一个基于UML顺序图的场景测试用例生成方法*)

沈剑乐　王林章　李宣东　郑国梁

(南京大学计算机科学与技术系　南京210093)

摘　要　UML顺序图是基于UML开发的软件设计模型的重要组成部分,它描述了软件系统的动态行为,是软件集成测试过程中的一个重要的信息来源。本文提出了一个基于UML顺序图的场景测试方法,它以UML顺序图为主要测试模型,结合UML状态图和类图生成所有的测试场景,最后使用范畴-划分方法找到与每一场景相关的环境条件并将它与方法序列、输入、输出合理组合作为覆盖该场景的测试用例,用于测试该场景中对象之间的交互。由于UML已广泛用于软件分析和设计阶段,通过UML模型生成测试用例可充分利用已有的设计结果,减少测试阶段所需的费用,对于已使用UML的工业界有着重要的意义。

关键词　测试用例生成,UML顺序图,场景测试,面向对象,集成测试

An Approach to Generate Scenario Test Cases Based on UML Sequence Diagrams

SHEN Jian-Le　WANG Lin-Zhang　LI Xuan-Dong　ZHENG Guo-Liang

(Department of Computer Science and Technology, Nanjing University, Nanjing 210093)

Abstract　UML sequence diagram is an important component of software design model based on UML. Because it describes the dynamic behaviors of a software system, it becomes a valuable information source in software integration test. This paper proposes an approach to generate scenario test cases based on UML sequence diagrams. It takes UML sequence diagrams as its main test model and generates all test scenarios combining UML state diagrams and class diagrams, then it applies category partition method to find environmental settings related to each scenario and combines them with method call sequence, as well as input and output, to form a test case for each scenario, thus we can test the interactive behavior of the software. Because UML has been widely used in software analysis and design phrases, using UML to generate test cases can make the most of the existing design results and reduce the expenses of test phrase. Accordingly, it is significant to industry in which UML has been used.

Keywords　Test cases generation, UML sequence diagram, Scenario test, Object-oriented, Integration test

1　引言

软件测试是软件工程中一个重要的组成部分,软件测试的过程可分为单元测试、集成测试、确认测试和系统测试四个阶段。单元测试着重于软件以源代码形式实现的各个单元;集成测试着重于对软件的体系结构的设计和构造;确认测试以软件需求为依据,对已经建造好的系统进行验证;系统测试则是对软件与硬件和其他相关因素的功能测试。

集成测试是整个测试过程中的重要一环,集成测试的目标是验证系统的行为以及发现由于模块的接口所带来的错误和新单元加入所导致的副作用。传统的集成测试策略分为增量集成和非增量集成两种方式。增量集成方式先将程序分成小的部分进行构造和测试,然后可分别选择自顶向下或自底向上的集成策略将小的程序模块逐步集成;非增量方式正好相反,它使用一步到位的方法来构造程序,所有的模块都预先结合在一起,整个程序作为一个整体来进行测试。

当考虑到面向对象(OO)软件的集成测试时,由于面向对象软件没有层次的控制结构,传统的自顶向下和自顶向上集成策略就没有意义。此外,由于构成类的成分的直接和间接的交互,一次集成一个操作到类中(传统的增量集成方法)经常是不可能的。

对OO软件的集成测试有两种不同的策略,第一种称为基于线程的测试(thread-based testing),一个线程对应于系统的一个输入或事件所引发的一个执行序列,每个线程执行所需的一组类被集成并进行了测试,本文中的场景测试就是一种基于线程的测试。第二种称为基于使用的测试(use-based testing),通过测试那些几乎不使用服务器类的类(称为独立类)而开始构造系统,在独立类测试完成后,下一层的使用独立类的类,称为依赖类,被测试。这个依赖类层次的测试序列一直持续到构造出完整的系统[1]。

UML是面向对象系统分析、设计的标准的建模语言,它使用一系列的视图来描述被建模系统的各个方面,例如,使用用况图捕捉用户需求,使用类图捕捉对象的静态结构,使用顺序图和协作图捕捉对象和系统间的动态交互,并用包和实施视图来组织各种设计元素。

UML用况图由一组用况组成。一个用况捕捉了系统的一个完整的功能需求,描述了实现该功能的一组动作序列,其中的每一个序列被称作一个场景(scenario)。场景是一个表示行为的特定动作序列,是用况的一个实例,系统通过场景的执行来为参与者产生一个可观察的结果值[2]。用况的场景可以用一个或多个顺序图来描述,在顺序图中,场景被定义为在相互交互的对象间传递的一个消息序列,每一个消息序列代表该用况的一个可能的事件流,在一个顺序图中可包含多个场景(通过分支和循环结构来实现)。

本文提出了一个通过软件设计阶段的UML图模型来测试场景的方法,它以顺序图为主要测试模型,从中导出所有的

*)本课题研究得到国家863高科技项目(2002AA116090),国家自然基金(6027036),江苏省自然基金(BK2002079)的资助。**沈剑乐**　硕士研究生,研究方向为面向对象的软件测试。**王林章**　博士研究生,研究方向为软件测试。**李宣东**　教授,博导,主要研究方向为面向对象技术,形式化方法和模型检验。**郑国梁**　教授,博导,研究方向是软件工程,形式化方法。

场景，并使用著名的范畴-划分方法[2,3]最终生成所有的场景测试用例。下面第2部分介绍了部分 UML 图的基本概念，第3部分给出了本测试方法所遵循的测试衡量标准（test-criteria），第4部分结合一个实例给出了方法的详细步骤，最后是总结和今后的研究工作。

2 UML 图

2.1 类图

类图[4,5]反映了一个系统的静态结构，它通常由类和类间关系构成，类包含属性和方法，类间关系包括关联、泛化和依赖。图1是本文后面课程注册系统中的一个类图，它包含 CourseOffering 和 Schedule 两个类，CourseOffering 类具有属性 numStudent、hasProfessor 和方法 closeRegistration、getStudentNum；Schedule 类有 commit 和 cancel 方法；两个类间的实线箭头表示一个关联关系。

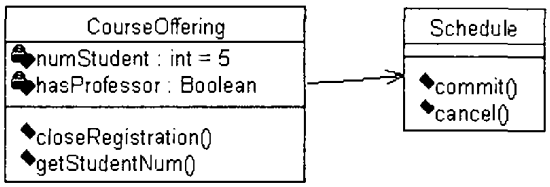

图1 课程注册系统的部分类图

2.2 顺序图

顺序图[4,5]是交互图的一种形式，它显示了一个交互并强调消息的时间顺序。在图形上，顺序图有两个轴，其中水平轴表示参与交互的各个对象，垂直轴表示时间。顺序图中的对象用带有垂直线的矩形框表示，矩形框内标有类名、对象名或角色名。垂直线称为对象的生命线，代表在对象之间的交互作用中该对象的生命周期。对象间的通信消息通过对象生命线之间的箭头表示，箭头上方标注消息名和一些控制信息。箭头分为实线箭头和虚线箭头两种，实线箭头表示一个实际的被触发的方法，虚线箭头仅表示一个方法的返回。图2是课程注册例子中所用到的顺序图，其中 Register 代表一个外部的参与者，它触发了顺序图中的消息序列的执行，controller 和 schedule 和 offering 是对象名，aOffering 是角色名。

在图2中，各个消息名前有一个序号，用来代替具体的消息名，这是为了下面叙述的方便，与顺序图本身无关。消息3和消息4构成了一个分支，其分支条件由消息名前的条件子句指明。消息4、5、6、7或4、5、8、9（6、7与8、9同样构成了一个分支）处于一个循环之中，图中用了一个标签对其作了说明，消息名4和5前指出了循环条件子句。

为了能在测试中寻找出所有的场景，下面给出顺序图的形式化定义：

定义1（顺序图） UML 中的顺序图 SD 可以表示为一个五元组：$SD=\langle O,M,E,\rightarrow,obj\rangle$。其中：

- $O=\{O_1,O_2,\cdots,O_m\}$，是对象的集合。$O_1,O_2,\cdots,O_m$ 都是顺序图中的对象。
- $M\subseteq$guard×message_name×parameter_list，是消息的集合。顺序图中的每一个消息都形如："[卫式条件]消息名(参数)"。

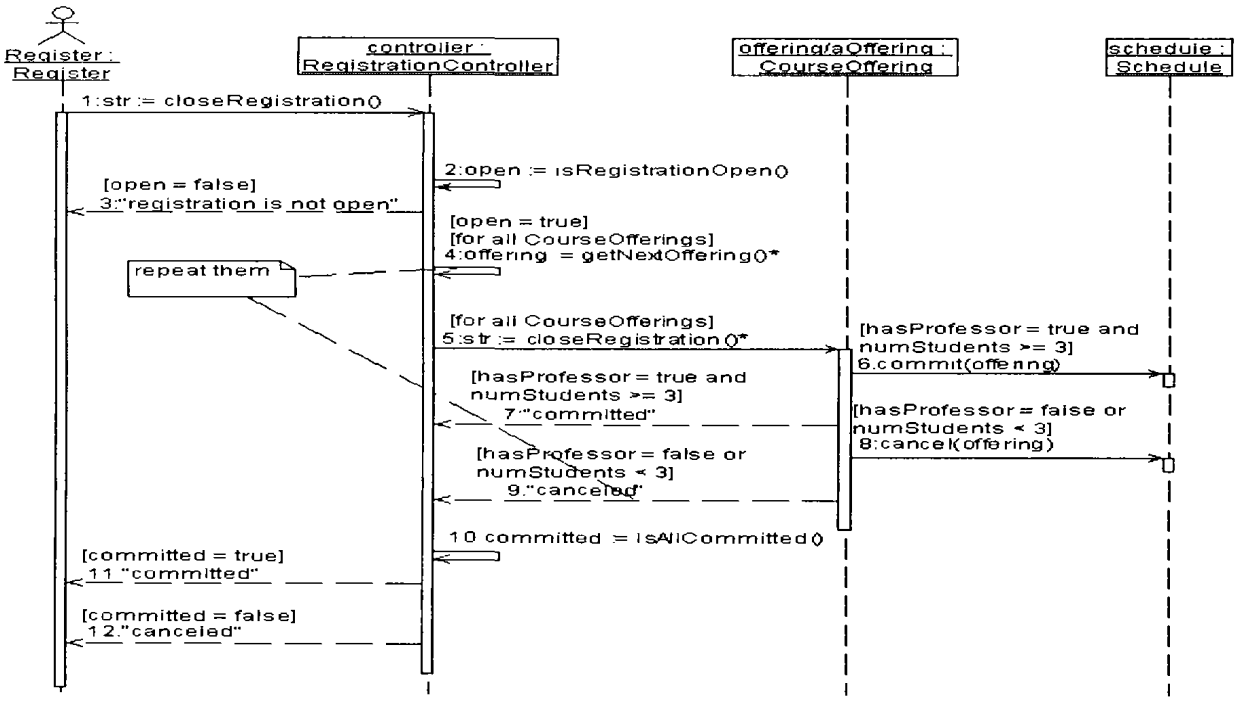

图2 课程注册顺序图

- $E=M\times\{s,r\}$，是事件集合。事件是指消息的发送和接收。对于消息 msg，发送事件用〈msg,s〉表示，接收事件用〈msg,r〉表示。顺序图中所有发送消息事件的集合记为 S，所有接收消息事件的集合记为 R。$S\cap R=\Phi$，$S\cup R=E$。
- $\rightarrow$是消息集合 M 上的一个全序关系，表示顺序图中的消息在纵向时间轴上的先后关系。
- obj 是从 E 到 O 的一个函数关系，$obj(e)\in O$，表示事件 e 所对应的对象。对象 O_i 上所有事件的集合记为 E_i，$E_i=\{e\mid e\in E\wedge obj(e)=O_i\}$。

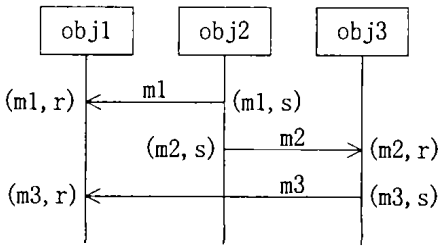

图3 一个简单的顺序图

在图3所示的顺序图中：

$O=\{obj1,obj2,obj3\}$；$M=\{m1,m2,m3\}$；

$E=\{\langle m1,s\rangle,\langle m1,r\rangle,\langle m2,s\rangle,\langle m2,r\rangle,\langle m3,s\rangle,\langle m3,r\rangle,\}$；

$\rightarrow=m1\rightarrow m2\rightarrow m3$。

顺序图主要描述了消息间的前后顺序关系，也就是发送消息的事件和接收消息的事件之间的时间顺序。我们用符号'≪'来两个表示事件之间的先后关系，它满足如下三个性质：

1)因果性：对同一消息而言，发送事件先于接收事件。

2)可控性：在同一个对象的生命线上，如果事件 $e1$ 出现在发送事件 $e2$ 的上方，则 $e1$ 先于 $e2$。

3)队列性($FIFO$)：在同一个对象的生命线上，如果接收事件 $e1$ 出现在接收事件 $e2$ 的上方，并且它们分别对应的发送事件也位于同一个对象的生命线上，则 $e1$ 先于 $e2$。

'≪'顺序关系是系统必须满足的根本时间顺序，如果两个事件之间不存在'≪'关系，那么即使它们在顺序图的时间轴上存在先后关系，这两个事件实际发生的先后顺序也是不确定的。例如在图3中，尽管$(m1,r)$在$(m3,r)$的上方，但是在系统的实际运行中由于 $m1,m2,m3$ 都是异步消息，因此$(m1,r)$并不一定先于$(m3,r)$发生，可能是消息 $m1$ 传递所需的时间大于消息 $m2$ 和 $m3$ 传递所需的时间之和，这是由于图形表示的局限性造成的。

一个简单顺序图(不包括分支和循环)刻画了系统运行的一个场景，其运行过程表现为一个事件的序列$\langle e_1,e_2,\cdots,e_m\rangle$，其中事件 e_{i+1} 在事件 e_i 之后发生$(1\leqslant i\leqslant m-1)$。由于事件之间存在顺序关系≪，因此并不是所有的事件序列都是顺序图允许的。同时，由于≪并不是一个全序关系，因此一个顺序图的场景可能允许多个事件序列。

定义2(顺序图的语义)　对于简单顺序图 $SD=\langle O,M,E,\rightarrow,obj\rangle$，事件序列$\langle e_1,e_2,\cdots,e_m\rangle$是一个有效的序列当且仅当以下两个条件得到满足：

(1)所有 E 中的事件在序列中出现且仅出现一次。也就是说$\{e_1,e_2,\cdots,e_m\}=E$ 且对于所有的 $i\neq j,e_i\neq e_j$；

(2)对于任意两个事件 $e_i,e_j\in E$，如果 $e_i\ll e_j$，那么在序列中 e_i 在 e_j 的前面。

我们可以用一个有向无环图(DAG)来表示'≪'关系，对于任意两个事件 $e_i,e_j\in E$，如果 $e_i\ll e_j$，则从 e_i 画一条指向 e_j 的边直到所有事件都在这个有向图上。通过对 DAG 图的遍历，我们可以很容易地得出顺序图中的每一个有效的事件序列。在图3的例子中，$\langle(m1,s),(m2,s),(m2,r),(m3,s),(m3,r),(m1,r)\rangle$和$\langle(m1,s),(m1,r),(m2,s),(m2,r),(m3,s),(m3,r)\rangle$就是两个有效的事件序列。

定义3(顺序图的场景)　对于简单顺序图 $SD=\langle O,M,E,\rightarrow,obj\rangle$，场景定义为一个消息序列$\langle M1,M2,\cdots,Mm\rangle$并且满足以下两个条件：

(1)所有 M 中的事件在序列中出现且仅出现一次。也就是说$\{M1,M2,\cdots,Mm\}=M$ 且对于所有的 $i\neq j,Mi\neq Mj$；

(2)对于任意两个消息 $Mi,Mj\in M$，如果$\langle Mi,s\rangle\ll\langle Mj,s\rangle$，那么在序列中 Mi 在 Mj 的前面。

根据场景的定义，场景中的各个消息的顺序是由发送事件的顺序所决定的，通过在一个合法的事件序列中找出发送事件序列就可以确定与该事件序列相应的场景。因此在遍历出所有的事件序列后我们就可以导出顺序图中的所有场景。在图3中，$\langle m1,m2,m3\rangle$就是一个有效场景并且是唯一的一个。

在一个复杂的顺序图(如图2)中，由于分支和循环的存在，可能包含多个场景，因此对场景的生成方法要做少许的修改，在下面会对此进一步说明。

2.3　状态图

状态图[4-5]描述了一个对象的行为，它说明对象在它的生命期中响应事件所经历的状态序列以及它们对那些事件的响应。状态图通常包括两个部分：状态和转换。

一个状态是指在对象的生命期中的一个条件或状况，在此期间对象将满足某些条件、执行某些活动或等待某些事件。状态可进一步分为简单状态和组合状态，组合状态可包括多个不相交或并发的子状态。

转换是两个状态之间的一种关系，表示对象将在第一个状态中执行一定的动作，并在某个特定事件发生和某个特定的条件(监护条件)满足时进入第二个状态。当状态发生这样的转变时，转换被称作激活了。在图形上，一个转换用一条从原状态到目标状态的有向实线来表示，在实线的上方标明相应的事件、监护条件和动作，其格式为：事件[监护条件]/动作，其中的每一项都是可选的。

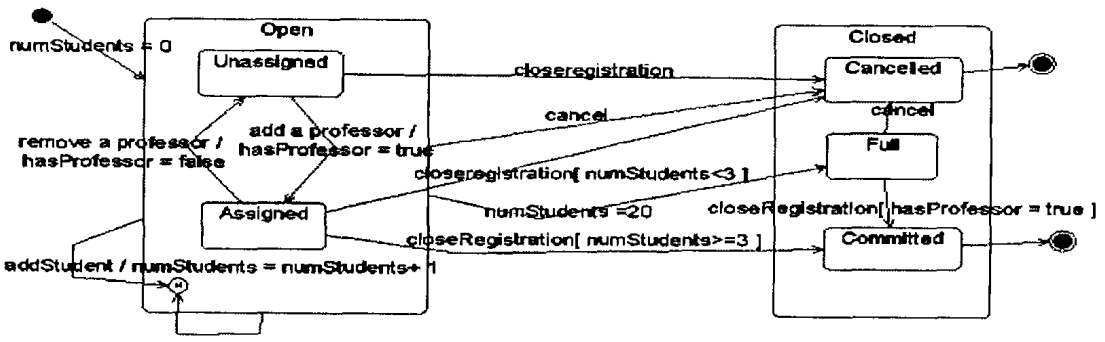

图4　CourseOffering 类的状态图

图4是课程注册系统中 CourseOffering 类的状态图，其中 Open、Closed 为组合状态，Unassigned、Assigned、Cancelled、Full 和 Committed 为简单状态(子状态)。各状态间的连线表明了相应的转换。

3　测试衡量标准

软件测试中的一个重要的问题是决定何时停止测试。这是因为被测试的软件域通常都是非常庞大的，因此无法对软件进行穷尽测试。为此，软件测试人员通过定义一个测试衡量标准来决定软件是否经过了充分的测试。

测试衡量标准[6,7]是一系列反映测试需求的规则的集合，测试需求指明了一个软件被测试的时候需要被覆盖的部分。如果一个软件经过测试后满足了测试衡量标准，就可以认为软件经过了充分的测试，可以通过测试覆盖率来评估测试衡

量标准被满足的程度。

对于顺序图的场景测试，最基本的测试衡量标准就是要求针对每一个场景都生成一个测试用例。但是顺序图本身并不能充分地对系统的交互行为进行建模[4]，仅通过顺序图并不能生成充分的场景测试用例。以图2中的部分场景(5,6)为例，通过对图4中 CourseOffering 类的状态图的分析可以知道，在执行这一场景之前 offering 对象可以处于 Assigned 和 Full 两个状态中的任何一个，它们反映了该场景执行时不同的上下文环境，因此该场景在每个状态下都应该被单独测试一次。由此可以看出，通过将顺序图与状态图结合，可以生成更为充分的测试用例，为此，本文定义了如下的测试衡量标准：

1)顺序图中的每个场景至少要被测试一次。

2)如果顺序图中的对象存在状态图，那么与场景相关的每个状态至少要被测试一次。

在顺序图中，可以将对象的状态分为两大类：原始态和结果态。原始态是指在场景执行之前各个对象所处的状态；结果态是指在场景执行的过程中和执行之后，各个对象所处的状态，它们都由相应的原始态转换而来。因为各个结果态是由相应的原始态和场景决定的，所以在把状态图中的状态信息加入顺序图中的时候(下面将要详述)，只要考虑各对象在顺序图中的原始态即可。

4 基于 UML 顺序图生成场景测试用例的方法

4.1 测试方法概述

顺序图的场景测试是一种基于线程的集成测试方法。由于各种编码的错误，顺序图中的场景设计往往与实现不一致，例如由于消息名的编码错误、错误的参数、不正确的参数值、不正确的或缺少输出以及非预期的运行时绑定导致的错误方法调用。如果顺序图的实现中存在错误，那么通过执行顺序图中的所有可能场景(包括不同上下文环境下的场景，见第3部分)，至少能在其中的一个场景的执行过程中达到该错误，因此只要从顺序图中生成覆盖所有场景的测试用例就能有效地找出代码实现中存在的错误。

在测试用例的生成过程中，使用了范畴-划分(Category Partition)方法[2,3]来确定一个场景的上下文环境条件，然后将它与该场景的方法调用序列、输入、预期输出进行组合生成最终的测试用例。范畴-划分方法是众所周知的一个基于规格说明的功能测试方法，其基本思想是对输入领域进行划分，并从等价划分的等价类中选出一个或少量的测试来代表整个划分类的行为。在执行测试的时候还能够通过动态插装方法[9]在代码中加入不影响软件功能的观察代码，使测试人员能够观察到场景实际执行时的方法调用序列。最后将系统测试后的实际输出和方法调用序列与预期的输出和方法调用序列进行比较，从整体上验证系统的最终实现是否与设计一致，从而完成顺序图的场景测试。

下面将结合一个课程注册系统的实例片断来描述算法的具体步骤。该片断的顺序图如图2所示，注册的学生(Register)通过窗口菜单启动关闭注册的操作(closeRegistration)，该操作将完成如下功能：1) 检查注册系统的状态，如果注册系统处于打开注册的状态就继续，否则就返回；2)在注册系统处于打开状态下，取得该学生申请注册的所有课程并对每一门课程进行查询，如果该课程已有教授并且申请注册的学生数大于3，那么就提交该课程，否则就取消该课程。3)如果所有的课程都被提交，就向注册者返回注册已提交的消息，否则返回注册取消的消息，从而结束整个关闭注册的过程。该顺序图包含了分支和循环等复杂的控制流，描述了关闭注册过程中的多个可能的场景，其中的每一个场景都应该被测试到。

4.2 测试用例生成算法步骤

1. 添加状态信息到顺序图中　检查顺序图中的每一个对象，如果其存在状态图，就从中找出该对象的所有原始态(定义见第三部分)，对于每个原始态，用"对象名.in(状态名)"的格式将其加入到相应对象的生命线上并置于此对象收到的第一个消息之前；多个原始态之间用 or(逻辑或)运算符相连。

以 offering 对象为例，其状态图如图4所示。首先从顺序图可以知道 offering 对象所接收到的第一个消息为 closeRegistration，然后分析状态图可知 closeRegistration 事件被触发之前，offering 对象可能处于 Unassigned、Assigned 和 Full 状态中的任何一个，这三个状态就是 offering 对象的原始态，因此将以上三个状态信息加入顺序图中。用同样的方法将 controller 对象的状态信息加入顺序图中(schedule 对象不存在状态图，不用加入状态信息)。最终的结果如图5所示。

2. 识别出顺序图中所有场景　使用第二部分所介绍的方法通过遍历顺序图中的事件序列从而找出所有的场景。由于图5中包含分支和循环，因此需要对该方法作少许的修改。处理分支时，可以为顺序图构造多个 DAG 图，每个图包含其中一条分支路径，这样就将一个复杂的顺序图简化成多个简单顺序图来处理，遍历每一个 DAG 图就可以得到所有的场景。处理循环时，可将整个循环体看作一个节点，然后对节点内外的事件分别进行遍历并导出各自的场景，最后将节点内外的场景进行恰当地连接(用节点内的场景代替该节点)就可以了。

以图5为例，首先我们将整个循环体(消息4,5,6,7,8,9)看成一个单一的节点，记为 T。然后通过遍历的方法找出节点 T 内外的所有场景，结果如下：

T 外：〈1,2,3〉,〈1,2,T,10,11〉,〈1,2,T,10,12〉

T 内：〈4,5,6,7〉,〈4,5,8,9〉

上面的结果中，T 内的两个场景只是在循环一次的条件下可能的情况。在具体测试的时候，为了对循环体进行较为充分的测试，需要指定恰当地循环次数，可以采用跳过循环、循环一次、循环一个有代表性的次数、循环最多次的方式进行处理[10]。本实例中循环次数就是学生申请注册的课程的数目，出于验证的目的，在此我们仅假设学生申请注册了两门课程(即循环两次)，分别命名为 offering1，offering2。通过对 T 内基本场景进行简单的排列组合可得出如下的四个场景：

〈4,5,6,7,4,5,6,7〉,〈4,5,6,7,4,5,8,9〉,〈4,5,8,9,4,5,6,7〉,〈4,5,8,9,4,5,8,9〉

上面的四个场景中，每个场景的前四个消息是作用在 offering1对象上的，后四个消息是作用在 offering2对象上的。

最后，将 T 节点内外的场景恰当地连接起来。在连接的时候需要注意，不是所有的场景都能相互连接的，例如场景〈1,2,T,10,11〉只有在课程全部提交成功的条件下才发生，而 T 内的四个场景中只有〈4,5,6,7,4,5,6,7〉符合要求，所以场景〈1,2,T,10,11〉只能和〈4,5,6,7,4,5,6,7〉连接起来。同样地，T 内的其它三个场景只能和〈1,2,T,10,12〉相连。最终我们可以生成如下的5个场景：

〈1,2,3〉;〈1,2,4,5,6,7,4,5,6,7,10,11〉;〈1,2,4,5,6,7,4,5,8,9,10,12〉;
〈1,2,4,5,8,9,4,5,6,7,10,12〉;〈1,2,4,5,8,9,4,5,8,9,10,12〉。

3. 在顺序图中遍历每一个场景，获取各个场景的约束条件、输入和预期输出　从步骤2的处理结果中选定一个场景，

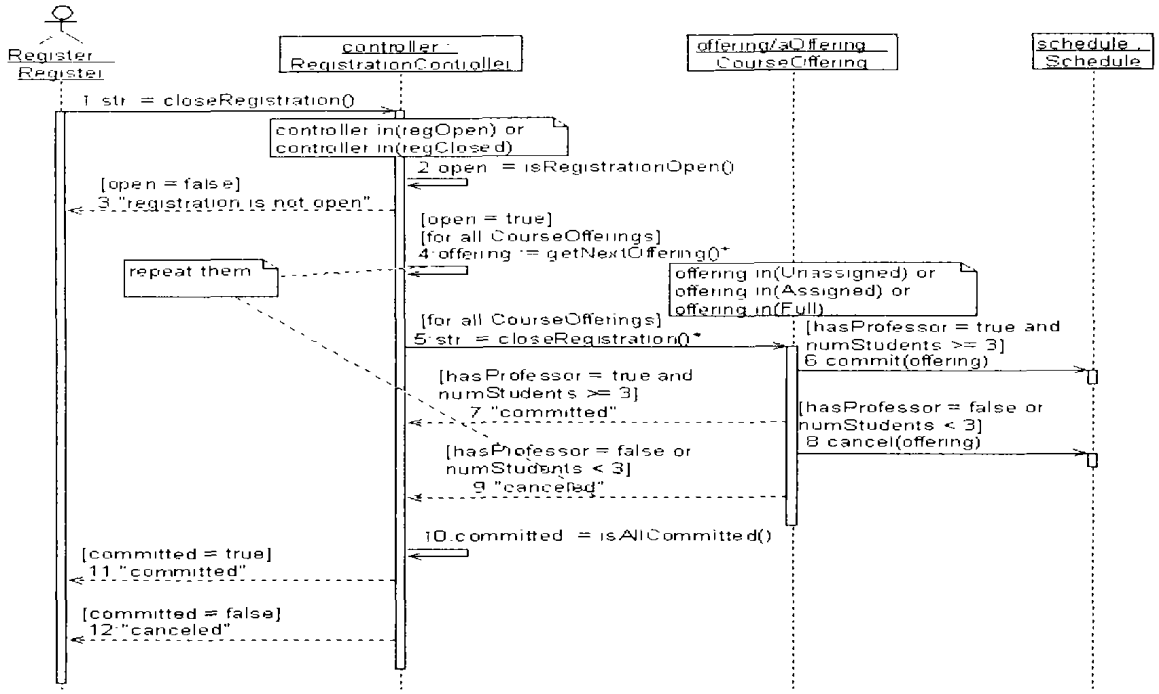

图5 加入状态信息后的 UML 顺序图

根据其消息序列在顺序图中遍历该场景，记录各个方法调用时的约束(包括状态信息和分支条件)以及场景的输入和最终输出，用 and(逻辑与)运算符将各个约束相连。以场景〈1,2,4,5,6,7,4,5,6,7,10,11〉为例，按式中的消息序列遍历顺序图的结果如下：

输入：学生通过控制台调用 closeRegistration 操作。
预期输出：返回"committed"消息，表示注册被成功提交。
约束条件：

(controller. in(regOpen)or controller. in(regClosed))and (open = true)$_{controller}$ and
((offering. in(Unassigned) or offering. in(Assigned) or offering. in(Full))
and hasProfessor = true and numStudents >= 3)$_{offering1}$ and
((offering. in(Unassigned) or offering. in(Assigned) or offering. in(Full))
and (hasProfessor = true or numStudents >=3))$_{offering2}$ (I)

其中，下标 controller、offering1和 offering2表示括号中的约束作用的对象。

(I)式中存在着一些冗余的信息，可以利用逻辑运算法则进行化简，例如 controller. in(regOpen)和 open = true 都表示注册系统处于打开状态，因此 controller. in(regOpen)and open = true 的结果等价于 controller. in(regOpen)。对(I)式进行化简的最终结果如下：

controller. in(regOpen)and ((offering. in(Assigned)and numStudents >=3)or
(offering. in(Full)and hasProfessor = true))$_{offering1}$ and ((offering. in(Assigned)and
numStudents >= 3) or (offering. in(Full) and hasProffessor = true))$_{offering2}$ (II)

(II)式中存在 or 运算符，需要将它化简成多个不含 or 的约束，结果如下：

controller. in(regOpen)and (offering. in(Assigned)and numStudents >= 3)$_{offering1}$ and (offering. in(Assigned) and numStudents >= 3)$_{offering2}$ (Ⅲ)

controller. in(regOpen)and (offering. in(Assigned)and numStudents >=3)$_{offering1}$ and (offering. in(Full)and hasProffessor = true)$_{offering2}$ (Ⅳ)

controller. in(regOpen) and (offering. in(Full) and hasProfessor = true)$_{offering1}$ and (offering. in(Assigned)and numStudents>=3)$_{offering2}$ (Ⅴ)

controller. in(regOpen) and (offering. in(Full) and hasProfessor = true)$_{offering1}$ and (offering. in(Full)and hasProffessor = true)$_{offering2}$ (Ⅵ)

以上四个约束条件反映了该场景的四种不同的上下文环境，针对其中的每一个约束条件都要为该场景生成一个测试用例。可用相同的分析方法为其余的各个场景生成约束条件。

4. 使用范畴-划分方法确定每个场景的环境条件　首先从顺序图中找出所有的测试单元。在顺序图中，每一个交互的对象就是一个测试单元；对于每一个测试单元，我们从类图中导出相应的环境设置(包括各对象的属性和消息中的参数)，环境设置中的每一项被称为一个范畴(category)。分析图5和相应的类图所得结果如表1所示。

表1

测试单元	controller	offering	schedule
环境设置	open:Boolean	numStudents:int hasProfessor:Boolean	无

表中，regOpen、numStudents、hasProfessor 分别为 controller 和 offering 对象的属性值，可从类图中获得，其中的每一个被称为一个范畴。

找出环境设置以后，就要将其中的每一个范畴(category)划分成多个选择(choice)，划分的依据是类图、顺序图和状态图中的规约信息，一个选择是一组相似值的集合，与一个等价类类似，可从中选出任意一个代表性的值来构造测试用例。以 numStudents 为例，根据类图规约可知0<=numStudents<=10，在顺序图中 numStudents<3和 numStudents>=3被用作分支条件，在状态图中 numStudents=10代表着 offering 对象处于 Full 状态，因此根据以上信息可将 numStudents 划分成三个选择，分别为0～3,4～9,10。

最后一步就是为每一个场景找出相应的选择，从而确定其环境条件。例如，场景〈1,2,4,5,6,7,4,5,6,7,10,11〉在约束(Ⅳ)下的分析如表2所示。

5. 测试用例的生成　一个测试用例包括4个部分：环境条件、输入、方法调用序列、预期输出。所有这些信息已从前面的四步中生成，只要将它们组合在一起就可以了。仍以场景〈1,2,4,5,6,7,4,5,6,7,10,11〉为例，在约束(Ⅳ)下，其测试用例为：

环境条件：
controller:open = true
offering1:numStudents=6,hasProfessor=true

offering2:numStudents=10,hasProfessor=true

输入:学生通过控制台调用 closeRegistration 操作。

方法调用序列:

controller.closeRegistration,controller.isRegistrationOpen,
controller.getNextOffering,(offering.closeRegistration) offering1,
schedule.commit,(offering.closeRegistration)offering2,
schedule.commit,controller.isAllCommitted

预期输出:返回"committed"消息,表示注册被成功提交。

表2

范畴	选择	约束	取值
open	true false	Controller.in(regOpen)	true
numStudents	0～3 4～9 10	(numStudents>=3)$_{offering1}$ offering.in(Full)$_{offering2}$	(4～9)$_{offering1}$ (10)$_{offering2}$
hasProfessor	true false	offering.in(Assigned)$_{offering1}$ (hasProffessor=true)$_{offering2}$	(true)$_{offering1}$ (true)$_{offering2}$

在这个测试用例中,offering1的 numStudents 可取4～9中的任意一个值,此处取为6。方法调用序列就是该场景中的消息序列,但去掉了其中所有返回消息,因为它们并不是真正的方法调用操作。可用同样的方法为所有场景生成测试用例。

结论 UML 是面向对象分析和设计时的标准建模语言,使用 UML 图模型来生成测试用例不仅能充分利用已有的设计结果,减少测试的费用,而且由于 UML 是一种半形式化的建模语言,测试用例的生成可以部分地实现自动化,因此基于 UML 图模型生成测试用例的研究[2~12]近年来受到越来越广泛的关注。

本文提出了一个根据 UML 顺序图生成场景测试用例的方法,包括场景的生成和使用范畴-划分方法生成测试用例。整个方法具有如下几方面的特点:1)完全基于 UML,使它容易被已使用 UML 的工业界采用。2)对顺序图进行了形式化定义,能够通过对顺序图中事件的遍历方便地得到所有的场景,而且加入了对分支和循环的处理,使得该方法的适用范围较为广泛。3)生成测试用例的过程中考虑了对象的状态信息,因此生成的测试用例较为充分。文[2]同样提出了一个使用范畴-划分方法生成测试用例的方法,但是它没有给出生成场景的算法。文[12]提出了一个可测试的顺序图模型,此模型无法对分支情况进行处理,限制了它的适用范围。文[8,9]都提出了对 UML 图进行测试的基本的策略和覆盖标准,文[9]中还给出了一个插桩算法用于跟踪协作图中的方法调用序列,可用来对测试结果进行验证,但它们都没有说明如何生成测试用例。然而我们的工作也存在一些限制,生成的测试用例只能覆盖所有可能的消息序列(场景),无法覆盖所有的事件序列,这是需要进一步改进的地方。

我们今后的研究工作是:1)提出一种与面向对象软件开发过程集成的测试过程。2)针对 UML 单个模型图,研究其在不同测试层次生成测试用例的方法,提出相应可行的测试评价标准,尽量能够直接使用 UML 模型文档,减少形式化的工作量。3)针对某一测试层次的测试用例生成时,研究综合利用待测试系统的各种模型图生成测试用例的方法。4)为上述方法提供自动的工具支持,并希望能够与主流建模工具、测试工具集成。

参考文献

1 Pressman R S 著,梅宏 译. 软件工程－实践者的研究方法(第5版). 机械工业出版社,2002

2 Basanieri F, Bertolino A. A Practical Approach to UML-based Derivation of Integration Tests. In: Proc. of QWE2000, Bruxelles, November 20-24, 3T

3 Ostrand T J, Balcer M J. The Category Partition Method For Specifying and Generating Functional Tests. Communication of the ACM,1988,31(6):676～686

4 Booch G,Rumbaugh J,Jacobson I,著,邵维忠等译. UML 用户指南. 机械工业出版社,Addison-Wesley ,2001

5 UML Specification 1.5. available at http://www.omg.org/uml

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

7 Offutt A J, Abdurazik A. Generating Tests from UML specifications. In: Proc. 2nd Intl. Conf. on the Unified Modeling Language (UML'99), Fort Collins, CO, Oct. 1999. 416～429

8 Wu Y, Chen M-H, Offutt J. UML-based Integration Testing for Component-based Software. In: The 2nd Intl. Conf. on COTS-Based Software Systems (ICCBSS). Ottawa, Canada, Feb. 2003. 251～260

9 Offutt A J, Abdurazik A. Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Proc. 3rd Intl. Conf. on the Unified Modeling Language (UML'00), York, UK, Oct. 2000. 383～395

10 Briand L, Labiche Y. A UML-Based approach to system testing. In: Proc. of the 4th intl. Conf. in Unified Modeling Language (UML 2001), volume 2185 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 2001. 194～208

11 Hartmann J, Imoberdof C, Meisenger M. UML-Based Integration Testing. In: ISSTA 2000 conf. proc. Portland, Oregon, 2000. 60～70

12 Fraikin F, Leonhardt T. Sequence Diagram Based Test Automation. http://www.pi.informatik.tu-darmstadt.de/publikationen/technische%20Berichte/2002/pi2002-2.pdf

(上接第175页)

中安插审计钩子来实现审计模块化,对内核影响较小,在内核升级时也能方便地将审计系统进行升级。我们还采用内核线程作为审计后台进程,使得审计完全内核化,更加安全。该审计子系统按照 Posix.1e 标准和中国国家标准 GB17859-1999 "计算机信息系统安全保护等级划分准则"第四级对审计的要求进行设计。目前,原型系统已在支持 LSM 框架的 Linux 2.5.72版本上实现,同时能够对 LSM 的安全模块进行审计。

参考文献

1 Alliance I. SNARE-System intrusion Analysis&Reporting Environment. http://www.intersectalliance.com/projects/Snare/index.html

2 Morris J, Smalley S, Kroah-Hartman G. Linux Security Modules: General Security Support for the Linux Kernel. In Linux Security Modules: General Security Support for the Linux Kernel, 2002. http://www.citeseer.nj.nec.com/wright02linux.html

3 石文昌,孙玉芳,等. 安全 Linux 内核安全功能的设计与实现. 计算机研究与发展,2001,38(10):1255～1261

4 中华人民共和国国家标准. 计算机信息系统安全保护等级划分准则 GB17859-1999,中国,1999

5 Final Evaluation Report-Trusted Information Systems-Trusted XENIX version 4.0, ReportNo. CSC-EPL-92/001.A, National Computer Security Center, USA, Jan 1994

6 贾春福,徐伟,郑辉. Linux 系统内核级安全审计方法研究. 计算机工程与应用,2002(6):53～55