5	BuildLogger	18	ProjectHelper
6	DefaultLogger	19	RuntimeConfigurable
7	DemuxOutputStream	20	Target
8	DirectoryScanner	21	Task
9	FileScanner	22	TaskAdapter
10	IntrospectionHelper	23	TaskContainer
11	Launcher	24	UnknownElement
12	Local	25	XmlLogger
13	Main		

表 7　打破 ANT 系统静态依赖关系构成的环路过程

次序	处理的 *SCCi*	环路数目	处理 *SCCi* 删除的边
1	*SCC*{4,10,18,19,20,2,16,17,21,22,24,23}	654	19→18
2	*SCC*{4,10,18,20,2,16,17,21,22,24,23,19}	306	20→18
3	*SCC*{4,10,24,20,2,16,17,21,22,23,19}	135	20→24
4	*SCC*{4,16,2,17,21,20,22,23,19}	50	19→16
5	*SCC*{4,16,2,17,21,20,22,23}	33	17→16
6	*SCC*{4,16,2,20,21,22,23}	22	21→16
7	*SCC*{4,16,2,20,21,22,23}	12	20→16
8	*SCC*{4,16,2,22}&&*SCC*{20,21,23}	3&&2	2→16&&21→20
9	*SCC*{16,22}	1	16→22
10	处理结束	0	—

表 8　增加动态依赖关系后打破环路过程

次序	处理的 *SCCi*	环路数目	处理 *SCCi* 所删除的边
1	*SCC*{1,16,2,4,5,10,18,19,20,21,22,23,24,25,17}	8894	19→18
2	*SCC*{1,16,2,4,5,10,18,20,21,22,23,24,25,17,19}	4017	20→18
3	*SCC*{1,16,2,4,5,10,24,20,21,22,23,25,17,19}	1841	4→2
4	*SCC*{1,16,2,25,10,24,20,21,22,23,17,19}	897	16→4(16→1)(16→5)(16→10)(16→25)
5	*SCC*{16,2,10,24,20,21,22,23,17,19}	385	16→2
6	*SCC*{16,10,24,20,21,22,23,17,19}	204	16→21(16→24)
7	*SCC*{16,10,24,20,21,22,23,17,19}	100	20→24
8	*SCC*{16,10,24,23,20,21,22,17,19}	56	23→21(23→22)(23→24)
9	*SCC*{24,10}&&*SCC*{16,17,21,20,22,19}	1&&18	24→10&&16→22
10	*SCC*{16,17,21,20,22,19}	12	20→22
11	*SCC*{16,17,21,20,19}	7	17→16
12	*SCC*{16,19,20,21}	5	19→16
13	*SCC*{16,20,21}	3	20→16
14	*SCC*{16,21,20}	2	21→16
15	*SCC*{20,21}	1	21→20
16	处理结束	0	——

注：表 8 中最后一列括号中的边表示随关联边的删除而消失的动态依赖边.

本文考虑动态依赖关系,增加 20 个动态依赖关系,环路数目由 654 增至 8894. 表 8 给出了 *SCCs* 中环路的打破过程.

从表 8 可以看出,增加类间动态依赖关系时,打破 EORD 中环路共删除 23 条边,其中,随关联边的删除而消失的动态依赖边有 7 条. 由于需要为删除的这 23 条边的源类创建测试桩,因此,打破 EORD 中所有环路需要为类 18,2,4,1,5,10,25,21,24,22,16,20 构造测试桩,总共需要构造 12 个测试桩.

最后,利用本文的 TLOG 工具自动生成 ANT 系统的类测试顺序,共 13 个测试级,其中 10 个静态测试级(*S*),3 个动态测试级(*Dy*). 如表 9 所示. 其中:第 1 列为主测试顺序号,第 2 列为被测试的类,第 3 列为被测类所依赖的类,即在被测类之前进行测试的类. 第 4 列为被测类所属测试类型. 对于主测试顺序号相同的类,它们之间没有依赖关系,其测试顺序可以是任意的. 因此,表 9 中的测试顺序并不是唯一的测试顺序.

5.2.2 DNS 系统

对于 DNS 系统,我们采用同样的方法进行实验. 表 10 首先给出我们的方法打破静态依赖关系构成的环路过程(此时假设不考虑动态依赖).

表 9 ANT 系统基于测试级的测试顺序

主测试级	被测类 *X*	依赖的类 *SD*(*X*)	类型
1	4,9,12,15,17,23	—	*S*
2	3 5	12 4	*S*
3	8 19	3,12,9 3,12	*S*
4	21	3,12,17,19	*S*
5	20	3,12,19,21,17,23	*S*
6	16	3,12,15,17,20,19,21,23	*S*
7	1 2 7 10 22	3,12,4,16,15,17,20,19,21,23 16,3,12,15,17,20,19,21,23 16,3,12,15,17,20,19,21,23 3,12,4,16,15,17,20,19,21,23 3,12,16,15,17,20,19,21,23	*S*
8	6 11 24 25	2,16,3,12,15,17,20,19,21,23,5,4 1,3,12,4,16,15,17,20,19,21,23 3,12,16,15,17,20,19,21,23,22 2,16,3,12,15,17,20,19,21,23,4,25	*S*
9	14 18	2,16,3,12,15,17,20,19,21,23,6,5,4 3,12,10,4,16,15,17,20,19,21,23,22,24	*S*
10	13	3,12,4,5,16,15,17,20,19,21,23,18,10,22,24	*S*
11	2 10	16,3,12,15,17,20,19,21,23,22,24 3,12,4,16,15,17,20,19,21,23,22,24	*Dy*
12	6 25	2,16,3,12,15,17,20,19,21,23,22,24,5,4 2,16,3,12,15,17,20,19,21,23,22,24,4	*Dy*
13	13 14	3,12,4,5,16,15,17,20,19,21,23,18,10,22,24,1,25,2,6 2,16,3,12,15,17,20,19,21,23,22,24,6,5,4	*Dy*

表 10 打破 DNS 系统静态依赖关系构成的环路过程

次序	处理的 *SCCi*	环路数目	处理 *SCCi* 删除的边
1	*SCC*{33,38,52}&&*SCC*{8,11,21,25,32,48,58}	3&&13	33→52&&21→11
2	*SCC*{33,38}&&*SCC*{32,48,58} &&*SCC*{8,21}	1&&2&&1	33→38&&32→48&&21→8
3	*SCC*{32,58}	1	32→58
4	处理结束	0	—

表 10 说明采用本文方法,如果只考虑类间静态依赖关系时,打破环路共删除 6 条边. 打破 ORD 中所有环路需要为类 52,11,38,48,8,58 构造测试桩,总共需要构造 6 个测试桩.

本文考虑动态依赖关系,增加 175 个动态依赖关系,但环路数目不变. 表 11 给出了 *SCCs* 中环路的打破过程.

从表 11 可以看出,增加类间动态依赖关系时,打破 EORD 中环路共删除 6 条边. 打破 EORD 中所有环路需要为类 33,11,48,8,58 构造测试桩,总共需要构造 5 个测试桩.

针对 DNS 系统,利用本文的 TLOG 工具,我们最后得到 19 个测试级,其中 12 个静态测试级(*S*)、7 个动态测试级(*Dy*). 最终的具体测试顺序如表 12 所示.

表 11 增加动态依赖关系后打破环路过程

次序	处理的 *SCCi*	环路数目	处理 *SCCi* 删除的边
1	*SCC*{38,33,52}&&*SCC*{8,11,21,25,32,48,58}	3&&13	38→33&&21→11
2	*SCC*{52,33}&&*SCC*{32,48,58}&&*SCC*{8,21}	1&&2&&1	52→33&&32→48&&21→8
3	*SCC*{32,58}	1	32→58
4	处理结束	0	——

表 12 DNS 系统基于测试级的测试顺序

主测试级	待测类 *X*	类型	主测试级	待测类 *X*	类型
1	9,10,13,15,17,27,28,31,36,44,46,49	*S*	11	3,52	*S*
2	5,47,53	*S*	12	33	*S*
3	21	*S*	13	20,35,55	*Dy*
4	8,14,22	*S*	14	37,42,59,61	*Dy*
5	32	*S*	15	51,60	*Dy*

（续表）

主测试级	待测类 X	类型	主测试级	待测类 X	类型
6	1,2,4,6,16,18,19,23,25,26,29,34,39,40,43,45,48,56,58	S	16	38	Dy
7	7,11,20,24,30,35,41,54,55,57	S	17	52	Dy
8	37,42,59,61	S	18	12,33,50	Dy
9	51,60	S	19	3	Dy
10	12,38,50	S			

5.2.3　BCEL 系统

BCEL 系统包含 416091 个环路(不考虑动态依赖),由于篇幅有限,只简单给出采用本文方法打破静态依赖关系构成的环路过程,如表 13 所示.打破环路共删除 73 条边,总共需要构造 56 个测试桩.

考虑动态依赖关系后,增加 138 个动态依赖关系,环路数目增加至 626826 个.表 14 简单给出了 $SCCs$ 中环路的打破过程.打破 EORD 中环路共删除 87 条边,总共需要构造 70 个测试桩.

针对 BCEL 系统,利用本文的 TLOG 工具,我们最后得到 17 个测试级,其中 10 个静态测试级(S)、7 个动态测试级(Dy).最终的具体测试顺序如表 15 所示.

表 13　打破 BCEL 系统静态依赖关系构成的环路过程

次序	处理的 $SCCi$	环路数目	处理 $SCCi$ 删除的边
1	SCC{2,4,35,30,45,5,6,7,18,8,10,13,21,26,15,9,11,12,14,16,17,19,22,25,27,28,29,32,33,34,37,38,39,40,20,31,36,41,43,44}	416091	45→30
2	SCC{2,4,35,45,5,6,7,18,8,10,13,21,26,15,9,11,12,14,16,17,19,22,25,27,28,29,32,33,34,37,38,39,40,20,31,36,41,43,44}	380735	45→35
3	SCC{2,4,45,5,6,7,18,8,10,13,21,26,15,9,11,12,14,16,17,19,22,25,27,28,29,32,33,34,37,38,39,40,20,31,36,41,43,44}	244823	2→21
…	…	…	…
73	SCC{2,44}	1	2→44
74	处理结束	0	—

表 14　增加动态依赖关系后打破环路过程

次序	处理的 $SCCi$	环路数目	处理 $SCCi$ 删除的边
1	SCC{2,3,4,35,30,45,5,6,7,9,18,8,10,13,21,26,15,9,11,12,14,16,17,19,20,22,25,27,28,29,32,33,34,37,38,39,40,20,31,36,41,43,44}	626826	45→26
2	SCC{2,3,4,35,30,45,5,6,7,9,18,8,10,13,21,15,9,11,12,14,16,17,19,20,22,25,27,28,29,32,33,34,37,38,39,40,20,31,36,41,43,44}	587032	34→33
3	SCC{2,3,4,35,30,45,5,6,7,9,18,8,10,13,21,15,9,11,12,14,16,17,19,20,22,25,27,28,29,32,34,37,38,39,40,20,31,36,41,43,44}	523401	45→4
…	…	…	…
87	SCC{2,44}	1	2→44
88	处理结束	0	—

表 15　BCEL 系统基于测试级的测试顺序

主测试级	待测类 X	类型	主测试级	待测类 X	类型
1	1,17,36,42,45	S	10	3,23,24	S
2	2,6,31,41	S	11	4,21,22,25,29,32,34,37,38,39,40,43,44	Dy
3	8,9,11,12,14,16,20,22,25,32,34,39,40,43,44	S	12	3,23,26,27,35	Dy
4	10,15	S	13	8,9,11,12,14,16,19,20,33	Dy
5	18	S	14	18,21	Dy
6	5,7,19,27,28,33,37,38	S	15	10,13,15	Dy
7	4,21,29	S	16	23	Dy
8	26,35	S	17	2,5,6,7,28,30,31,36,41	Dy
9	30	S			

5.3　实验结果及分析

这一节中,我们就打破环路所需构造测试桩的数目,分别与文献[6]中 Briand、Traon 和 Tai 等人的 3 种只考虑静态依赖关系的测试方法、文献[13]中 Li 的增加动态依赖关系的类集成测试方法进行比较,比较结果如表 16～18 所示.其中横坐标表示在算法执行中所花费测试桩的个数,纵坐标表示在 100 次算法执行中与测试桩的个数相对应的次数.

表 16 ANT 系统结果

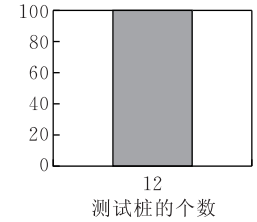
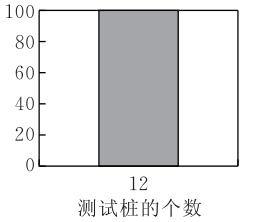
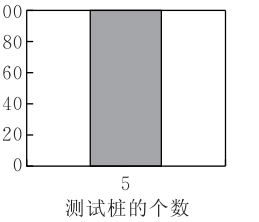
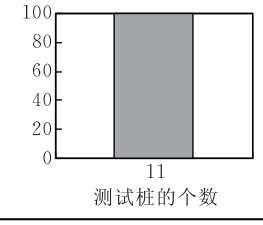
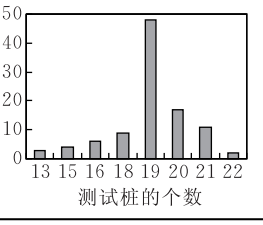
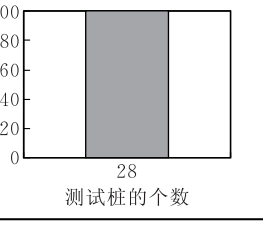
	本文方法(增加动态依赖)	Li 的方法[13](增加动态依赖)	本文方法(仅考虑静态依赖)
算法执行次数			
	Briand 等人的方法[6](静态依赖)	Le 等人的方法[4](静态依赖)	Tai 等人的方法[3](静态依赖)
算法执行次数			

表 17 DNS 系统结果

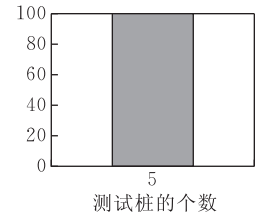
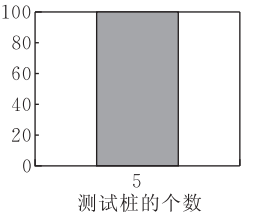
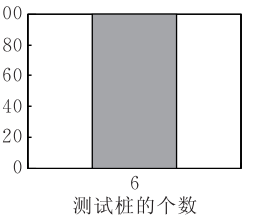
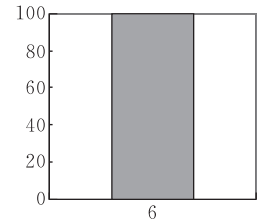
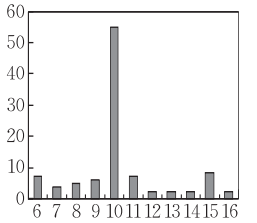
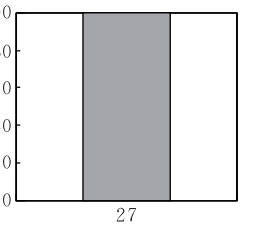
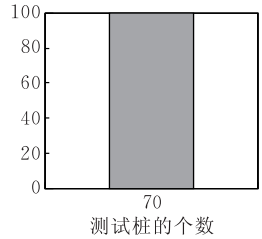
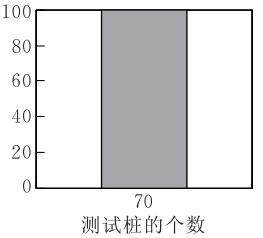
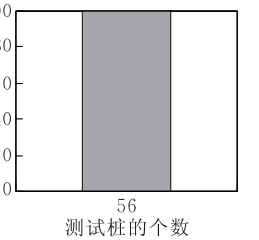
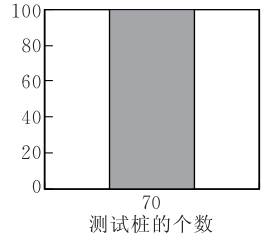
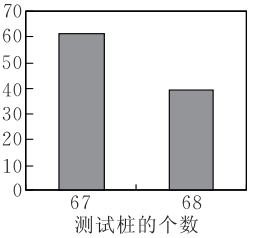
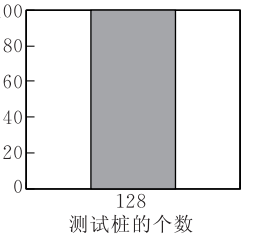
	本文方法(增加动态依赖)	Li 的方法[13](增加动态依赖)	本文方法(仅考虑静态依赖)
算法执行次数			
	Briand 等人的方法[6](静态依赖)	Le 等人的方法[4](静态依赖)	Tai 等人的方法[3](静态依赖)
算法执行次数			

表 18 BCEL 系统结果

	本文方法(增加动态依赖)	Li 的方法[13](增加动态依赖)	本文方法(仅考虑静态依赖)
算法执行次数			
	Briand 等人的方法[6](静态依赖)	Le 等人的方法[4](静态依赖)	Tai 等人的方法[3](静态依赖)
算法执行次数			

通过表16～18中的结果可以发现：对于ANT系统，如果仅考虑类间的静态依赖关系，采用本文方法执行算法100次，均需要构造5个测试桩，而采用Briand等人[6]、Le等人[4]和Tai等人[3]的3种方法均构造多于10个测试桩；如果增加类间的动态依赖关系，采用本文方法需要构造12个测试桩，Li的方法[13]同样需构造12个测试桩. 对于DNS系统，如果仅考虑类间的静态依赖关系，采用本文方法需要构造6个测试桩，与采用Briand等人[6]的方法构造相同个数的测试桩，而采用Le等人[4]和Tai等人[3]的3种方法均构造多于6个测试桩；如果增加类间的动态依赖关系，采用本文方法需要构造5个测试桩，Li的方法[13]同样需构造5个测试桩. 对于BCEL系统，如果仅考虑类间的静态依赖关系，采用本文方法需要构造56个测试桩，而采用Briand等人[6]、Le等人[4]和Tai等人[3]的3种方法均构造多于67个测试桩；如果增加类间的动态依赖关系，采用本文方法需要构造70个测试桩，Li的方法[13]同样需构造70个测试桩.

实验结果证明：从这两个方面与已有的集成测试方法相比，如果仅考虑类间的静态依赖关系，本文方法与Briand、Le和Tai等人的3种方法相比，构造的测试桩的个数最少；如果增加类间的动态依赖关系，采用本文方法需要构造的测试桩个数与采用Li的方法[13]相等，但是，他的方法存在缺陷：允许断开继承边和聚集边等强联系边来打破环路，导致测试桩复杂度的提高，因此，如果在不允许断开强联系边的情况下，Li的方法将会大大增加测试桩的个数. 因此，体现了本文方法的有效性.

6　相关工作讨论

现有的类间测试顺序研究方法大多仅限于静态分析：Kung等人[1]最早提出解决类间测试顺序问题的方法，并证明如果对象关系图中没有环，则可以通过逆向拓扑排序得到类间测试顺序. 如果类图中有环，则首先识别其中的强联通分量，然后删除部分关联边使之成为无环图. Tai等人[3]提出的测试顺序分配策略将ORD中的3种关系分为两个层次：继承关系和聚集关系位于一个层次，关联关系处于一个层次. 当穿越主层的关联关系没有形成环路时，该策略导致构造不必要的测试桩. Le等人[4]采用了一种基于测试依赖图模型的方法进行集成测试，测试依赖图是由类和方法之间的测试依赖关系构成的，他们的策略减少了桩的数量，但可能断开继承和聚集关系. 之后，Briand等人[6]在不打破继承、聚集等强联系关系的前提下，给出基于图论的测试顺序策略，通过最小化测试桩的数目找到一个最佳测试顺序，与Le和Tai的方法相比，所需测试桩数目最少. Jaroenpiboonkit等人[12]提出了一种使用测试依赖图和面向对象切片技术找到一个满足最小化测试桩的数目的最佳测试顺序的方法. 以上方法均未考虑类间的动态依赖关系.

考虑类间的动态依赖关系的文献相对较少：Labiche等人[7]给出了基于测试级的类间测试序列生成方法，Kraft等人[2]和Paradkar[14]在类集成测试中考虑了动态依赖关系，但是以上4种方法均没有考虑动态依赖关系对构成的环路的影响，因此也没有给出相关的边的删除规则；就我们所知，目前只有Li等人[13]提出了考虑动态依赖关系时的环路中边的删除规则，解决了忽略类间动态依赖关系所导致的测试桩的数目不足以完成测试的问题，但是他的方法允许断开聚集依赖关系，引起构造复杂测试桩的问题，并且在某些情况下可能产生多余测试桩.

将本文方法与已有的算法所得的类测试顺序进行比较，可以得出如下结论：

(1) 动态测试级的引入是本文的一大特点. 我们的方法解决了测试桩的数量不能满足被测类充分测试的问题.

(2) 本文对存在环路的类簇进行集成测试，打破环路时仅允许删除关联边和动态依赖边，遵循以最少的边打破更多环路的原则，花费较低的测试代价.

(3) 本文的测试级表示方法可以获得更多的信息量，能够更加清晰、充分地表示类簇的测试序列. 其中，相同测试级中的类的测试顺序可以交换. 该方法不仅能够灵活体现出各测试级的序列，也能够反映出当测试某一个类时，该类所依赖的类以及一个测试级相对于上一级所增加的类. 因此，根据测试级可以得到一种增量测试，能够达到测试用例的高度重用，而测试用例的重用使派生类测试中所需重新设计的测试用例减少，加快测试的进度.

7　结束语

类间测试顺序的确定是类间集成测试中最重要的问题. 测试桩的复杂性标准目前国内外大多只针对类间静态依赖关系进行分析，而忽略了类间动态依赖关系，虽然一些相关研究中已经考虑类间动态

依赖关系,但是并没有进行定量分析,因此基于类间动态依赖关系建立测试桩的方法是一个新的研究领域.为了解决已有的方法忽略动态依赖关系的问题,为测试提供足够的测试桩,本文不仅分析了类之间的静态依赖关系,更重点分析了类间的动态依赖关系,并考虑了动态依赖边对消除环路的影响,给出了环路消除算法.由于本文增加了动态依赖关系的分析,对于动态依赖关系的测试桩的复杂性标准难以确定,因此,我们从构造测试桩数目的角度与已有方法进行比较,结果证明本文方法能够以相对较少的测试桩满足类的静态测试和动态测试.

对于大型面向对象程序来说,被测程序中类的数量很多,工作量相当庞大.在此基础上我们开发了测试级生成器 TLOG,能够自动生成基于测试级的类集成测试序列,减少测试的工作量,提高测试效率.

本文通过最小化测试桩的数量来确定类间测试顺序,一方面是由于目前针对动态依赖边的复杂度还没有一个度量标准;另一方面是由于现有已考虑动态依赖边的文献中,均采用构造测试桩的数量作为衡量标准,而没有从测试桩复杂度的角度进行研究,为了与已有的方法进行比较,我们也将构造测试桩的数量作为衡量标准.在满足最小化测试桩复杂度的前提下,确定包含动态依赖关系的类间测试顺序,作为我们下一步要研究的内容之一.

此外,可以发现,本文中我们没有考虑抽象类的特点,实际上,抽象类的特性,将会影响类间的依赖性,进而将影响类间测试序.这是我们目前正在解决的主要问题之一.

致谢 在此,我们向对本文给予建议的同行表示感谢.同时,对审稿人提出的有益建议表示感谢.

参 考 文 献

[1] Kung D C, Gao J, Hsia P. Class firewall test order and regression testing of object oriented programs. Journal of Object-Oriented Programming, 1995, 8(2): 51-65

[2] Kraft N A, Lloyd E L, Malloy B A, Clarke P J. The implementation of an extensible system for comparison and visualization of class ordering methodologies. Journal of Systems and Software, 2006, 79(8): 1092-1109

[3] Tai K C, Daniels F. Test order for inter-class integration testing of object-oriented software//Proceedings of the 21st International Computer Software and Applications Conference. Washington, DC, USA, 1997: 602-607

[4] Le Traon Y, Jéron T, Jézéquel J M, Morel P. Efficient object-oriented integration and regression testing. IEEE Transactions on Reliability, 2000, 49(1): 12-25

[5] Briand L, Feng J, Labiche Y. Experimenting with genetic algorithms to devise optimal integration test orders. Carleton University, Technical Report SCE-02-03, 2002

[6] Briand L C, Labiche Y, Wang Y. An investigation of graph-based class integration test order strategies. IEEE Transactions on Software Engineering, 2003, 29(7): 594-607

[7] Labiche Y, Thévenod-Fosse P, Waeselynck H, Durand M H. Testing levels for object-oriented software//Proceedings of the 22nd International Conference on Software Engineering. Limerick, Ireland, 2000: 136-145

[8] Briand L, Labiche Y, Wang Y. Revisiting strategies for ordering class integration testing in the presence of dependency cycles//Proceedings of the 12th International Symposium on Software Reliability Engineering. Hong Kong, China, 2001: 287-297

[9] Tarjan R. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1972, 1(2): 146-160

[10] Labiche Y. Incremental class testing from a class test order. Carleton University, Technical Report SCE-05-06, 2005

[11] Qi Li-Na. Research and application of class integration testing strategy based on test order[Ph. D. dissertation]. Shanghai Normal University, Shanghai, 2007(in Chinese)
(齐丽娜. 基于测试顺序的类集成测试方法研究与应用[博士学位论文]. 上海师范大学, 上海, 2007)

[12] Jaroenpiboonkit J, Suwannasart T. Finding a test order using object-oriented slicing technique//Proceedings of the 14th Asia-Pacific Software Engineering Conference. Nagoya, Japan, 2007: 49-56

[13] Li Du. Towards test order selection strategy. Computer Engineering and Design, 2008, 29(4): 781-783(in Chinese)
(李都. 测试顺序选择策略研究. 计算机工程与设计, 2008, 29(4): 781-783)

[14] Paradkar A M. Inter-class testing of O-O software in the presence of polymorphism//Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research. Toronto, Ontario, Canada, 1996: 137- 146

[15] Abdurazik A, Offutt A J. Using coupling-based weights for the class integration and test order problem. The Computer Journal, 2009, 52(5): 557-570

[16] Wang Zheng-Shan, Li Bi-Xin. Using coupling measure technique and random iterative algorithm for inter-class integration test order problem//Proceedings of the 34th Annual IEEE Computer Software and Applications Conference Workshops. Seoul, Korea, 2010: 329-334

[17] Malloy B A, Clarke P J, Lloyd E L. A parameterized cost model to order classes for class-based testing of C++ applications//Proceedings of the 14th International Symposium on Software Reliability Engineering. Denver Colorado, USA, 2003: 353-364

ZHANG Yan-Mei, born in 1982, Ph. D. candidate. Her research interests include software analysis and testing, exception handling.

JIANG Shu-Juan, born in 1966, professor, Ph. D. supervisor. Her research interests include compilation techniques, software engineering.

ZHANG Hong-Chang, born in 1989, M. S. candidate. His research interests include software analysis and testing.

Background

Class integration testing is an important part in object-oriented software testing, and the determination of class order is a key and difficult problem of class integration testing. An appropriate test order for software testing can reduce test cost.

For the class test order problem, the existing solutions mostly focused on the static analysis, but inter-class dynamic dependency is universal. Therefore, neglecting dynamic dependencies tends to make the shortage of test stubs, causing the test insufficiency.

So far, there are some scholars that consider the inter-class dynamic dependency, but they just considered simply ordering of classes without involve cyclic dependency calls. Although there are few studies that involved cycles and gave rules of removing edges, taking account of dynamic dependencies, inheritance and aggregation dependencies are allowed to be removed. Thus this significantly increases the redundant stubs in some situations.

The goal of the authors' study is to devise an optimal order with the minimum number of stubs. This can solve the problem that the stubs are insufficient for completing test led by ignoring dynamic dependencies.

The main contribution of this work is: under the premise of minimizing the number of test stubs, propose several rules of removing edges, taking account of dynamic dependencies. The advantage of the proposed method is that the test order is novel, which lies in the fact that it is the test order after eliminating the cycles that formed by static and dynamic dependency.

This work was supported in part by awards from National Natural Science Foundation of China under Grant No. 60970032, the Key Project of Chinese Ministry of Education under 108063, Natural Science Foundation of Jiangsu Province under Grant No. BK2008124, Qing Lan Project, and Graduate Training Innovative Projects Foundation of Jiangsu, China under Grant No. CX10B_157Z.