并发面向对象中的继承反常现象[1]

王生原　杨良怀　袁崇义 杨萍

Abstract. 如果不考虑继承性，并发性与对象技术的结合是很自然的。继承反常[2]（*inheritance anomaly*）现象是继承性和并发性不相容的主要原因之一。现阶段人们对继承反常现象的认识有许多模糊之处，出发点不尽相同，形式化的工作也很少。本文认为继承反常这一概念应有针对性，对不同的 *subtyping* 关系应考虑其特有的渐增式继承（*incremental inheritance*）方法，这有利于把握继承反常现象的实质，而不被各类不同的观点所误导，同时也丰富了“在并发面向对象语言中应将 *inheritance* 层次和 *subtyping* 层次区别对待”这一认识的内涵。在阐述基本观点后，本文采用范畴论的术语对继承反常现象给出了一种形式化的定义，并对并发面向对象技术中继承性的建模提出相关的建议。

Key Words: 并发性；面向对象；继承反常；渐增式继承；范畴论

1. 引言

Inheritance（继承）是面向对象理论和技术中主要的概念之一。大多数人已认同这样的观点：一个程序设计语言，即使它支持对象（*Object*）和类（*Class*）的概念，如果不支持 *inheritance*，那么也不能称为是面向对象语言。继承是类之间的一种层次关系。*Superclass* 描述了同类 *subclass* 的共性（泛化），而 *subclass* 旨在继承其 *superclass* 的这种共性，并扩展其特性（特化）。继承是代码重用（*Code Reuse*）的重要途径之一。

在一般的面向对象语言中， 类层次自动对应了一种类型层次[WZ88]，这样，*superclass* 和 *subclass* 之间的关系自然是一种 *supertype* 和 *subtype* 之间的关系。这种对应关系难免使人们在 *inheritance* 和 *subtyping* 之间产生模糊的认识。对于大多数顺序的面向对象语言来说，*inheritance* 和 *subtyping* 不加以区分几乎不会造成什么影响。但随着把并发性(*concurrency*）引入面向对象语言中（如[Agha86] [America90] [Yonezawa90]），人们发现 *inheritance* 和 *subtyping* 之间并不容易协调，会出现所谓的继承反常（*Inheritance Anomaly*）现象 [MY93]。

在并发面向对象语言（*COOL*）中，对象是很自然的并发单元。对象之间的通信和同步依靠消息传递，而对象内部的并发活动则是通过共享变量方式进行通信和同步。无论对象之间还是对象内部的通信和同步，都和同步代码（*Synchronization Code*）有关。同步代码的主要作用是对对象内部及其与外界的接口的并发活动（如并发方法，内部线程，对共享变量的操作等）进行一种约束，即同步约束（*Synchronization Constrains*）。在 *inheritance* 层次中，方法（及属性）的增改很可能会破坏这种约束。换句话说，*subclass* 要兼容其 *superclass* 的这种同步约束，很可能必须对相关的 *superclass* 代码进行实质性修改，而不可能重用它们。这种现象在[MY93]中被首次称为继承反常。在此之前也有许多这方面问题的讨论（[America87][KL89]），目前仍很受关注（[LLNW96][CRR98]）。

对于继承反常现象，目前的解决方案主要有两类：（1）设计更强的同步机制[MY93]；（2）去掉同步原语[Meseguer93][LLNW96]。但这些方法只能有限地减小继承反常的危害，并不能从根本上解决问题。面对所遇到的困难，有人在考虑是不是存在概念上的误区[BCC95]，目前这仍是一个开放的话题。实际上，除了同步约束，继承反常现象也可能是由其它情况引起的（[ABSB94]），这说明继承反常应属 *inheritance* 本身的特征。

本文认为继承反常这一概念应有针对性，并发面向对象中的继承不能只狭隘地沿用顺序程序中渐增式的继承（*incremental inheritance*），对不同的 *subtyping* 关系应考虑其特有的渐

[1] Supported by the National Natural Science Foundation of China under grant No. 69973003, and by the China NKBRSF (973) under grant G1999032706.

[2] 又译：继承异常.

增式继承方法。这有利于把握继承反常现象的实质，而不被各类不同的观点所误导，同时也丰富了“在并发面向对象语言中应将 *inheritance* 层次和 *subtyping* 层次区别对待”这一认识的内涵。

另一方面，不同背景和意义的“继承反常”足以使读者产生困惑。如有些工作称可以避开继承反常现象[Meseguer93][LLNW96]，而另一些工作则认为继承反常是不可避免的[CRR98]。究其原因主要是这方面的形式化工作很少，且不够全面。本文在上述观点的基础上，采用范畴论的术语对继承反常现象给出了一种形式化的定义。

第 2 节通过经典实例使读者初步认识继承反常现象；第 3 节分析继承反常现象的实质及阐述本文理解这种现象的基本观点；第 4 节用范畴论的术语形式地定义继承反常现象；第 5 节利用本文的定义对文献中出现的有关继承反常现象的典型情形进行了解释，并结合本文的观点，对并发面向对象技术中继承性的建模提出一些有益的建议。

2. 认识继承反常现象

```
class   Buffer {
int   in=out=0,   buf[SIZE];
method   put()   when   (in < out + SIZE) { in++;   //rest code of put() };
method   get()   when   (in >= out + 1) {out++;   //rest code of get() };
}
class   Buffer2: Buffer {
method   get2()   when   (in >= out + 2) { out + = 2;   //rest code of get2() };
}
class   GBuffer: Buffer {
bool   after_put = false;
method   gget()   when   (!after_put && (in >= out + 1)) { out++;   after_put = false;   //rest code of gget() };
method   put()   when   (in < out + SIZE) { in++;   after_put = true;     //rest code of put() };
method   get()   when   (in >= out + 1) {out++;   after_put = false;   //rest code of get() };
}
class   LockableBuffer: Buffer {
bool   locked = false;
method   lock()   when   (!locked) {locked = true };
method   unlock()   when   (locked) {locked = false };
method   put()   when   (!locked && (in < out + SIZE)) { in++;   //rest code of put() };
method   get()   when   (!locked && (in >= out + 1)) {out++;     //rest code of get() };
}
```

图 2.1　有界缓冲区类 Buffer，及其子类 Buffer2、GBuffer 和 LockableBuffer

考虑图 2.1，类 Buffer 实现了一个有限缓冲区类型，可以并发地接受消息 *put* 和 *get*。同步机制采用了“*method guards*”方式，即为每一 *method* 附加一个 *guard* 谓词，在图 2 .1 中形如“method *method_name* when *guard*”。如果对类 Buffer 增加新方法 *gget* 来构造子类 GBuffer，就不得不修改同步代码，从而方法 *put* 和 *get* 的代码须重新定义。这里 *gget* 的行为同 *get*，但不可以紧跟在 *put* 之后执行。同样，构造 Buffer 的另一个子类 LockableBuffer 也会引发同样的问题。

这种“为获有效继承而必须对父类代码进行实质性修改的现象”就是所谓的继承反常（*Inheritance Anomaly*）[MY93]。

继承反常这一概念是针对某种特定的语言机制而言的。图 2.1 中采用“*method guards*”同步机制，类 Buffer2 可以不加修改地继承 Buffer 的方法 *put* 和 *get*（其中新方法 *get2* 为每次从缓冲区取出 2 个元素）。而语言 ACT++（一种基于 Actor 的语言[KL89]）采用 *Behavior Abstractions* 机制，Buffer2 在继承 Buffer 时不可避免地要重新定义方法 *put* 和 *get*，见图 2.2。

下一节我们将讨论如何准确地理解继承反常现象，阐述本文的基本观点。

3. 理解继承反常现象

综合各种观点，要理解和解释继承反常现象，首先须正确区别 *inheritance* 层次和 *subtyping* 层次及深刻领会二者的联系。

3.1 区别 Inheritance 和 Subtyping

Inheritance 是类（*class*）之间的一种层次关系，而 *subtyping* 是类型（*type*）之间的层次关系。*Type* 具有类型检测（*type-checking*）的语义，可以利用谓词来定义，即满足该谓词的所有对象都具有这种类型；而 *class* 具有实例创建的语义，是产生实例的模板（*templates*）。每个 *class* 对应一种特定的 *type*，这种 *type* 的谓词是该 *class* 模板的一种规范；但反过来每个 *type* 不一定都对应一个 *class*，因为一种谓词未必可以成为某个模板的规范[WZ88]。

```
class   b_buf: ACTOR { // b_buf is an Actor
   int   in, out, buf[SIZE];
behavior:
   empty = {put};     partial = {put, get};       full     = {get};
public:
   void b_buf() {in = out = 0;   become empty;   }
   void put() { in++;   // store an item
                  if (in == out + size) become full;   else become partial;
   }
   void get() { out++;   // remove an item
      if (in == out) become empty;   else become partial;
   }
}

class   b_buf2: b_buf { // b_buf2 is a subclass of b-buf
behavior:
   x_empty = {put};   x_one = {put, get};   x_partial = {put, get, get2};   x_ full = {get, get2};
   // x_empty   renames empty,   x_partial   redefines partial;   x_full   redefines full;
public:
   void b_buf() {in = out = 0;   become x_empty;   }
   void get2() { out += 2;         // definiton of get2
      if (in == out)   become   x_empty;
      else   if   (in == out + 1) become x_one;   else become x_partial;
   }
   // The following re-defines the methods in b-buf.
   void put() { in++;
      if (in == out + size)   become x_full;
      else   if   (in == out + 1) become x_one;   else become x_partial;
   }
   void get() { out++;
      if (in == out)   become x_empty;
      else   if   (in == out + 1) become x_one;   else become x_partial;
   }
}
```

图 2.2　用 ACT++描述的有界缓冲区类 b_buf，及其子类 b_buf2

在面向对象语言中，*class* 是一个语法单位，它包含若干属性及方法的说明，每个 *class* 的实例共享这些说明。*Type* 可看作是共有某些外部可观察（*externaly observable*）行为特性的实例的集合，即 *type* 实际上是一种对其元素的行为规范（对应以上所述的“一种谓词”[America91]），如抽象数据类型（*abstract data types*）是较常用的行为规范技术。

Subtype 是根据谓词来修改，而 *subclass* 是根据模板来修改。*Inheritance* 属于后者，而 *subtyping* 属于前者。*Inheritance* 提供了一种类之间共享代码的机制。通过 *inheritance*，*subclass* 可重用（*reuse*）*superclass* 的代码。P.America 指出[America91]，通过 *inheritance* 进行代码共享不一定自动带来行为的特化（如引入或修改的新的属性和方法可能破坏原有的属性和方法赖以发挥作用的不变量），并认为应该用 *subtyping* 而不是 *inheritance* 来特指行为特化。

本文中 *subtyping* 的概念满足可替换性原则 (*Principle of Substitutability*)[WZ88]：*An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*

综上所述，*inheritance* 是在代码层次上作修改，而 *subtyping* 是在语义层次上作修改。前

者是代码共享的一种重要途径，但不能保证 *subclass* 能够继承 *superclass* 的行为；后者要求 *subtype* 保持 *supertype* 的某种外部可观察行为（或语义行为），在规范一级共享，同代码没有关系。*Inheritance* 层次关系可以理解为“*is_similar_to*”（或“*like*”）的关系，而将“*is_a*”关系用在理解 *subtyping* 层次关系更妥（可以描述成“从某种语义行为来看，*B* 是 *A* 的 *subtype* ⇔ 每个 *B* 的实例 *is_a A* 的实例”）。

3.2 渐增式继承

Subtyping 要求 *subtype* 保持 *supertype* 的某种行为（可看作是一种不变量，比如同步约束），*subclass* 在增加新的属性或方法时，为了避免破坏这种不变量，难免要对从 *superclass* 继承下来的代码进行扩展或修改。而这种扩展或修改很可能是重大的或实质性的，结果使得 *inheritance* 层次从代码共享的角度看变得没有意义。这便是继承反常的（非形式）含义。

那么什么样的扩展或修改算是重大的或实质性的呢？这是比较含糊的说法，以至于多种关于继承反常的说法使读者产生误解。如有些工作称可以避开继承反常 [Meseguer93] [LLNW96]，而另一些工作则认为继承反常是不可避免的[CRR98]。[CRR98]在 *inheritance* 必须是渐增式的假设下，给继承反常下了一个形式化的定义。按此定义，先前人们声称对继承反常问题的解决都是部分的，即没有任何一种解决方案是彻底的。这是一个极端的定义，它所规定的渐增式继承不允许对继承下来的代码作任何形式的修改。为了对人们在解决这个问题上的努力有所区别，同时兼顾到不同的语言特征，我们有必要对“渐增式继承”采取较为灵活的解释。

首先，对于不同的并发面向对象语言（*COOL*），渐增式继承的含义应该有所不同。如对基于 Petri 网的 *COOL*（[Lakos95] [BCC95]），将子网（*Subnets*）关系看作渐增式继承是恰当的，没必要苛求子类网中新增的部分与从父类继承的部分不相交。参考图 3.1，N_0 是 N_1、N_2 和 N_3 的子网，N_1、N_2 和 N_3 渐增继承了 N_0。

然而，这种子网关系的继承只强调了网结构（代码）的重用，而未顾及行为的重用。在顺序的面向对象程序设计中，代码的重用是 *inheritance* 的一个主要目的，而行为的重用几乎可以不去考虑，因为方法之间最多只有调用关系，没有同步代码带来的同步约束，其它约束（如实时方面的约束）只是在特定情况下才考虑。对于并发面向对象的程序，情况就不同了，代码的重用固然是 *inheritance* 的一个重要目的，但行为的重用更突显其重要地位，也成为问题的焦点。

行为的重用在 *COOL* 中是由 *subtyping* 关系体现的。从前面章节的讨论可知，代码重用和行为重用之间的不兼容是产生继承反常的根源。因此，为了兼顾两种重用，在 *COOL* 中严格区分 *inheritance* 层次和 *subtyping* 层次成为一种趋势，许多语言设计了这样的机制，如 CO-OPN/2 [Biberstein97]。甚至我们认为，为方便不同的应用，一个 *COOL* 支持多个不同的 *subtyping* 层次是很有必要的。实际上，在多数（顺序的和并发的）面向对象语言中，至少有一个蕴涵的 *subtyping* 层次：即 *supertype* 的实例可接受什么消息，在其 *subtype* 的实例中

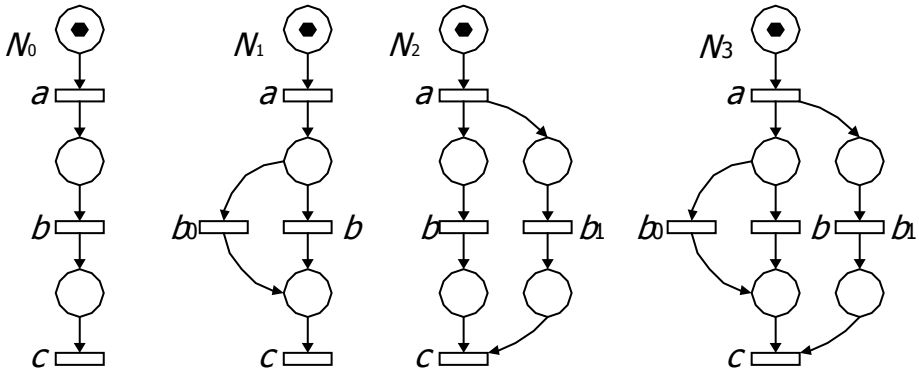


图3.1 ($N_1 \le_{pt} N_0$, $N_2 \le_{pj} N_0$ 及 $N_3 \le_{lc} N_0$)

也可接受同样的消息（这是多态性的一种）。

至于要支持什么样的 *subtyping* 层次关系，应根据应用的目标来确定，这方面人们已有

许多工作[LS94]。作为一个例子，[VB97]中基于封装（*encapsulation*）和抽象（*abstraction*）的思想利用分支双模拟等价（*branching-bisimulation equivalance*）关系定义了 Petri 网间的两个 *subtyping* 关系(原文称为动态行为继承)，分别记为$\leqslant_{pt}$和$\leqslant_{pj}$，前者具有 *blocking* 语义（即限定 *subtype* 的实例只接受其 *supertype* 的实例所能接受的消息），后者具有 *hiding* 语义（即将 *subtype* 的实例中不属于其 *supertype* 的实例所能接受的消息隐藏起来，看作内部活动）。在图 3.1 中，有 $N_1 \leqslant_{pt} N_0$，即在阻止 N_1 中 b_0 发生的前提下，N_1 和 N_0 的动态行为（在分支双模拟的意义下）是一致的；同时 $N_2 \leqslant_{pj} N_0$，即将 N_2 中的 b_1 看作内部活动，N_2 和 N_0 的动态行为是一致的。图中还有 $N_3 \leqslant_{1c} N_0$，等价于 $N_3 \leqslant_{pt} N_0 \vee N_3 \leqslant_{pj} N_0$。

面对不同的 *subtyping* 关系，继承反常的多少是不相同的，[CRR98]的定义中也注意到了这一点。进一步，我们还认为，对不同的 *subtyping* 关系，“渐增式继承”的含义也应有所区分，这既有利于更客观地比较和评价人们提出的各种解决继承反常的方法，也利于在实际开发中更方便地实现相应的 *subtyping* 关系。例如，对于在[VB97]中，针对 *subtyping* 关系$\leqslant_{pt}$、$\leqslant_{pj}$ 和$\leqslant_{1c}$（参见图 3.1）分别给出了相应的继承保持变换规则（*inheritance-preserving transformation rules*），它们都可被包括在相应 *subtyping* 关系的“渐增式继承”集合之中。

3.3 理解继承反常（非形式定义）

本文基于以下观点来理解继承反常现象（读者可结合第 4 节的形式定义一起阅读）：

（1）不同的语言机制有不同的继承反常现象，因此继承反常的定义是针对语言的（可参考第 2 节）。

（2）语言机制确定后，其 *class* 和 *type* 系统就确定了，有什么样的 *inheritance* 规则和什么样的 *subtyping* 关系也确定了。每一个 *class* 都有一个唯一的 *type* 与之对应（可以是多对一），但一个 *type* 不一定有与其对应的 *class* (参见第 4 节中 *type* 函数的定义)。

（3）我们把一个 *inheritance* 规则理解为所有 *class* 的集合上的一种先序（*preorder*）关系。一组 *inheritance* 规则定义了一种 *inheritance* 层次关系，当一个 *class* A 仅采用该组的 *inheritance* 规则可构造出另一个 *class* B，则 A 和 B 之间就具有这种 *inheritance* 层次关系（可对照定义 4.4 中范畴 $\boldsymbol{C}_{\mathrm{R}}$ 的定义）。某个语言中由全部 *inheritance* 规则定义的 *inheritance* 层次关系称为该语言的完整 *inheritance* 层次关系（可对照定义 4.4 中范畴 $\boldsymbol{C}_{\boldsymbol{L}}$ 来理解）。

（4）一种 *inheritance* 层次关系实现某种 *subtyping* 关系，如果任何具有这种 *inheritance* 层次关系的两个 *class* 对应的两个 *type* 之间都具有这种 *subtyping* 关系。

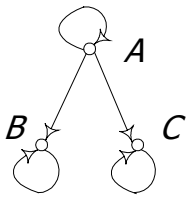
在这里应注意的是：本文中，继承层次关系 *P* 实现 *Subtyping* 关系 *p*，是指存在从关系 *P* 到关系 *p* 的保序映射，不必要是双射（参见定义 4.6，实现函子 **F** 是从 Class 范畴映射到 Type 范畴）。

大多数面向对象的语言都默认如下的 *Subtyping* 关系：*type*（*B*）是 *type*（*A*）的子类型 *iff B* 是 *A* 的子类 （这里指在完整 *inheritance* 层次关系下，*B* 继承 *A*）。若对任何一种继承层次关系都这样定义其相应的 *Subtyping* 关系，自然它就实现这个 *Subtyping* 关系。在实用中，可以对某一种特定的继承层次关系规定一种相应的 *Subtyping* 关系（定义方法类似，也可参考第 5 节例 5.1.1），如[wz88]中所述的 4 种继承关系都可对应各自相应的一种 *Subtyping* 关系。但在本文的理论研究中，不必规定任何一种继承层次关系都实现该语言中的某一 *Subtyping* 关系。

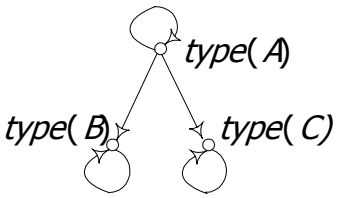
（5）设 P 和 Q 是可以实现某种 *subtyping* 关系 *p* 的两个 *inheritance* 层次关系，且 P 包含 Q。*A* 是任何一个 *class*。若在关系 P 下，*A* 的任何 *subclass B*，都存在与 *B* 有着（对于 *Subtyping* 关系 *p*）相兼容的（即语义行为一致的）*type* 的 *class C*，使得在关系 Q 下 *C* 是 *A* 的 *subclass*，则称在这种 *subtyping* 关系下 *inheritance* 层次关系 P 可继承归约到 *inheritance* 层次关系 Q。（这已经是一个半形式化的叙述了，完全非形式的说法很难准确表述，形式化定义参见定义 4.7）

对于某种 *Subtyping* 关系 *p*，两个 *class B* 和 *C* “具有兼容的 *type*”意味着 *type*（*B*）是 *type*（*C*）（在 *p* 下）的子类型，同时 type（*C*）也是 type（*B*）（在 *p* 下）的子类型。本文中 *Subtyping* 关系是一种先序关系（*pre-order*），而非偏序关系（*partial order*）（前者不要求反对称性），因此 *B* 和 *C* “具有兼容的 *type*”不代表 *type*（*B*）= *type*（*C*）。

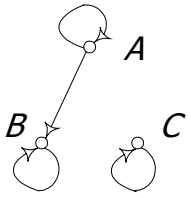
现举例说明两个继承层次之间 “可继承归约到”关系的含义。设继承层次关系 P 和 Q 分别用继承规则集 $R^{/}$ 和 R 来刻画，满足 $R \subseteq R^{/} \subseteq \mathbf{R}_{\boldsymbol{L}}$。这里，用 $\mathbf{R}_{\boldsymbol{L}}=\{r_1, r_2, \ldots, r_n\}$表示语言 $\boldsymbol{L}$ 的继承机制，其中 $r_1, r_2, \ldots, r_n$ 为语言 $\boldsymbol{L}$ 的所有 *inheritance* 规则（参见第 4 节）。不妨设 $R^{/}=\{r_1, r_2\}$，$R=\{r_1\}$。假定对于类 A，B 和 C，有$<A, B>\in r_1$，$<A, C>\in r_2$，这样在继承层次关系 P 和 Q 下，对应于类 A，B 和 C 的关系图可由图 3.3.1（a）和（b）表示。设 P 实现某种 *Subtyping* 关系 *p2*，因为 $R \subseteq R^{/}$，所以 Q 也实现 *Subtyping* 关系 *p2*（参考第 4 节命题 4.1 和命题 4.2）。设图 3.3.1（d）表示在关系 *p2* 下，对应于类型 *type*（A），*type*（B）和 *type*（C）〉的关系图。从该图可知，对于 *Subtyping* 关系 *p2*，B 和 C 具有兼容的 *type*，即满足<*type*（B），*type*（C）$>\in$*p2* 以及<*type*（C），*type*（B）$>\in$*p2*（参考上述讨论）。这个例子中，在关系 P 下，对 A 的子类 C，存在与 C 有（对于 *Subtyping* 关系 *p2* 而言）相兼容的 *type* 的类 B，而在关系 Q 下，B 也是 A 的子类。若这个命题对任何的 A 和 C 都成立，就有：在 *Subtyping* 关系 *p2* 下，继承层次关系 P“可继承归约到”继承层次关系 Q。其实际含义为：对于由 *p2* 所定义的行为继承关系（即 *Subtyping* 关系）而言，继承层次关系 Q 的实现能力足以替代继承层次关系 P 的实现能力。在此例中，继承规则 r_2 只是丰富了实现（由 *p2* 所定义的）行为继承的手段（增加了可用性）。



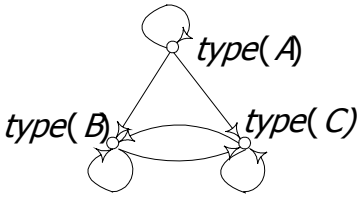
(a) 在继承层次关系 *P* 下



(c) 在 *Subtyping* 关系*p1*下



(b) 在继承层次关系 *Q* 下



(d) 在 *Subtyping* 关系*p2*下

图3.3.1

（6）一种语言机制中可以存在多种 *subtyping* 关系。每一种 *subtyping* 关系都规定一种 *inheritance* 层次关系为其相应的“渐增式继承”关系。这个“渐增式继承”关系一定是可以实现该 *subtyping* 关系的。

这里，“渐增式继承”是一个相对的概念，意指“理想”的继承方式。例如，“只修改分离出来的同步代码而不修改方法体”的继承方式已经足够“理想”了，你可以将这种继承看作是“渐增式继承”；但你也可规定“同步代码和方法体都不修改”的继承方式才是“理想”的“渐增式继承”。本文中，“继承反常”的定义依赖于“渐增式继承”，因而也是一个相对的概念。“渐增式继承” 概念的相对性有利于区分不同的研究对“继承反常”问题的解决程度。

（7）对某一种 *subtyping* 关系，若所有可以实现这种 *subtyping* 关系的 *inheritance* 层次关系都可继承规约到该 *subtyping* 关系的“渐增式继承”关系，则对于这种 *subtyping* 关系，该语言没有继承反常（*anomaly-free*）。（参见定义 4.8（2））

“无继承反常”意味着任何一种 *inheritance* 层次关系可实现的行为继承层次（*subtyping* 关系）都可由“理想”的继承方式（“渐增式继承”）来实现。

（8）若对于某一语言的所有 *subtyping* 关系，该语言都没有继承反常，则这种语言不存在继承反常，否则就存在继承反常。（参见定义 4.9）

（9）对某一种 *subtyping* 关系，“渐增式继承”关系的限制条件越放宽（即“渐增式继承”

关系所包含 *inheritance* 规则越多），继承反常越少（参见命题 4.3）。因为一个 *inheritance* 层次关系都可继承规约到“渐增式继承”关系的可能性增大了。

（10）对某一种 *subtyping* 关系，若加强此关系的条件，而“渐增式继承”关系不变，则继承反常会减少（参见命题 4.4）。因为可实现这种 *subtyping* 关系的 *inheritance* 层次关系变少了。

下一节中将采用范畴论的术语对上述概念进行完全形式化的描述，其中 *inheritance* 层次关系对应一个 Class 范畴，*subtyping* 关系对应一个 Type 范畴。

4. 定义继承反常现象

范畴论[Peirce91][ML71]的观点抽象层次较高，能帮助我们把握继承反常现象的本质，给其一个清晰且精练的形式定义。先介绍本节将用到的三个概念：范畴（*category*）、子范畴（*subcategory*）和函子（*functor*）。

定义 4.1 一个范畴 $\boldsymbol{C}$ 包含：（1）对象（*objects*）集 $\mathrm{ob}\,\boldsymbol{C}$；（2）对每个对象有序对（A，B），相应有一个射（*morphisms*）的集合 $\mathrm{hom}_{\boldsymbol{C}}$（A，B）（若可从上下文得知 $\boldsymbol{C}$，可简记为 hom(A，B)），其中的射具有论域（*domain*）A 和余论域（*codomain*）B；（3）对任何对象三元组（A，B，C），有一个从 hom(A,B) × hom(B,C)到 hom(A,C)的映射：（f,g）→ $g\,f$。$\boldsymbol{C}$ 中的对象和射满足以下条件：(C1) 若（A,B）≠（C,D），则 hom(A,B)和 hom(C,D)不相交；(C2) 若 $f\in$ hom(A,B)，$g\in$hom(B,C)，$h\in$hom(C,D)，则 $(hg)f=h(gf)$；(C3) 对应每个对象 A 都有一个恒等射 $\mathbf{1}_{\mathrm{A}}\in$hom(A,A)，使得对任何 $f\in$hom(A,B)，有 $f\mathbf{1}_{\mathrm{A}}=f$，对任何 $g\in$hom(B,A)，有 $\mathbf{1}_{\mathrm{A}}\,g=g$。

定义 4.2 范畴 $\boldsymbol{D}$ 是范畴 $\boldsymbol{C}$ 的一个子范畴，当且仅当（1）$\mathrm{ob}\,\boldsymbol{D}\subseteq\mathrm{ob}\,\boldsymbol{C}$；（2）对所有 A,B∈ ob $\boldsymbol{D}$，有 $\mathrm{hom}_{\boldsymbol{D}}$(A,B) ⊆ $\mathrm{hom}_{\boldsymbol{C}}$(A,B)；（3）对 A∈ob $\boldsymbol{D}$，和 $\mathbf{1}_{\mathrm{A}}\in\mathrm{hom}_{\boldsymbol{C}}$(A,A)有 $\mathbf{1}_{\mathrm{A}}\in\mathrm{hom}_{\boldsymbol{D}}$(A,A)；（4）对 $f\in\mathrm{hom}_{\boldsymbol{D}}$(A,B)，$g\in\mathrm{hom}_{\boldsymbol{D}}$(B,C) 及 $g\,f\in\mathrm{hom}_{\boldsymbol{C}}$ (A,C)，有 $g\,f\in\mathrm{hom}_{\boldsymbol{D}}$ (A,C)。若 $\boldsymbol{D}$ 是范畴 $\boldsymbol{C}$ 的子范畴，同时 ob $\boldsymbol{D}$ = ob $\boldsymbol{C}$，则 $\boldsymbol{D}$ 称为范畴 $\boldsymbol{C}$ 的宽子范畴（*wide subcategory*）。

定义 4.3 设 $\boldsymbol{C}$，$\boldsymbol{D}$ 为范畴，从 $\boldsymbol{C}$ 到 $\boldsymbol{D}$ 的一个函子 **F** 包含：（1）一个从 ob $\boldsymbol{C}$ 到 ob $\boldsymbol{D}$ 的映射 A → **F**A；（2）对任何 A,B∈ob $\boldsymbol{C}$，从 $\mathrm{hom}_{\boldsymbol{C}}$ (A,B)到 $\mathrm{hom}_{\boldsymbol{D}}$ (A,B)有一个映射 f → **F**(f)。**F** 应满足以下条件（C1）如果 gf 定义在 $\boldsymbol{C}$ 中，则 $\mathbf{F}(gf)=\mathbf{F}(g)\,\mathbf{F}(f)$；（C2）$\mathbf{F}(\mathbf{1}_{\mathrm{A}})=\mathbf{1}_{\mathbf{FA}}$。

设（并发）面向对象语言 $\boldsymbol{L}$ 中，所有 *class* 的集合为 $\mathbf{Class}_{\boldsymbol{L}}$，所有 *type* 的集合为 $\mathbf{Type}_{\boldsymbol{L}}$。每一 $A\in\mathbf{Class}_{\boldsymbol{L}}$ 是产生实例的一个模板，用 *instances*(A) 表示 A 的所有实例的集合；对 $T\in\mathbf{Type}_{\boldsymbol{L}}$，*class* A 实现 T 当且仅当 $\forall a\in instances(A).a\in T$。记 $type(A)=\cap\{T\mid A\text{ 实现 }T\}$（参见[CRR98]），这里∩为最大下界算符。这里我们约定：对任何 $A\in\mathbf{Class}_{\boldsymbol{L}}$，都有 $type(A)\in\mathbf{Type}_{\boldsymbol{L}}$。

用继承规则集 $\mathbf{R}_{\boldsymbol{L}}=\{r_1,r_2,\ldots,r_{\mathrm{n}}\}$表示 $\boldsymbol{L}$ 的继承机制，其中每一 $r_{\mathrm{i}}\subseteq\mathbf{Class}_{\boldsymbol{L}}\times\mathbf{Class}_{\boldsymbol{L}}$ 为一先序（*preorder*）关系，即满足自反性和传递性。$\boldsymbol{L}$ 的所有 *subtyping* 关系表示为集合 $\mathbf{P}_{\boldsymbol{L}}=\{p_1,p_2,\ldots,p_{\mathrm{m}}\}$，其中每一 $p_{\mathrm{i}}\subseteq\mathbf{Type}_{\boldsymbol{L}}\times\mathbf{Type}_{\boldsymbol{L}}$，也是先序关系。对任何 $p\in\mathbf{P}_{\boldsymbol{L}}$，我们把相应于 p 的渐增式继承用继承规则集 $\mathrm{R}_p\subseteq\mathbf{R}_{\boldsymbol{L}}$ 表示。

定义 4.4 设 $\mathrm{R}\subseteq\mathbf{R}_{\boldsymbol{L}}$，范畴 $\boldsymbol{C}_{\mathrm{R}}$ 定义为：（1）$\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}}=\mathbf{Class}_{\boldsymbol{L}}$；（2）对所有 A，$B\in\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}}$，$hom$（$A$, B）是由下列算法确定的最小集：（ⅰ）若存在 $r\in$R，使（A, B）$\in r$，则 $r\in hom(A,B)$；（ⅱ）若对 $C\in\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}}$ 存在 $r_1\in hom(A,C)$，$r_2\in hom(C,B)$，则 $r_2\,r_1\in hom(A,B)$（ⅲ）反复使用（ⅰ）、（ⅱ）。易验证 $\boldsymbol{C}_{\mathrm{R}}$ 符合定义 4.1。称 $\boldsymbol{C}_{\mathrm{R}}$ 为一个基于 $\boldsymbol{L}$ 的 Class 范畴（在不致混淆的情形下，简称为 Class 范畴）。若 R= $\mathbf{R}_{\boldsymbol{L}}$，则称 $\boldsymbol{C}_{\mathrm{R}}$ 为基于 $\boldsymbol{L}$ 的完全 Class 范畴（对应语言 $\boldsymbol{L}$ 的完整 *inheritance* 层次关系），记为 $\boldsymbol{C}_{\boldsymbol{L}}$。

命题 4.1 设 $\mathrm{R}\subseteq\mathbf{R}_{\boldsymbol{L}}$，$\mathrm{R}'\subseteq\mathrm{R}$，R 定义的 Class 范畴为 $\boldsymbol{C}_{\mathrm{R}}$，$\mathrm{R}'$ 定义的 Class 范畴为 $\boldsymbol{C}_{\mathrm{R}'}$，则 $\boldsymbol{C}_{\mathrm{R}'}$是 $\boldsymbol{C}_{\mathrm{R}}$ 的子范畴，并称之为 $\boldsymbol{C}_{\mathrm{R}}$ 的子 Class 范畴。

证明：由定义 4.4，对任何 $A,B\in\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}}=\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}'}$，$hom_{\boldsymbol{C}_{\mathrm{R}'}}(A,B)\subseteq hom_{\boldsymbol{C}_{\mathrm{R}}}(A,B)$。

推论 4.1 任何 $\mathrm{R}\subseteq\mathbf{R}_{\boldsymbol{L}}$ 定义的 Class 范畴 $\boldsymbol{C}_{\mathrm{R}}$ 都是 $\boldsymbol{C}_{\boldsymbol{L}}$ 的子 Class 范畴。

定义 4.5 设 $p\in\mathbf{P}_{\boldsymbol{L}}$，定义范畴 $\boldsymbol{T}_{\mathrm{p}}$ 为：（1）$\mathrm{ob}\,\boldsymbol{T}_{\mathrm{p}}=\mathbf{Type}_{\boldsymbol{L}}$；（2）对任何 A，$B\in\mathrm{ob}\,\boldsymbol{T}_{\mathrm{p}}$，$(A,B)\in hom_{\boldsymbol{T}_{\mathrm{p}}}(A,B)$ 当且仅当 $(A,B)\in p$。称 $\boldsymbol{T}_{\mathrm{p}}$ 为一个 Type 范畴，它实际上也是一个先序集范畴。

定义 4.6 设 $\boldsymbol{C}_{\mathrm{R}}$ 为一个 Class 范畴，$\boldsymbol{T}_{\mathrm{p}}$ 为一个 Type 范畴，**F** 为满足以下条件的映射（从 ob $\boldsymbol{C}_{\mathrm{R}}$ 到 ob $\boldsymbol{T}_{\mathrm{p}}$）：对任何 $A\in\mathrm{ob}\,\boldsymbol{C}_{\mathrm{R}}$，$\mathbf{F}A=type(A)$。若 **F** 是从 $\boldsymbol{C}_{\mathrm{R}}$ 到 $\boldsymbol{T}_{\mathrm{p}}$ 的函子（参考定义 4.3），则称 $\boldsymbol{C}_{\mathrm{R}}$ 实现 $\boldsymbol{T}_{\mathrm{p}}$，**F** 为实现函子。（参考图 4.1）

命题 4.2　设 $C_{R/}$ 是 C_R 的子 Class 范畴，C_R 实现 T_p，则 $C_{R/}$ 也实现 T_p。
证明：$C_{R/}$ 实现 T_p 可使用与 C_R 实现 T_p 相同的实现函子。
定义 4.7　设 C_R 是 $C_{R/}$ 的子 Class 范畴，T_p 为一个 Type 范畴，$C_{R/}$ 实现 T_p (由命题 4.2，C_R 也实现 T_p)，**F** 为实现函子。称对于 Type 范畴 T_p，$C_{R/}$可继承归约到 C_R，记为 $C_{R/} \Rightarrow_p C_R$，当且仅当对任何 $A, B' \in \text{ob } C_R = \text{ob } C_{R/}$，若存在 $f' \in hom_{C_{R/}}(A, B')$，则存在 $f \in hom_{C_R}(A, B)$，使得$(\mathbf{F}B, \mathbf{F}B') \in hom_{T_p}(\mathbf{F}B, \mathbf{F}B') \wedge (\mathbf{F}B', \mathbf{F}B) \in hom_{T_p}(\mathbf{F}B', \mathbf{F}B)$。（参考第 3.3 节（5）及图 4.2）
注：$(\mathbf{F}B, \mathbf{F}B') \in hom_{T_p}(\mathbf{F}B, \mathbf{F}B')$意味着 type（$B'$）是 type（$B$）（在 p 下）的子类型，而$(\mathbf{F}B', \mathbf{F}B) \in hom_{T_p}(\mathbf{F}B', \mathbf{F}B)$意味着 *type*（$B$）是 *type*（$B'$）（在 p 下）的子类型。$(\mathbf{F}B, \mathbf{F}B') \in hom_{T_p}(\mathbf{F}B, \mathbf{F}B') \wedge (\mathbf{F}B', \mathbf{F}B) \in hom_{T_p}(\mathbf{F}B', \mathbf{F}B)$说明对于 *Subtyping* 关系 p，B 和 B' “具有兼容的 *type*”。

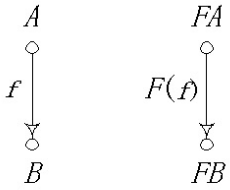


图4.1

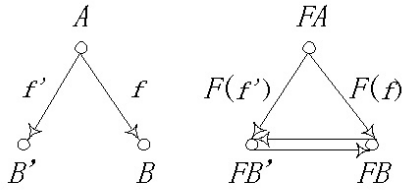


图4.2

定义 4.8　（1）设 $R \subseteq \mathbf{R}_L$ 且 C_R 实现 T_p。若对于任何 $C_{R/}$，$C_{R/}$ 实现 T_p，都满足 $C_{R/} \Rightarrow_p C_R$，则称在继承规则集 R 作用下，对于由 p 确定的子类型关系，L 无继承反常（*anomaly-free*）。
（2）若在渐增式继承规则集 R_p 作用下，对于由 p 确定的 *subtyping* 关系，L 无继承反常，则称对于由 p 确定的子类型关系，L 无继承反常。
命题 4.3　设 $R \subseteq R' \subseteq \mathbf{R}_L$。若对于由 $p \in \mathbf{P}_L$ 确定的 *subtyping* 关系，$C_{R/}$ 实现 T_p，L 在继承规则集 R 作用下无继承反常，则在 R' 作用下也无继承反常。
证明：对任何 $R'' \supseteq R'$，且 $C_{R''}$ 实现 T_p ($C_{R/}$，C_R 也实现 T_p)，易从 $C_{R''} \Rightarrow_p C_R$ 证得 $C_{R''} \Rightarrow_p C_{R/}$。
命题 4.4　设 $p, p' \in \mathbf{P}_L$，且 $p \subseteq p'$。若在继承规则集 $R \subseteq \mathbf{R}_L$ 作用下，L 对于由 p' 确定的 *subtyping* 关系无继承反常，则 L 对于由 p 确定的 *subtyping* 关系也无继承反常。
证明：对任何 R'，且 $C_{R/}$ 实现 T_p ($C_{R/}$ 也必然实现 $T_{p'}$)，易从 $C_{R/} \Rightarrow_{p'} C_R$ 证得 $C_{R/} \Rightarrow_p C_R$。
定义 4.9　语言 L 无继承反常，当且仅当对任何 $p \in \mathbf{P}_L$ 都有：对于由 p 确定的 *subtyping* 关系，L 无继承反常。若不满足该条件，则语言 L 存在继承反常。

5. 解释

本节首先通过几个典型的继承反常现象的例子说明本文定义的普适性，并有助于进行深层次的理解，然后对并发面向对象技术中继承性的建模提出相关的建议。

5.1 解释继承反常现象

例 5.1.1 在谈到顺序面向对象语言时，一般不涉及继承反常现象。这是因为，任一这样的语言 L 都默认一个特殊的 *subtyping* 关系，它对应的 Type 范畴记为 T_L，满足 C_L 实现 T_L，实现函子为 **F**。这里，**F** 满足$\forall A$（$A \in \text{ob } C_L \rightarrow \mathbf{F}A = type(A)$）。由定义 4.7、4.8，在继承规则集 $\mathbf{R}_L$ 作用下，对于上述 *subtyping* 关系，L 无继承反常（$\mathbf{R}_L$ 本身被默认为这种 *subtyping* 关系的渐增式继承规则）。因为顺序面向对象语言不提供定义 *subtyping* 关系的机制，所以可以认为这种默认的 *subtyping* 关系是语言中唯一的 *subtyping* 关系，由定义 4.9，这些语言 L 无继承反常。
例 5.1.2 在图 2.1 中，由类 Buffer 构造子类 GBuffer 和子类 LockableBuffer 时会产生 History-Sensive 式的继承反常现象[MY93]。按照本文的观点，这种继承反常所针对的 *subtyping* 关系可认为是关系 **P1**（在下面定义），其渐增式继承是最严格的一种，即子类中对于被继承的父类代码不可作任何形式的修改。但如果我们面对的 *subtyping* 关系是关系 **P2**（在下面定义），而不是关系 **P1**，则上述继承反常就被“绕过”，因为从类 Buffer 到子类 Gbuffer 或子类 LockableBuffer 的 Class 层次不能实现 *subtyping* 关系 **P2**。**P1** 和 **P2** 分别具有 *blocking* 和 *hiding* 的语义（即 *encapsulation* 和 *abstraction* 的语义[VB97]）。

Subtyping 关系 **P1**：*Subtype* 实例接受消息串的能力不亚于 *supertype* 实例，即后者可接

受什么样的消息串，前者也可接受；同时一个前者可接受的消息串，除非包含后者未定制的消息，否则后者也可接受该消息串。

Subtyping 关系 **P2**：（1）*Subtype* 实例接受消息串的能力不亚于 *supertype* 实例，即后者可接受什么样的消息串，前者也可接受；（2）对任何后者可接受的消息串 α，及前者可接受的消息串 β（α 与屏蔽掉前者未定制的消息后的 β 相等），设前者在接受消息串 β 后下一个可接受的消息的集合 R_β，而后者在接受消息串 α 后下一个可接受的消息的集合 R_α，则满足在 R_β中将前者未定制的消息去掉后与 R_α相等。

例 5.1.3 若将图 2.1 中每个 *method* 的 *guard* 部分与其功能部分的代码分离开，则 LockableBuffer 中 *get* 和 *put* 的功能部分可直接从 Buffer 继承下来，不必作任何修改。一般来说，将同步代码分离可减少继承反常。进一步，根据本文的定义，若将“可对分离出来的同步代码进行修改”作为一条规则添入“渐增式继承”规则集中，则某些继承反常可以“消失”。如对图 2.1，若在“渐增式继承”规则集中加入“可对 *guard* 部分进行修改”，则 LockableBuffer 对 Buffer 的继承就无反常现象。

5.2 继承性的建模

本文的观点对于并发面向对象语言（包括规范语言）设计及软件开发中继承性的建模都有相关的启示。

首先，在“并发面向对象语言中应将 *inheritance* 层次和 *subtyping* 层次区别对待”的共识下，我们强调了继承反常这一概念也应有针对性，对不同的 *subtyping* 关系应考虑其特有的渐增式继承方法。*Subtyping* 关系愈强，其行为规范的能力就愈强，但对渐增式继承方法的限制愈强，即渐增式继承的规则愈少，可应用性也愈小。从继承反常的角度来看，对于较强的 *subtyping* 关系，继承反常较少（命题 4.4），但较少的渐增式继承规则却会带来较多的继承反常（命题 4.3）。因此，*subtyping* 关系的设计应根据实际情形权衡利弊。

其次，在并发面向对象的软件开发中，应针对每一种 *subtyping* 关系分别研制其“渐增式继承”的设计方法、工具，增强它们的可应用性，减小开发人员的负担。例如，对于图 3.1 中的 *subtyping* 关系 $\leqslant_{pt}$、$\leqslant_{pj}$ 及 $\leqslant_{lc}$，[VB97] 设计出一些便于应用的 *Inheritance-Preserving-Transformation* 规则（以下简称 *IPT* 规则）。若将这些 *IPT* 规则与本文所述的“渐增式继承”规则相对应，可以得出结论：*IPT* 规则开发得越多，应用者避开继承反常的机会就越多，同时相应 *subtyping* 关系的可应用性越好。

6. 结语

本文形式化地给出了“继承反常现象”的一种一般性定义。定义对每一种 *Subtyping* 关系都有其相对应的“渐增式继承”，使定义更具有广泛性。文中“渐增式继承”是一个相对的概念，这有助于对人们“使继承反常现象得到缓解”的努力得以分类和评价。除了有助于人们更深刻理解“继承反常”的含义之外，文中一些相关的结论对于并发面向对象语言（包括规范语言）设计及软件开发中继承性的建模有一定的启示。

参考文献

[ABSB94] M.Akis, J.Bosch, W. Vander Sterren, L.Bergmans.. Real-time Specification Inheritamce Anomalies and Real-time Filters. In ECOOP’94, M.Jorovo, R.Pareschi(Eds.), LNCS 821, Springer-Verlag, 1994.

[Agha86] G.Agha, Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1986.

[America87] P.America. Inheritance and Subtyping in a Parallel Object-oriented Language. In Proc. of ECOOP’97 LNCS 276, 234-242, Springer-Verlag,1987.

[America90] P.America. A Paralel Object-oriented Language with Inheritance and Subtyping. In Proc. of OOPSLA’90, Vol.25: 161-168, SIGPLAN Notices, ACM Press, 1990.

[America91] P.America. Designing an Object-oriented Programming Language with Behavioural Subtyping. In Proc. of REX School/Workshop on Foundations of Object-oriented Languages(REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990, LNCS 489, 60-90, Springer-verlag, 1991.

[BCC95] E.Batiston, A,Chizzoni, Fiorella De Cindo. Inheritance and Concurrency in CLOWN. In Proceedings of the “Application and Theory of Petri Net 1995” workshop on “object-oriented programs and models concurrency”, Torino, Italym 1995.

[Biberstein97] Oliver Biberstein. CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems. PhD thesis, University of Geneva, April 1997.

[CRR98] Lobel Crnogorac, Amand S.Rao, and Kotagiri Ramamohanarao. Classifying Inheritance Mechanisms In Concurrent Object-oriented Programming. LNCS 1445, ECOOP'98——Object-oriented Programming, Springer-Verlag, pp572-600, 1998.
[KL89] Dennis G.Kafura and Keung Hae Lee. Inheritance in Actor Based Concurrent Object-oriented Languages. In Proc. of ECOOP'89, 131-145, Cambridge University Press, 1989.
[Lakos95] Charles Lakos. From Coloured Petri Nets to Object Petri Nets. Proceedings of 16th Int. Conference on the Application and Theory of Petri Nets, LNCS 935, Turin, Italy, Springer-Verlag,1995.
[LLNW96] U.Lechner, C.Lengauer, F.Nickl and M.Wirsing. (Objects + Concurrency)&Reuability——A Proposal to Circumvent the Inheitance Anomaly. LNCS 1098, ECOOP'96——OOP, Pierre Cointe(Ed.), Linz, Austria, Springer-Verlag, pp.222-247, 1996.
[LS94] B.Liskov and J.M.Wing. A behavioural notion of subtyping. ACM Transaction on Programming Languages and Systems. Vol. 16, No.6. PP1811-1841, Nov., 1994.
[Meseguer93] José Meseguer. Solving the Inheritance Anomaly in Concurrent Object-oriented Programming. In ECOOP'93——Object-oriented Programming, LNCS 707, pp 220-246, Springer-Verlag, 1993.
[ML71] S. Mac Lane. Categories for the Working Mathematician, Springer-Verlag, 1971.
[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In Reserch Directions in Concurrent Object-oriented Programming, edited by G.Agha, P.Wegner and A.Yonezawa, The MIT Press, pp.107-150, 1993.
[Peirce91] B.C.Peirce. Basic Category Theory for Computer Scientists, MIT Press, 1991.
[VB97] W.M.P. Vander Aalst and J.Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. 18th International Conference on the Application and Theory of Petri Nets, Lecture Notes in Computer Science 1248, pp 62-81, Toulouse, France, Springer-Verlag, 1997.
[WYY01] S.Y.Wang, J.Yu and C.Y.Yuan. A Pragmatic Behavior Subtyping Relation Based on Both States and Actions. To appear in Journal of Computer Science and Technology.
[WZ88] P.Wegner, S.B.Zdonik. Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like. In ECOOP'88 Proceedings. Lecture Notes in Computer Science 322, pp 55-77, Springer-Verlag, 1988.
[Yonezawa90] Akinori Yonezawa, editor. ABCL: An Object-oriented Concurrent System. Computer Systems Series. The MIT Press, 1990.

Inheritance Anomaly in Concurrent Object-orientation

Sheng-yuan Wang Liang-huai Yang Chong-yi Yuan Ping Yang

Abstract. The combination of concurrency and object orientation is definitely natural except for inheritance. One of the interference between inheritance and concurrency is *inheritance anomaly*. Although have been researched extensively, inheritance anomalies are still only vaguely defined and often misunderstood, and almost no formal work has been done. This paper sets forth a new viewpoint of understanding inheritance anomalies, in which each *subtyping relation* would have its specific *incremental inheritance*. Related concepts and definitions are formalized through the language of Category. Some issues of the paper are well adapted to distinguish and explain different standpoints about inheritance anomalies, and can serve as guidelines in the modeling of inheritance.

Key Words. Concurrency；Object Orientation；Inheritance Anomaly；Incremental Inheritance；Category Theory