计算机科学 2004Vol. 31№. 1

基于状态机的 UML 行为继承关系*)

李润博　李明树

(中科院软件所互联网软件技术实验室　北京 100080)

摘　要　行为继承是面向对象领域的重要概念,UML 是面向对象设计中重要的建模语言。本文以一种抽象状态机为模型,给出了 UML 行为继承关系的形式化定义,同时证明了该定义的合理性。文章最后说明了该行为继承定义方法在 UML 中的具体实现。所讨论的行为继承与 Harel,Sourrioulle 等人的定义相比,具有更精确的含义。

关键词　行为继承,状态,状况

Statechart-Based Behavior Inheritance in UML

LI Run-Bo　LI Ming-Shu

(Lab. for Internet Software Technologies, Institute of Software Chinese Academy of Sciences, Beijing 100080)

Abstract　Behavior Inheritance is an important issue of object-oriented analysis and design paradigm. UML is an important modeling language. In this paper, the behavior inheritance relationship in UML is investigated, based on an abstract state machine model. A formal definition for behavior inheritance in UML is proposed and analyzed. At last, the implementation for such definition is given. Compared with previous work such as Harel's and Sourrioulle's, this defintion is more precise.

Keywords　Behavioral inheritance, State, Status

1　引言

继承是面向对象领域的核心概念,它对应于类与类之间的 is-a 关系。在这种意义下,当类 A 继承类 B 时,最基本的要求是类 A 和类 B 之间的接口保持一致,这样才能使类 A 在方法调用上与类 B 没有差别。然而,有时我们不仅需要类 A 和类 B 之间的接口保持一致,还希望类 A 与类 B 之间的行为保持一致(行为上 A is-a B)。这便是本文要讨论的行为继承关系。

行为继承实际上是子类型继承,它强调了父类对象与子类对象行为的一致性,从而保证了在多态环境下父类对象可在运行时被子类对象代替,同时系统的行为不发生变化。目前对行为继承的研究,主要是从对象方法的静态语义和对象的进程语义这两个角度展开[1~4],但这些模型都没有很好地说明对象行为对系统状态的潜在影响,因此不能保证运行时系统的行为不发生变化。

作为软件开发领域重要的建模语言,UML 的继承机制被定义为子类型继承关系(即行为继承)[5]。但这样的定义并不严格[6]。文[3]指出,UML 中有关行为继承的定义实际上是一些启发性的规则,不能保证任何清晰的属性。所以有必要对 UML 中的行为继承给出严格的形式化定义,从而完善 UML 的语义模型,并以此来提高程序设计的质量。

在 UML 中,对象行为主要通过状态机来描述。随着 MDA 技术的发展,状态机模型越来越趋向于可运行。凭借可运行的状态机模型,我们完全可以自动生成相应的对象行为的程序代码。因此,状态机已经渐渐成为最重要的对象行为描述方法,我们在状态机的框架内讨论 UML 的行为继承就显得必要而自然。

在下面有关 UML 行为继承关系的讨论中,我们首先给出一个抽象的状态机模型,我们称之为外视状态机。在此状态机模型上,我们给出了 UML 行为继承的形式化定义,并证明了该定义的合理性。为了实现我们的行为继承定义方法,我们还指出了该抽象状态机模型在 UML 中的具体实现。

2　UML 中的行为继承关系

Sourrioulle 定义的 UML 行为继承是一种简单的模型,只能保证子类对象可以接受父类对象的请求序列,不能保证相同的请求序列下两者的行为结果一致。Harel 扩充了 Sourrioulle 的模型,把行为继承扩展到基于状态的对象系统的框架内进行讨论,但还是以对象的进程语义为基础,如果系统行为依赖于对象状态,该模型不能保证子类对象替代父类对象时系统行为不发生变化。

我们认为,要实现所期望的行为继承关系,首先,要保证子类对象与父类对象行为的时序关系一致;其次,要保证子类对象与父类对象在相同行为时序下的行为结果一致。行为结果一致不仅意味着子类对象与父类对象在相同行为时序下具有一致的未来行为,还意味着子类对象与父类对象在相同行为时序下的观察是一致的。

为了方便下面的讨论,首先定义一些基本概念。

定义 1(观察)　是对象某些属性的值的集合$\{\langle a,d\rangle \mid a\in B, B\subseteq A, d\in dom(a)\}$。其中,A 是对象的属性集,$dom(a)$是属性 a 的值域,我们称 a 为一个可观察的属性。

定义 2(外视状态机)　一个对象的外视状态机 $\mu=(\Sigma, Q, S_0, S, G, E, R)$。

其中,Σ 是对象的可观察属性集,是对象所有可被外界观察的属性的集合;$\Sigma\subseteq A$,A 是对象的属性集;Q 是对象的可访问请求集,是对象所有可被外界访问的请求的集合;$Q\subseteq M$,M 是对象的请求集;定义对象的外视状态 $P=\Sigma\times P(dom(\Sigma))=\{(a,val(a)) \mid a\in\Sigma, val(a)\subseteq dom(a)\}$;$S_0$ 是对象的初始外视状态,$S_0\in S$,S 是对象的外视状态集;G 是条件集,是

*)本课题得到国家"八六三"高技术研究发展计划基金资助(2001AA113131)和国家自然科学基金资助(69773023)。

与外视状态无关的请求的前提条件的集合，前提条件用一阶谓词公式表示；E 是状态转换集，$E=Q/S\times G\times S=\{m/\langle S_1,g,S_2\rangle \mid m\in Q,S_1,S_2\in S,g\in G\}$，如果请求的执行没有前提条件，则 $g=1$；R 是结果集，是所有请求执行后产生的结果的集合，$R\subseteq Q\times S\times P(dom(Q))$，$R=\{\langle m,s,val(m)\rangle \mid m\in Q,s\in S,val(m)\subseteq dom(m)\}$，$dom(m)$是请求 m 的结果的值域，如果 m 的结果为空，$dom(m)=\varnothing$。

定义 3(状态包含) $S_1\subseteq S_2$，当且仅当$\forall\ \langle a,val(a)\rangle\in S_1,\exists\ \langle b,val(b)\rangle\in S_2,a=b$ 且 $val(a)\subseteq val(b)$。

定义 4(结果包含) $R_1\subseteq R_2$，当且仅当$\forall\ \langle m,s,val(m)\rangle\in R_1,\exists\ \langle n,t,val(n)\rangle\in R_2,m=n$ 且 $val(m)\subseteq val(n)$。

定义 5(初始状态) 设状态转换 $e=m/\langle S_1,g,S_2\rangle$，定义状态转换 e 的特征请求为 $method(e)=m$，定义状态转换 e 的初始外视状态为 $origin(e)=S_1$，定义状态转换 e 的终止外视状态为 $end(e)=S_2$，定义状态转换 e 的前提条件为 $guard(e)=g$。

请求 m 在状态转换 e 下的初始状况为 $O(m)=\langle S,g\rangle$，$m=method(e)$，$S=origin(e)$，$g=guard(e)$。

设 $O(m)=\langle S,g\rangle$，$O(n)=\langle S',g'\rangle$。

定义 $O(m)\subseteq O(n)$，当且仅当 $S\subseteq S'$ 且 $g\Rightarrow g'$。

定义 6(终止状况) 设结果 $r=\langle m,s,val(m)\rangle$，定义结果 r 的隶属请求为 $sub(r)=m$，定义结果 r 的伴生末状态为 $end(r)=s$，定义结果 r 的值为 $val(r)=val(m)$。

请求 m 在状态转换 e 下的终止状况为 $E(m)=S\cup\{\langle m,val(r)\rangle\},r\in R,m=method(e)=sub(r),S=end(e)=end(r)$。

定义 $E(m)\subseteq E(n)$，当且仅当$\forall\ \langle x,val(x)\rangle\in E(m),\exists\ \langle y,val(y)\rangle\in E(n),x=y$ 且 $val(x)\subseteq val(y)$。

定义 7(请求序列) 对一个对象而言，如果它可以从功能上替代另一个对象，首先它要能够接受原对象的请求序列[9]。

定义对象的一个请求序列为 $seq=e_1e_2\cdots e_n,e_k\in E,1\leqslant k\leqslant n,origin(e_1)=S_0,end(e_i)=origin(e_{i+1}),0<i<n$，对象 A 的请求序列集为 Traces(A)，$\varepsilon\in$ Traces(A)

定义 8(可访问状态) 根据上面的讨论，我们定义：请求序列 seq 的可访问状态为 $Access(seq)=S,S=end(e_n)$，$Access(\varepsilon)=S_0$。

2.1 行为继承的定义

行为继承关系的定义：我们用$<$来表示。

设对象 A 的外视状态机为$\langle\Sigma_A,Q_A,S_{0A},S_A,E_A,R_A\rangle$，对象 B 的外视状态机为$\langle\Sigma_B,Q_B,S_{0B},S_B,G_B,E_B,R_B\rangle$，存在映射：

$f:\Sigma_B\to\Sigma_A,\forall\ a\in\Sigma_A,\exists\ b\in\Sigma_B,a=f(b)$且$\forall\ a_1,a_2\in\Sigma_A,a_1\neq a_2,a_1=f(b_1),a_2=f(b_2)$，有 $b_1\neq b_2$

如果 A，B 满足如下原则：

(1)符号量原则：$\forall\ m\in Q_A,n\in Q_B$，若 $m=n$，那么

参数抗变性：m,n 的参数数目相同。设 m 的参数序列为 a_i,n 的参数序列为 b_i 那么$\forall\ i,a_i<b_i$

结果协变性：或者 m,n 都有返回结果，或者两者都没有。如果有返回结果，设 m 的返回结果类型为 a,n 的返回结果类型为 b，那么 $b<a$

异常原则：n 产生的异常包含在 m 产生的异常集中。

(2)状态原则：$\forall\ e\in E_A,\exists\ e'\in E_B$，

$method(e)=method(e')$

$O(e)\subseteq f(O(e'))$

$f(E(e'))\subseteq E(e)$

(3)轨迹原则：$\forall\ seq\in$ Traces$(A),\exists\ seq'\in$ Traces(B)，

$seq=e_1e_2\cdots e_n,seq'=e'_1e'_2\cdots e'_n,e_i,e'_i$ 满足状态原则，$1\leqslant i\leqslant n$

(4)访问原则：$\forall\ seq\in$ Traces$(A),seq'\in$ Traces(B)若

$seq=e_1e_2\cdots e_n,seq'=e'_1e'_2\cdots e'_m,n\leqslant m,e_i,e'_i$ 满足状态原则 $1\leqslant i\leqslant n$，那么，不存在 $e\in E_A,e'$，与 e 满足状态原则 $n<j\leqslant m$ 且 $Access(seq)\supseteq f(Access(seq'))$

那么我们称对象 B 行为继承对象 A，即 A$<$B。

我们定义 UML 中类 A 行为继承类 B，当且仅当在外视状态机模型中类 A 的对象行为继承类 B 的对象。

2.2 行为继承定义的说明

符号量原则指出了行为继承对对象方法的最基本约束要求。我们的描述参考了文[2]中的符号量原则。前两条是基本的协变/抗变性原则，它保证了父类对象和子类对象的相关方法在输入输出上的一致性。而异常原则则保证了子类对象相应的方法不能产生父类对象原方法不能识别的异常。

状态原则保证了父类对象和子类对象相关方法的状态一致性。第一条原则保证了子类对象相应的方法可以在任何父类对象允许的状态下被调用。而第二条原则保证了子类对象相应的方法对相关属性的修改没有超出父类对象状态变化的范围，也就是说，相同的方法执行后，子类型对象与父类型对象的观察一致。

轨迹原则强调了子类对象对父类对象历史属性的保持，这一点在多用户环境下是相当重要的[2]。从轨迹原则我们可以直接推出子类对象的请求序列集包含父类对象的请求序列集，从而保证了子类对象可以完全接收父类对象的请求序列。

然而，单纯依靠轨迹原则不能保证多用户环境下子类对象对父类对象历史属性的保持。在图 1 中，对象 A 和对象 B 遵循符号量原则、状态原则和轨迹原则，但对象 B 和对象 A 之间不存在行为继承关系。假定系统中有两个用户 x,y，用户 x 的视角中对象 B 就是自身，用户 y 的视角中对象 B 是对象 A。当 y 调用对象 B 的 a,b 方法时，对象 B 的外视状态与对象 A 的外视状态保持一致，也就是说，在用户 y 的视角中，对象 B 仍然是对象 A。但如果这时 x 调用了对象 B 的新增方法 c，对象 B 的外视状态就从 S_3 变成了 S_4，而 S_4 并不是对象 A 的一个有效外视状态，这时从用户 y 的视角看，对象 B 就不再是对象 A 了，因而破坏了对象 B 对对象 A 的历史属性的保持。所以在考虑请求序列的同时，我们还需要考虑子类对象的新增方法不会对父类对象的历史属性产生影响。

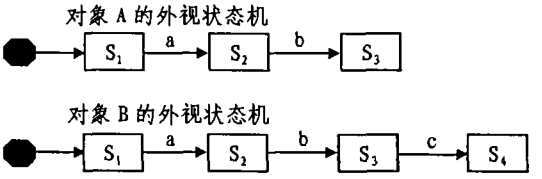

图 1 子类对象的新增行为

最后的访问原则解决了这一问题，它保证了子类对象的新增方法不会破坏父类对象的历史属性。在图 1 中，对象 A 的 Access(ab)为 S_3，而对象 B 的 Access(abc)为 S_4，S_3 并不包含 S_4，所以对象 B 与对象 A 之间不存在行为继承关系。

值得注意的是，在外视状态机的框架内，我们定义的状态原则比 Liskov 等人所定义的前件/后件原则范围要大，它允许子类型方法在父类型方法定义域之外仍然有效。

例如：下面两个操作模式分别表示父类对象和子类对象的方法 m 所产生的状态转换，可以看出它们满足状态原则。

父类对象方法 m 的前置条件为 $1\leqslant a\leqslant 10$，后置条件为 $1\leqslant a\leqslant 2$。

子类对象方法 m 的前置条件为 $1\leqslant a\leqslant 10$ 且 $1\leqslant b\leqslant 10$，后置条件为 $1\leqslant a\leqslant 2$ 且 $b=1$。

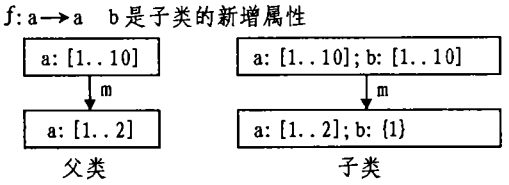

图 2 状态原则的例子

子类对象方法的前置条件并不蕴含父类对象方法的前置条件，但在外视状态机的框架内，条件 $1\leqslant b\leqslant 10$ 可以在这个转换发生之前就被满足。因此从父类对象方法的定义域看，在行为上子类对象依然可以替代父类对象。

由此可见，我们定义的行为继承比 Liscov 定义的行为子类型化继承要弱，如上例并不满足 Liskov 的前件/后件原则，但在行为上，子类对象可以替代父类对象。因此，我们定义的行为继承可以给程序设计者使用继承时提供更大的灵活性。

2.3 行为继承定义的合理性

我们将从两个方面证明定义的合理性。首先，我们要证明定义保证了子类对象和父类对象行为时序关系的一致；其次，我们要证明定义保证了在相同行为时序下子类对象和父类对象的行为结果一致。

证明 1 定义保证了子类对象和父类对象行为时序关系的一致。

子类对象与父类对象的行为时序关系一致，实际上就是子类对象可以接受父类对象的全部请求序列。

证明：设类 A、B 满足定义的行为继承关系，B 是 A 的子类。A 的实例的外视状态机为$\langle \Sigma_A, Q_A, S_{0A}, S_A, G_A, E_A, R_A\rangle$，B 的实例的外视状态机为$\langle \Sigma_B, Q_B, S_{0B}, S_B, G_B, E_B, R_B\rangle$，由轨迹原则

$\forall\ seq\in \mathrm{Traces}(A), \exists\ seq'\in \mathrm{Traces}(B)$

$seq=e_1e_2\cdots e_n, seq'=e'_1e'_2\cdots e'_n, e_i, e'_i$ 满足状态原则 $1\leqslant i\leqslant n$

因为 $\epsilon_A\Leftrightarrow\epsilon_B$

所以设 $seq=seq_1\cdot e_n, seq'=seq'_1\cdot e'_n, seq_1, seq'_1$ 是等价的请求序列，则系统请求 B 执行序列 seq_1 时，B 实际执行子序列 seq'_1，

因为 $\mathrm{gurad}(e_n)\Rightarrow\mathrm{guard}(e'_n)$

所以在执行了请求序列 seq'_1 后，如果系统请求 B 执行 e_n，即系统满足了 $\mathrm{gurad}(e_n)$，则 e'_n 会执行，

因此 $e_n\Leftrightarrow e'_n, seq\Leftrightarrow seq'$

seq, seq' 是等价的请求序列

所以，定义保证了子类对象和父类对象行为时序关系的一致。

证明 2 定义保证了在相同行为时序下子类对象和父类对象的行为结果一致。

证明：设类 A、B 满足定义的行为继承关系，B 是 A 的子类。A 的实例的外视状态机为$\langle \Sigma_A, Q_A, S_{0A}, S_A, G_A, E_A, R_A\rangle$，B 的实例的外视状态机为$\langle \Sigma_B, Q_B, S_{0B}, S_B, G_B, E_B, R_B\rangle$，由访问原则

$\forall\ seq\in \mathrm{Traces}(A), seq'\in \mathrm{Traces}(B)$若

$seq=e_1e_2\cdots e_n, seq'=e'_1e'_2\cdots e'_m, n\leqslant m, e_i, e'_i$ 满足状态原则 $1\leqslant i\leqslant n$

由证明 1 可得 seq, seq' 是等价的请求序列，

因为不存在 $e\in E_A, e'_j$ 与 e 满足状态原则 $n<j\leqslant m$，

所以 seq' 仅与 seq 是等价的请求序列。

因为 $\mathrm{Access}(seq)\supseteq f(\mathrm{Access}(seq'))$

所以等价请求序列下，子类对象与父类对象的行为结果一致，所以，定义保证了在相同行为时序下子类对象和父类对象的行为结果一致。

3 行为继承定义的评价

对于状态原则而言，判断两个对象是否满足状态原则的时间复杂度的上限是 $O(m\times n\times p)$，其中 m 为父类对象的外视状态机中的可观察属性数，n 为父类对象的外视状态机中的状态转换数，p 为子类型对象的外视状态机中的状态转换数。

对于轨迹原则而言，如果父类对象的状态机是确定性的，那么子类对象的请求序列集与父类对象的请求序列集是线性包含关系，其时间复杂度是 PSPACE-完全；如果父类对象的状态机是非确定性的，那么子类对象的请求序列集与父类型对象的请求序列集是树型包含关系，其时间复杂度是 PTIME-完全[4]。

对于访问原则而言，如果确定了等价的请求序列，判断两个对象是否满足访问原则的实质就是外视状态机的深度优先搜索问题。其时间复杂度是 $O(n+e)$，n 为子类对象外视状态机的节点数，e 为子类对象外视状态机的状态转换数。实际运行中，如果子类对象并不是面向多用户的，那么可以不考虑访问原则。这样整个方法的复杂度就仅与状态原则和轨迹原则相关。

4 行为继承定义的实现

UML 本身提供了状态机模型来描述对象的动态行为。然而这是一种抽象的状态机模型，其状态并没有和对象属性的观察绑定。因此我们需要对 UML 的状态机模型进行扩充，以使其可以表示上面建立的外视状态机模型。

UML 提供了三种嵌入的扩充机制：Stereotypes，Constraints，Tagged values。

(1)Stereotypes 是 UML 中最重要的扩充机制，提供了一种在模型中加入新的建模元素的方式，可以在 Stereotypes 中定义相关的 Constraints 和 Tagged valuse 以说明特定的语义和特征。

(2)Tag valuse 可以对模型中的建模元素加入新的属性。Tag 表明了建模元素可以扩展的特性的名称，value 可以是任意的值，值的范围取决于用户或工具对 tag 的解释。对每一个 Tag 名，一个建模元素至多有一个给定的 Tag value。

(3)Constraints 是对建模元素的语义上的限制。

4.1 扩充描述

这一节，我们对 UML 的状态机模型进行扩充。

(1)在 UML 状态机模型中增加一种状态，称为外视状态，我们用 ExternalState 来表示。在元模型中，外视状态是 SimpleState 的子类。

外视状态有一个 Tagged Value，observation。observation 表示在该状态下，可观察属性的值域集合。

observation：set

(2)初始外视状态没有输入状态转换，至多只有一个输出状态转换。

(self.kind=#initial)implies

((self.outgoing→size≤1) and (self.incoming→isEmpty))

(3)终止外视状态没有输出状态转换。

(self.kind=#final)implies(self.outgoing→isEmpty)

(4)在 UML 状态机模型中增加一种状态转换,称为外视状态转换。在元模型中,外视状态转换是 transition 的子类。

self.trigger=CallEvent

self.target.oclIsTypeof(ExternalState)

self.source.oclIsTypeof(ExternalState)

self.result 表示该状态转换发生后产生的结果 self.result:set

(5)在 UML 状态机模型中增加一种状态机,称为外视状态机。

self.state.oclIsTypeof(ExternalState)

self.top.incoming→isEmpty

4.2 实例

为了说明我们扩充的 UML 的外视状态机模型,下面给出一个简单的实例。类 Reducer 表示一个减速器。它有三个属性,upperLimit,speed 和 factor,分别表示减速器的速度上限,当前速度和衰减因子。其中 speed 是可观察属性。类 EnhancedReducer 表示一个增强型的减速器。它有四个属性,upperLimit,speed,factorA 和 factorB,speed 是可观察属性。其中 upperLimit,speed 分别与 Reducer 中的 upperLimit,speed 相对应,表示 EnhancedReducer 的速度上限和当前速度。factorA 和 factorB 是 EnhancedReducer 的特有属性,表示 EnhancedReducer 的两个衰减因子。Reducer 和 EnhancedReducer都有一个 decelerate 方法,负责减慢减速器当前的速度。伪代码表示如下:

```
Class Reducer{
Public:
        speed:Int
Private:
         upperLimit:Int
         factor:Int
Public:
        decelerate();
Reducer();
...
}
Reducer(){
           upperLimit=300;
           speed=upperLimit;
           factor=150;
}
decelerate(){
  if(speed≥factor)
        speed=speed-factor;
}
Class EnhancedReducer{
Public:
        speed:Int
Private:
         upperLimit:Int
         factorA:Int
         factorB:Int
Public:
        decelerate();
Private:
         reduceA(int x);
         reduceB(int y);
EnhancedRecuder();
...
}
EnhancedReducer(){
            upperLimit=300;
            speed=upperLimit;
            factorA=100;
            factorB=50;
  }
decelrate (){
          reduceA(speed);
          reduceB(speed);
}
reduceA(int x){
          if(x≥factorA)
                x=x-factorA;
          else
                reduceB(x);
}
reduceB(int y){
          if(y≥factorB)
            y=y-factorB;
}
```

Reducer 和 EnhancedReducer 的实例的 UML 的外视状态机模型表示如下:

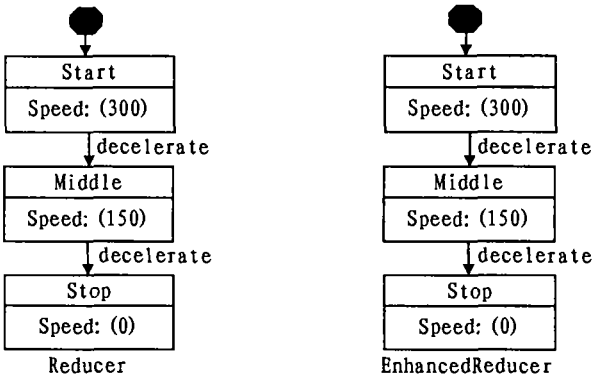

图3 外视状态机示意图

不难看出,尽管 Reducer 和 EnhancedReducer 采用了不同的衰减因子,而且彼此 decelerate 方法的实现具有很大的差异,EnhancedReducer 的 decelerate 方法实现中还包含若干内部状态变换,但从外视状态机的角度来看,两者是相同的模型。因此,它们满足我们的行为继承关系的定义,类 EnhancedReducer 是类 Reducer 的子类型。

结论和进一步的研究 本文讨论了 UML 的行为继承关系,在一种抽象状态机(外视状态机)的基础上,给出了 UML 行为继承关系的形式化定义,并证明了该定义的合理性。

在讨论时,我们没有考虑子类对象的定义域和父类对象的定义域的模拟关系,函数 f 仅仅是一个简单的名称变换函数,如何在模拟关系下实现状态包含值得进一步研究。另外,如何在我们扩充的 UML 状态图模型中,实现行为继承关系判断的自动化,这项工作还需要进一步研究。

参 考 文 献

1 Bolton C, Davies J. On giving a behavioural semantics to activity graphs. In: UML 2000 Workshop Dynamic Behaviour in UML Models: Semantic Questions

2 Liskov B H, Wing J M. A Behaviouralal Notion of Subtyping. ACM Trans. On Programming Languages and Systems, 1994, 16 (6): 1811～1841

3 Sourrouille J L. UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages. UML 99, 1999

4 Harel D, Kupferman O. On the behavioral inhritance of state-based objects. In Technology of Object-Oriented Languages (TOOLS). IEEE Computer Society Press, Los Alamitos, CA, July 30th--August 3rd 2000

5 OMG Unified Modeling Languages Specification, Version 1.3. March 1999

6 Linington P F. Options for Expressing ODP Enterprise Communities and their Policies by Using UML. In: Proc. of the Third Intl. Enterprise Distributed Object Computing Conf. IEEE, Sep. 1999. 72～82

7 Synder A. Encapsulation and inheritance in object-oriented programming languages. In: Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Portland, Oregon, Sep. 1986

8 Brookes S D, Hoare C A R, Roscoe A W. A Theory of Communicating Sequential Processes. Journal of the ACM, 1984, 31(3): 560～599