UML活动图在组件系统测试用例生成中的应用研究

李赋欣　傅鹤岗

(重庆大学计算机学院　重庆 400044)

摘　要　近年来,基于UML的测试成为组件系统测试的研究热点。为了从UML活动图模型中自动生成组件系统的测试用例,本文采用灰盒测试方法,给出了活动图的形式化描述。分析了活动图并发特征所引起的测试场景集数量爆炸问题,提出增加约束条件的解决方法,减少了测试场景集数量。提出了一种基于活动图的组件系统测试场景与测试用例生成方法,使自动化生成测试用例可行性有一定的提高。

关键词　UML活动图,测试场景,测试用例,组件系统

1　引言

基于组件的软件工程(CBSE)逐渐成为软件开发的主流范型,是组件开发工程化的可行途径。CBSE的基本元素组件往往已经过了严格的测试,并经过广泛的使用,因此它们的可靠性通常都比较高,因而也提高了新软件的质量。因为组件系统有其独特的特点,如组件源代码不可知、集成环境的差异等等,这会使得许多测试方法不可用以及产生交互性错误。这些都为组件系统的测试带来一系列的问题[1]。

组件系统测试的一个核心问题就是测试用例的生成,测试用例自动生成的方法很多,文[2]中提出了基于关系代数形式化说明的测试用例自动生成方法,文[3]中提出了基于布尔形式化说明的测试用例自动化生成方法,可见测试用例自动生成的一个重要步骤就是对系统进行形式化建模。而UML是一种可视化的建模语言,被广泛地应用于软件系统的集成测试中。其中,UML的活动图借用了流程图、状态转换图和Petri网的思想,表示了活动可能发生的顺序,其所具有的方便以及并发特征,使得它十分适合对组件系统进行建模。但是由于UML是一种半形式化的描述方式,因此必须要对活动图进行形式化定义;而活动图的并发线程导致测试用例数量巨大的问题,也是影响测试用例自动生成的关键。

2　UML活动图介绍及形式化定义

UML活动图[4]中的建模元素由结点和边组成,结点在语义上表示过程和过程控制,包括动作状态(action state)和活动状态(activity state)、决策、泳道、同步点、对象、信号接收者、信号发送者、初始动作状态和终止动作状态。边表示活动发生的顺序以及参加活动的对象,包括控制流、消息流和信号流。活动状态表示过程执行中的一条语句或工作流中某个活动的执行,动作状态与活动状态类似,只不过动作状态是原子的,在执行过程中不允许有转换。

一个动作状态或者活动状态完成后,控制流将立刻到达下一个动作或者活动状态。用转换(transition)来表示这种从一个状态到另一个状态的流动。在语义上,活动图中的这种转换是非触发的转换,或者说是完成转换,因为一旦动作或者活动状态完成,控制流就马上转换,有向边离开某个活动状态表示该活动已经完成,卫式条件一般表示某个活动是否完成,它的布尔值属于操作的属性,是由外部环境决定的。活动图中的决策结点(用菱形表示)可以有分支,分支有一个入度和多个出度,在每个出去的转换上,标明一个布尔表达式表示选择该分支必须满足的条件。活动图中用同步条(synchronization bar)来处理并行控制流的分叉(forking)和合并(joining)问题。同步条用一条粗的水平或者垂直线表示。分叉通常有一个入度,多个出度,分别表示独立的控制流,分叉后面不同的控制流中的活动是并发执行的。合并表明对多个并发控制流的同步,合并可以有多个入度,一个出度,在合并前,不同控制流上的活动是并发执行的,到达合并时,并发的控制流被同步。

活动图是一种特殊形式的状态机,用于计算流程和工作流程建模。活动图中的状态表示计算过程中所处的状态,而不是普通对象的状态。通常,活动图假设在整个计算处理的过程中没有外部事件引起的中断,否则,普通的状态机更适合描述这种情况[4]。

由于活动图缺乏精确的语义,不利于测试用例的生成,下面我们给出对操作进行建模的活动图的形式语义。

定义1　活动图可以表示为一个多元组:$G=(P,T,F,C,p_I,p_F)$,其中,

P表示非空的有限活动集;

李赋欣　硕士研究生,主要研究方向:软件测试;**傅鹤岗**　副教授,硕士生导师,主要研究方向:软件工程,电子商务等。

T 表示非空的有限转移集；

C 是有限条件转移表达式，c_i 对应 t_i；

$F \subseteq (P \times T \times C) \cup (T \times C \times P)$ 为活动状态与变迁之间的流关系；

$p_I \in P$ 是初始活动状态，$p_F \in P$ 是终止活动状态；

只有一个转换 t 满足 $(p_I, t) \in F$；

对于任何 $t' \in T$，$(t', p_I) \notin F$ 且 $(p_F, t') \notin F$；

定义 2 对于一个活动图 $G=(P, T, F, C, p_I, p_F)$ 中的任意转换 $t \in T$，$CS \subseteq A$ 是当前状态集，可以：

(1)用 $'t, t'$ 分别表示 t 的前提和后提状态集，则有 $'t=\{|p \in P|(p,t) \in F\}$ 和 $t'=\{p \in P|(t,p) \in F\}$；

(2)用 enabled(CS)表示当前状态集 CS 可触发的转换集，则有 enabled(CS)＝{t′ | t⊆CS 都已完成，C(t)为真}；

(3)用 fired(CS)表示在某一时刻从当前状态集 CS 能够触发的唯一转换，则有 fired(CS)＝{t| t∈enabled(CS)且$(CS-'t) \cap t'=\varnothing$}，并且 t 触发后得到的新状态 $CS'=(CS-t') \cup t'$，如果有不止一个转换满足条件，则可任取一个没有触发过的转换。

(4)用 Ep 表示活动图的一个运行，则有 $Ep=CS_0 \xrightarrow{t0} CS_1 \xrightarrow{t1} \cdots \xrightarrow{t_{n-1}} CS_n$ 是一个状态和转换的序列，其中 $CS_0=\{p_I\}$，$CS_n=\{p_F\}$，CS_i 为当前状态集，且 $t_i=\text{fired}(CS_i)$，$i \geqslant 0$；$CS_i=(CS_{i-1}-'t_{i+1}) \cup t'_{i-1}$，$i \geqslant 1$。

这里的定义是本文后面对活动图进行路径分析生成测试用例的基础。

3 测试用例覆盖准则及并行处理

传统的组件系统测试应该是对软件功能进行测试，应该使用黑盒测试。但是在基于活动图的组件系统测试中，为了保证测试的充分性，必须考虑测试用例集合的有效性和覆盖准则，因此采用灰盒测试策略[4]。灰盒测试是近年来提出的方法，基于预期的软件结构和行为的高级设计模式生成测试，设计是需求规约和代码实现的中间产品，保持了需求规约里的信息，也是最终代码实现的依据，灰盒测试方法有效结合了白盒测试和黑盒测试的要素，先基于扩展的白盒方法分析结构的逻辑覆盖，然后基于黑盒方法生成满足覆盖条件的输入和输出组合生成测试用例，能够发现容易被黑盒测试和白盒测试所忽略的问题。

3.1 基于活动图的测试覆盖准则

要达到相对应组件系统的充分测试，路径覆盖是最充分的测试，但是要对活动图上所有可能的路径进行穷尽测试，在实际情况下是无法达到的，因为分支和循环会导致路径组合爆炸，所以我们在活动图上进行深度遍历寻找路径的过程中，限定循环至多发生一次，同时覆盖活动图上所有的动作状态和转换，这样得到的路径称为基本路径，本文的覆盖准则为设计测试用例使活动图上每条基本路径至少执行过一次。

3.2 活动图的并行处理

在得到基本路径之后随之而来的是，当活动图中并发的线程比较多的时候，如果任意的组合，会导致路径的爆炸。

假设有 n 个进程并发，每个进程包含的活动数目为 m_n，则生成基本路径的数目为 $(\sum_{n=1}^{n} m_n)! / \prod_{n-1}^{n} (m_n!)$。例如有 4 个进程并发，每个进程有个活动，可能的基本路径数目为 63063000 个。这种情况下必须引入动态约束条件，在这里引入与测试人员交互的方法，测试人员可以通过以下几个方面对并发活动加以约束：

(1)输入约束条件，例如活动 S_i 必须发生在 S_j 前面。

(2)以一定的规则(如频次)对并发活动指定权值，权值大的生成路径比较多。

(3)测试人员指定生成路径的数目。

通过对并发活动进行约束，可以有效地减少基本路径的生成数目，避免组合空间爆炸。如果存在两个并发进程，进程一拥有四个状态 1,3,5,7；进程二也拥有四个状态 2,4,6,8；那么在并发部分所生成的基本路径数目为 70 个，如果引入约束条件状态 8 比状态 7 先发生，那么生成的基本路径数为 35 个，可见约束条件的引入确实减少了基本路径的生成，这种约束在并发活动较多时更为有效。

4 测试场景的提取和测试用例的生成

在用 UML 活动图对组件系统进行建模后，通过考察该模型就可以提取测试的基本路径，活动图上从起始结点到终止结点任意可能的路径都是待测试组件系统的一个测试场景 TS(test scenario)，它是由动作状态和转换组成的活动执行序列。下面给出测试场景的形式化定义：

定义 3 测试场景。$G=(P, T, F, C, p_I, p_F)$ 是一个活动图，G 的一个测试场景 TS 是从初始状态到终止状态的一个当前动作状态和触发的转换的序列：

$$TS=CS_0 \xrightarrow{[c0]t0} CS_1 \rightarrow \xrightarrow{[c1]t1} \cdots \rightarrow \xrightarrow{[c_{n-1}]t_{n-1}} CS_n$$

其中 $CS_0=\{p_I\}$，$CS_n=\{p_F\}$；CS_i 为当前活动状态集，且 $t_i \in \text{enabled}(CSsi)$，$C_i$ 为 t_i 上的卫式条件，$i \geqslant 0$；

$CS_i=(CS_{i-1}-'t_{i-1}) \cup, t'_{i-1}$，$i \geqslant 1$。

4.1 组件系统测试场景的提取

为了更好地描述组件系统测试场景和测试用例的生成，下面以一个典型的组件系统(自动饮料机)的活动图模型为例进行说明。通过对组件系统接口和耦合代码的考察可以得到如图 1 所示的自动售饮

料机活动图模型。

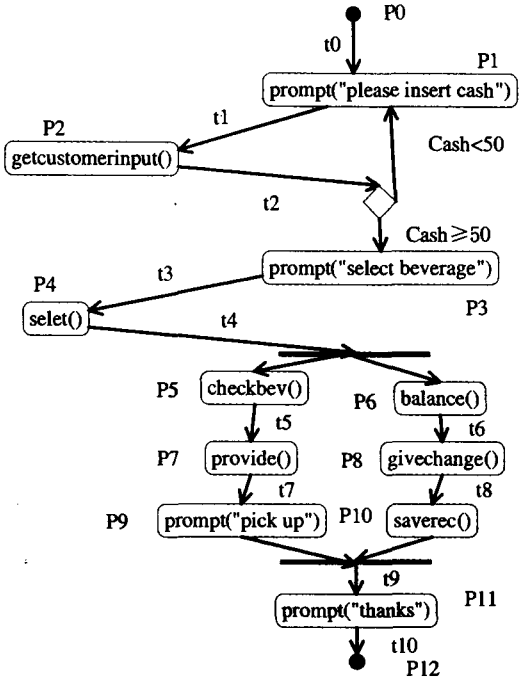

图 1　自动饮料机活动图

采用满足覆盖准则的方法，获取活动图上线程执行的所有基本路径就可以生成测试场景集。这个测试场景的生成过程应该考虑分支、并发、循环等特殊节点的影响，所以其主要思想是对条件分支和并发分支进行完全组合，在对并发分支进行组合的过程中引入约束条件，以减少测试场景的生成数量，对于循环至多遍历一次，以带回溯的深度优先方法对活动图进行遍历。

因此，测试场景提取的主要策略是：首先从 p_0 开始按照 DFS 方法遍历活动图，如果当前活动不为空，活动进栈，活动访问计数器加 1。当前状态 CS_i 存在 enabled(CS_i)，则设 CS_i 为任意 fired(CS_i)，enabled(CS_i)＝enabled(CS_i)－fired(CS_i)，把 CS_{i+1} 设置为 $CS_i-'t+t'$，继续遍历；当遇到循环，则活动访问次数达到两次，递归进入下一个可以触发的转移，然后当前活动的约束条件以及转移进栈，转移的计数器加 1；最后在程序的出口，如果 p_n 不存在可以触发的转移，则首先访问这个活动，然后读出栈中数据作为测试场景，符合约束条件的返回测试场景，不符合的置空。如果测试场景没有覆盖所有的活动和转移，回溯到上一个活动，继续遍历可以触发的活动。

按照上面的策略，当给出约束条件 P8 比 P7 优先发生时，所得到的一个测试场景 TS，其顺序如下：

p0t0p1t1p2t2(cash＜50)p1t1P2t2(cash≥50)p3t3pt4t4(p5,p6)t6(p5,p8)t5(p7,p8)t7(p7,p9)t8(p9,p10)t9p11t10p12

4.2　测试场景所对应的测试用例生成

按照测试场景的执行顺序遍历测试场景中所有结点和转换，根据结点的类型（输入、输出、操作）分别吸收相应的信息就可以生成测试用例。输入结点需要收集涉及到的输入变量信息以及相应的约束条件信息，输入变量可以从结点的文档属性中得到，约束条件需要分析从输入结点流出的转换；输出结点的输出内容可以从结点的文档属性中得到；操作结点的信息可以取结点泳道名称（即执行该操作的对象名称）加上结点的名称得到。

由于每个测试场景至少可以得到一个测试用例，因此通过上面的测试用例生成过程可得到上文所列出测试场景 TS 的一个测试用例，如表 1 所示。

表 1　TS 的一个测试用例

输入序列	期望方法序列	期望输出序列
1	prompt() Getcustomerinput()	insert cash
3	prompt() Getcustomerinput()	inset cash
	Prompt() Select() Checkbev() Balance()	select beverage
	Givechange()	找零 0
	Provide()	送出饮料
	Prompt() Saverec()	pick up
	Prompt()	thanks

最后，由测试用例的输出值与期望值作比较，如果不符合，则证明在这个对应的测试场景中存在错误。

结论　本文提出一种基于 UML 活动图的组件系统测试用例生成方法，它通过对活动图的形式化描述以及深度优先遍历，得到所需要的测试场景，进而生成测试用例。分析了测试用例生成过程中活动图的并发特征，以及其所带来的数量爆炸问题，提出了通过增加约束条件来限制测试场景的生成数量。而 UML 模型使得基于 UML 的开发过程和测试过程自然衔接成为可能，为下一步研究组件系统的测试过程和提高测试用例生成的自动化程度打下了坚实基础。

参 考 文 献

1　尚绪全，张毅坤. 基于 UML 的构件软件集成测试用例生成研究[J]. 计算机应用，2006，26(4)：96～98

2　Tsaiwt, Volovik D, Keefe T F. Automated test case generation for programs specified by relational algebra queries[J]. IEEE Transaction Software Engineering, 1990, 16(3): 316～324

3　Weyuker E, Gorad I A T, Sngh A. Automatically generating test case data from a Boolean specification[J]. IEEE Transactions on Software Engineering, 1994, 20(4): 353～363

4　邵维忠，张文娟，孟祥文. UML 用户指南. 北京：机械工业出版社，2001

5　Wang L Z, Yuan J S, Yu X F, et al. Generating Test cases from UML Activity Diagram Based on Gray-Box Method[A]. In: 11th Asia-Pacific Software Engineering Conference (APSEC 0)[C], 2004. 284～263