一种并发面向对象同步模型研究*

吕 建, 杨大军, 廖 宇, 唐 宝

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093);
(南京大学 计算机软件研究所,江苏 南京 210093)
E-mail: lj@nju.edu.cn
http://moon.nju.edu.cn

摘要: 同步是并发语言的一个重要特征,然而在面向对象的框架下,同步约束的存在可能导致不期望的方法的重定义.基于并发面向对象规约语言 VDM++(Vienna development method)的两种同步机制,提出一种适合于并发面向对象广谱规约语言的同步模型-卫式路径结构.它不但可以支持一般代码的复用,而且可以全方位地支持对同步代码的复用.

关 键 词: 并发面向对象;继承;同步;继承异常
中图法分类号: TP311 文献标识码: A

通常并发面向对象语言的设计主要是基于正交模型、同构模型和异构模型这 3 种对象模型[1].因为异构模型易于直接模拟客观世界并且能使设计者在单一的模块(对象)下描述问题,所以,异构模型逐渐成为并发面向对象规约语言的主要模型.在异构模型中通常存在主动对象和被动对象两种对象.由于被动对象可能同时为多个主动对象提供服务,所以必须提供相应的同步机制,以保证被动对象数据的一致性.但是,同步机制的引入在面向对象的框架下会带来继承异常问题[2].因此,同步模型是设计并发面向对象规约语言的一个重要问题.一个好的同步模型应该既可以满足各种同步需求,又可以尽可能地解决继承异常问题,对并发软件的复用提供较好的支持.

依据同步限制规约与功能性规约的关系,同步模型可以分为内置式和外挂式.内置式是指同步限制规约嵌入在功能性规约中,共同组成方法体.外挂式是指同步限制规约与功能性规约分离,从而使同步代码独立于具体方法的定义.由于内置式同步模型极易带来继承异常问题,而外挂式同步控制则很好地体现了将对象的功能特性与行为特性相分离的思想,其语义清晰,易于表达同步规约,同时同步代码与方法体分离能够很好地缓解继承异常问题,因此,在面向对象的框架下,基于外挂式的同步模型是合理的选择.

VDM++[3]是典型的基于外挂式同步模型的并发面向对象语言.它提供了路径(trace)机制和允用谓词(permission predicates)机制两种同步机制.其他外挂式的同步模型还有 Kafura 等人[4]提出的行为抽象模型、Rosette 提出[5]的 enabled sets,Neusius[6]提出的 Synchronizing Actions,Frølund[7]提出的 prevents 以及 Matsuoka[8]提出的基于方法集、同步器和转换规约的同步模式等.

在设计同步模式时,我们选择了 VDM++中的同步模型作为出发点,这主要是因为其中的路径同步机制结构优美,易于理解且理论基础比较好.但是,经过仔细分析发现,该同步机制不能很好地解决继承异常问题.因此,

* 收稿日期: 2000-03-17; 修改日期: 2000-07-31
基金项目: 国家自然科学基金资助项目(69873021);国家 863 高科技发展计划资助项目(863-306-ZT02-02-03);国家杰出青年基金资助项目(61525204);江苏省应用基础研究基金资助项目(BJ99016)

作者简介: 吕建(1960－),男,江苏南京人,博士,教授,博士生导师,主要研究领域为软件自动化,并行程序形式化方法,面向对象语言和环境;杨大军(1972－),男,辽宁锦州人,博士,主要研究领域为并发面向对象语言,形式化方法;廖宇(1974－),男,辽宁铁岭人,硕士,主要研究领域为并发面向对象语言,形式化方法;唐宝(1974－),男,宁夏银川人,硕士,主要研究领域为并发面向对象语言,形式化方法.

我们以 VDM++中的路径机制为基础,从同步机制的表达能力,同步代码的分离、分解和独立等不同角度出发,提出了一种新的同步模型.

1 VDM++中同步机制的分析

VDM++的同步机制是基于 DRAGOON Ada 的扩充版、POOL(parallel object oriented language)语言中的同步机制[9]和 Trace 理论[10].

1.1 路径和允用谓词

路径和允用谓词是 VDM++使用的两种同步机制.路径同步机制的前身是路径表达式(path expression).其核心是正则路径表达式.路径表达式的算子是方法名,它包括中缀操作符“;”和后缀操作符“*” “+”.其中,“;”表示路径执行的顺序关系,“*”表示零次或多次重复执行某条路径,而“+”表示 1 次或多次重复顺序执行某条路径.允用谓词同步机制弥补了路径机制只能描述顺序控制的不足,它针对每个方法给出其执行的必要但非充分条件,从而可以描述和限制对象内部的并行.关于其详细信息参见文献[3].

1.2 VDM++中同步机制的不足

通过分析发现,VDM++中的两种同步机制具有以下不足:

(1) 路径机制表达能力的有限性.有限性体现在两个方面:首先,Trace 很难表达与对象状态有关的同步问题,例如对于有界缓冲区问题,即使能够表达,其表达式的长度也将非常大[8];其次,它不能表达对象内部的并发问题.

(2) 允用谓词机制的局限性.虽然允用谓词比路径机制的表达能力强,但其丧失了路径机制直观、易于理解和使用的优点,特别是在表达顺序和互斥方面.

(3) 两者不能互补.VDM++中的这两种机制并不具有抽象与具体的关系,它们处于软件开发的同一层次,但是不能共存在一个类中,因此两者不能互补.

(4) 独立性不强.尽管这两种机制可以通过继承机制将某种同步模式用于不同的类,但是,由于缺少必要的辅助设施而不能将某种同步模式用于已存在的顺序类中.

针对上面的问题,我们从表达能力、同步代码的分解和分离 3 个方面对 VDM++中的路径机制进行扩充,从而使其成为具有强表达能力的易于复用的同步机制.这 3 个方面将在下文详细加以介绍.

2 卫式-路径表达式

为了增强路径表达式的表达能力,我们首先对路径表达式的原子项进行修改,使其能表达与状态相关的问题;其次对路径表达式的操作符进行扩充,使其能表达内部并行.下面详细介绍这种同步机制最重要的部分——卫式-路径表达式.

2.1 卫式-路径表达式的语法

我们可以将 Guard-Trace 表达式递归定义如下:

· 〈Bm,m〉是 Guard-Trace 表达式,其中 m,Bm 均为方法名,且 Bm 方法的返回值为布尔型,我们将其称为条件方法;

· 若 gt1,gt2 是 Guard-Trace 表达式,则 gt1;gt2 是 Guard-Trace 表达式;

· 若 gt 是 Guard-Trace 表达式,则 gt*是 Guard-Trace 表达式;

· 若 gt 是 Guard-Trace 表达式,则 gt#是 Guard-Trace 表达式;

· 若 gt1,gt2 是 Guard-Trace 表达式,则 gt1|gt2 是 Guard-Trace 表达式;

· 若 gt1,gt2 是 Guard-Trace 表达式,则 gt1||gt2 是 Guard-Trace 表达式;

其中,操作符的优先级为‘*’=‘#’>‘;’>‘|’>‘||’.

2.2 卫式-路径表达式的非形式语义

为了描述方便,我们先给出以下定义.

定义 **1(Gt1** 与 **Gt2** 在 **Gt** 中是并行的**)**. 设 Gt 是 Guard-Trace 表达式,如果存在 Gt 的两个子表达式 Gt1′和 Gt2′,Gt1 和 Gt2 分别是 Gt1′和 Gt2′的子表达式,且满足下面条件之一,则称 Gt1 与 Gt2 在表达式 Gt 中是并行的:

· Gt1′||Gt2′是 Gt 的子表达式;

· Gt1′=Gt2′且(Gt1′)#是 Gt 的子表达式.

定义 **2(**包含 **Gt** 的顺序 **Guard-Trace** 表达式**)**. Gt 是 Guard-Trace 表达式,则包含 Gt 的顺序 Guard-Trace 表达式递归定义如下:

· Gt 是包含 Gt 的顺序 Guard-Trace 表达式;

· 若 Gt1 为 Guard-Trace 表达式,Gts 是包含 Gt 的顺序 Guard-Trace 表达式,则 Gt1;Gts 和 Gts;Gt1 是包含 Gt 的顺序 Guard-Trace 表达式;

· 若 Gts 是包含 Gt 的顺序 Guard-Trace 表达式,Gt1 是 Guard-Trace 表达式,则 Gt1|Gts 是包含 Gt 的顺序 Guard-Trace 表达式;

· 若 Gts 是包含 Gt 的顺序 Guard-Trace 表达式,则 Gts*是包含 Gt 的顺序 Guard-Trace 表达式;

只有当有限次使用上面 4 步后得到的 Guard-Trace 表达式才是包含 Gt 的顺序 Guard-Trace 表达式.

下面非形式地描述 Guard-Trace 表达式的操作语义:

· ⟨Bm,m⟩.当存在 m 方法的调用时,执行 Bm,若其结果为真,则执行方法 m,当 m 执行完以后,⟨Bm,m⟩原子项被成功地执行;否则,检测失败.

· gt1;gt2 表示在成功执行 gt1 的情况下,继续 gt2 的执行;若发生检测失败,则最大的包含 gt1;gt2 的顺序 Guard-Trace 表达式被挂起.当 gt1,gt2 均被成功地执行,此表达式被成功地执行.

· gt1|gt2 表示选择 gt1 执行或选择 gt2 执行.若当前调用的方法同时满足两个表达式时,则任意择其一执行;若不被任意一个满足,则挂起最大的包含此表达式的顺序 Guard-Trace 表达式.当被选择的表达式成功地执行完毕,则 gt1|gt2 被成功地执行完毕.

· gt1||gt2 表示 gt1 和 gt2 可被并行执行.若当前调用的方法同时满足两条表达式所示的执行顺序及条件,则任选其一执行该方法.若均不满足,则挂起.当且仅当 gt1 与 gt2 所示的路径均执行完毕时,gt1||gt2 所示的路径执行完毕.

· gt*表示 gt 所示的路径可以连续重复 0 次或多次.只有当调用的方法不满足与其并行的所有路径时,gt*才执行完毕.

· gt#表示可以存在 0 个或多个 gt 所示的路径并行执行.只有当调用的方法不满足与其并行的所有路径时,gt#执行完毕.

2.3 卫式-路径表达式的语用

定义 **3(**不相交性**)**. 设 var(m)为方法 m 所存取的变量的集合,change(m)为方法 m 所修改的变量的集合,则算子⟨Bm1,m1⟩与⟨Bm2,m2⟩是不相交的,当且仅当以下条件成立:

$$(change(m1)\cup change(Bm1))\cup(var(m2)\cup var(Bm2))=\varnothing \text{且}$$

$$(var(m1)\cup var(Bm1))\cup(change(m2)\cup change(Bm2))=\varnothing.$$

尽管在使用卫式-路径表达式时没有强制性的限制,但为了正确地描述方法间的同步,应该遵循以下准则:

· 原子项中的条件方法对实例变量只能进行读操作而不能对其进行修改.这是因为条件方法的目的仅是识别当前对象的状态.

· 原子项的条件方法定义应尽可能少地使用实例变量.这是因为一方面要增加其独立性,另一方面又要减少干扰的可能性.

· 对于一原子项⟨m,Bm⟩,如果存在与其并行的原子项,则它们应满足不相交性.这样可以避免干扰和二义性.

· 对象对外可见的方法均需出现在表达式中,否则客户对象无法得到相应的服务.因为方法执行严格遵循

卫式-路径表达式规定的顺序.

2.4 卫式-路径表达式的代数性质

Guard-Trace 除了符合 Trace 所符合的'|'的结合律和交换律以及"|"在";"上的分配律以外,还符合 " || " 的结合律和交互律、||在|上的分配及以下条件分解规则:

$$(\langle c1,a\rangle;gt1)|(\langle c2,a\rangle;gt2)=(\langle c1\wedge\neg c2,a\rangle;gt1)|(\langle c1\wedge c2,a\rangle;(gt1|gt2))|(\langle\neg c1\wedge c2,a\rangle;gt2);$$

$$(\langle c1,a\rangle \parallel gt1)|(\langle c2,a\rangle\|gt2)=(\langle c1\wedge\neg c2,a\rangle\|gt1)|(\langle c1\wedge c2,a\rangle\|(t1|t2))|(\langle\neg c1\wedge c2,a\rangle\|gt2)$$

和合并规则 T#||T#=T#.

值得注意的是,"||"对";"不具有分配律.这些代数性质便于我们将 Guard-Trace 表达式进行等价转换,以消除其非确定性.

3 卫式-路径结构

卫式-路径表达式结合了路径表达式和允用语句的优点.然而卫式-路径表达式还不能完全解决同步代码的复用问题,因为根据文献[1,2]的分析,它没能很好地解决同步代码的分解和独立问题.本节主要阐述如何对其进行分解.

3.1 卫式-路径表达式的扩充

在上一节,基于 VDM++中的 Trace 表达式和允用谓词,我们提出了 Guard-Trace 表达式.它具有易于规约,便于理解和表达能力强的特点.然而实践证明,同步控制的独立性可通过下面的扩充进一步加强.

3.1.1 扩 充

为了达到上述目的,我们对卫式-路径表达式的项作了如下修改:

· 提出了控制状态空间的概念,它是由控制变量的值所组成的.控制变量只能由同步控制部分存取.

· 为了增强规约的易读性,提出了控制方法的概念.它或者是未定义方法体的抽象方法,或者是仅存取控制变量的方法.

· 方法执行的条件不仅仅限制为真值方法,而且可为一般的布尔表达式,当同步控制部分定义为一个独立的控制类*时,组成这个布尔表达式的基本项只能是控制方法名和抽象控制变量;但是,在完全类**(即定义了同步代码又定义功能代码的类)中,还可以包括非控制方法名和实例变量***.

· 为了能在同步控制部分修改抽象控制变量,增加了转换部分.在转换部分中建议只存取控制变量.

这样,卫式-路径表达式的项变为以下形式:

$$\langle\text{bool-expression,m,transition-statement}\rangle.$$

它表示当 bool-expression 计算为 True 时,可以执行方法 m,当 m 执行后,执行由块语句 transition-statement 指定的语句序列.其中,为了表示方便,当 bool-expression 为 True,或者 transition-statement 为 skip 时,在表示上忽略相应的元素.例如,我们可将⟨true,m,statements⟩简写为⟨m,statements⟩,特别地,可将⟨true,m,skip⟩简写为⟨m⟩或者直接写为 m.

3.1.2 例 子

下面使用扩充的卫式-路径表达式定义有界缓冲区的例子来说明同步控制的独立性.它自包含了一个完整同步控制的所有部分,如接受条件的判断.

```
class Buffer_Control
 Values
      MaxSize=BufferSize
```

* 控制类是指仅包括控制变量、控制方法和同步代码的类,它不能被实例化.

** 完全类是指既包括同步代码,又包括功能性规约的并行具体类.它不包括抽象方法,可以被实例化.

*** 在完全类中,同步控制已丧失了独立性.

control variables

currentsize=0

methods

Put(x:ElementsType) **is not specified yet**

Get() **is not specified yet**

Size() **is not specified yet**

synchronization

(〈(currentsize〈MaxSize),Put,(currentsize=currentsize+1)〉|

〈(currentsize〉0),Get,(currentsize=currentsize–1)〉|Size$^{\#}$)*

end Buffer

在有了上面的同步类以后,功能性开发人员只需关心方法 Put,Get 和 Size 的开发而无须涉及任何与并发有关的规约.

3.2 卫式-路径表达式的分解

上一节主要从独立性的方面介绍了卫式-路径表达式的扩充.本节从分解的角度介绍卫式-路径结构.

对卫式-路径表达式进行分解一方面可增强其易读性,更主要的是可增强其复用性和扩充性[2].幸运的是,卫式-路径表达式本身为其分解提供了自然的依据.

如果用逆波兰方法表示一个卫式-路径表达式,那么依据操作符的评估顺序将形成一个树形结构.这个树形结构的每个非叶节点均对应一个操作符,叶节点对应一个基本项.这为同步代码的分解提供了有效的手段.图 1 显示了分解的方法.

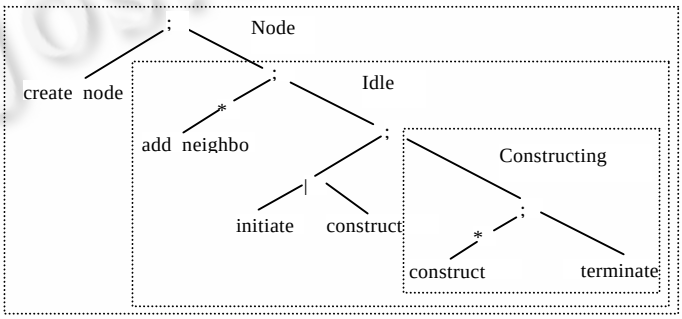


Fig.1 The tree structure of guard-trace expression of node

图 1 Node 卫式-路径表达式的树形结构

3.2.1 卫式-路径结构

通过分解,我们得到了一个具有层次性的卫式-路径结构.它由两部分组成,一部分是由卫式-路径表达式指定的路径集,另一部分是此路径集中涉及到的功能性方法(即非控制方法),记为[trace-set,methodname-set].基于这个卫式-路径结构,我们可将同步控制分为多个具有层次关系的部分,其中每个部分均由一个唯一确定的名来标识,其中关键字 general guard trace,用于说明同步控制的起点;而其他部分使用关键字 subguard-trace 修饰,表示它是某个卫式-路径结构的组成部分.这样,一个类的同步定义通常具有如下的形式:

synchronization

general guard trace *name*=[…]

…

subguard-trace $name_n$=[…]

3.2.2 例子

类 Node 同步机制用卫式-路径表达式规约如下:

create_node;add_neighbour*;(initiate|construct);construct*;terminate.

可以将其表示为如图 1 所示的树形结构,

根据图 1 的分解,可得到如下的类规约.

class Node

…

methods

create_node(idn:nat)…;

add_neighbour(nn:@Node)…; //@Node 为对象指引类型

```
initiate()…;
construct(caller,root:@Node)…;
terminate()…
synchronization
 subguard-trace  Constructing=[(construct*;terminate),{construct,terminate}];
 subguard-trace  Idle=[(add_neighbour*;(initiate | construct);Constructing),
                                {add_neighbour,initiate,construct,terminate}];
 general guard trace Node=[(create_node;Idel),
                          {create_node,add_neighbour,initiate,construct,terminate}]
end Node
```

有了上面的 Node 的定义之后,我们可以定义 Node 的子类,可以复用 Idle 或者 Constructint.下面我们定义 InitiatorNode 类,用于定义 Node 同步规约行为的特例.

```
class InitiatorNode is subclass of Node
      synchronization
            subguard-trace InitiatorIdle=[(add_neignbour*;initiate;Constructing),
                                          {add_neighbour,initiate,construct,terminate}];
            subguard-trace InitiatorNode=[(create_node;InitiatorIdle),
                                          {create_node,add_neighbour,initiate,
                                          construct,terminate}]
end InitiatorNode
```

4 关联机制

依据有效解决同步代码复用的原则[1,2],迄今已解决了其中的两个.首先,由于路径机制具有与方法体部分天然的分离性,卫式路径机制同样也具有,因而满足了同步代码与方法体代码分离的原则.其次,通过定义子卫式路径(sub-guard-trace)对同步控制进行模块化处理,可以复用相同的子卫式路径,满足了同步代码的分解原则.

对于独立性原则,尽管使用上一节提出的抽象控制空间和继承机制能够部分地得到解决,但是在某种意义上来说这种解决是不完备的.这体现在仅使用继承机制无法将一个用于描述同步模式的控制类用于一个已存在的具体类,反之亦然;另外,描述同步模式的控制类中的方法与期望采用此模式的类规约中的方法可能不是一一对应的,因为具体类中的多个方法可能同时采用相同的同步模式,继承机制无法实现这种多对一的映射.因此,为了建立相对完备的独立性,引入了关联机制,用于同时复用同步代码和具体类.

4.1 关联定义语法

在类的定义中,关联机制具有如下的形式:

controlled by controlling-class-name
where
implemented-method-name$_1$=>abstract-method-name$_1$
…
implemented-method-name$_n$=>abstract-method-name$_n$

其中,implemented-method-name 指该类继承或定义的方法,而 abstract-method-name 代表在控制类 controlling-class-name 中被控制的方法.映射满足以下等式:

$$\forall i{:}N_1((1\le i<j\le n)\wedge((\text{implemented-method-name}_i\neq\text{implemented-method-name}_j)\Rightarrow \text{abstract-method-name}_i\neq\text{abstract-method-name}_j))).$$

4.2 例 子

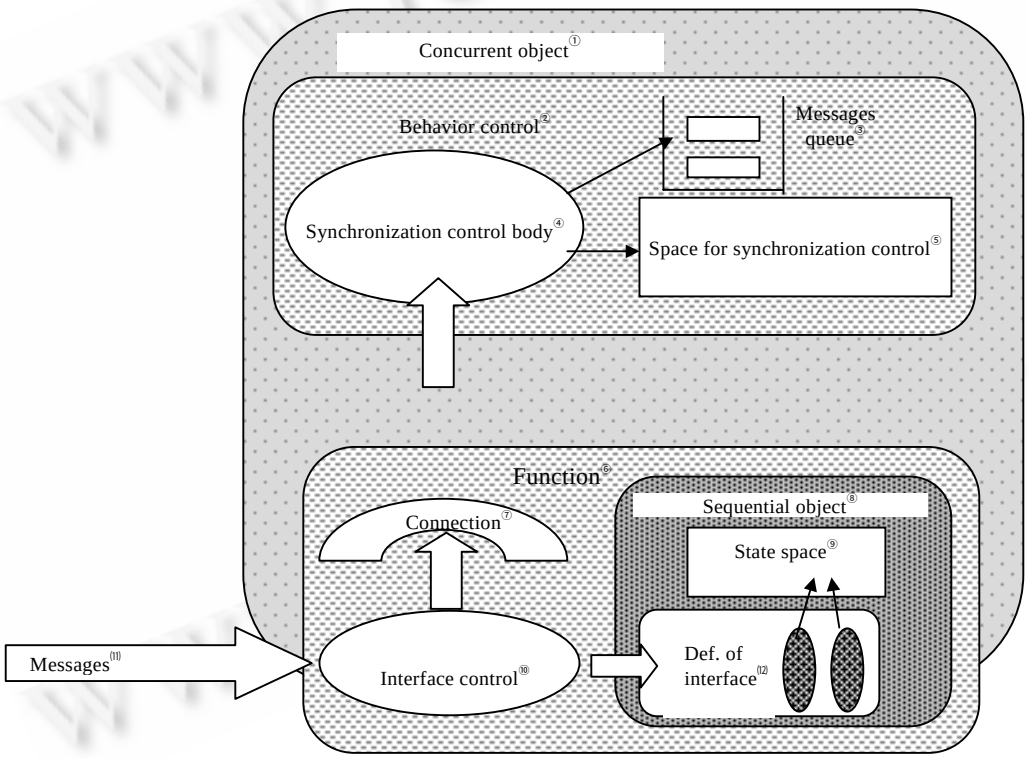
下面给出通过关联机制,将前面的缓冲区同步控制用于不同缓冲区规约的例子.

```
class S_Buffer
    instance variables
        …
    methods
        put(…)…
        putlong(…)…
        get()…
        is_full()…
        size()…
end S_Buffer
```

```
class Buffer is subclass of S_Buffer
    controled by Buffer_Control
    where
        put,putlong=>Put;
        get=>Get;
        size,is_full=>Size
end Buffer
```

5 基于卫式-路径的对象执行模型

并发对象的执行过程如图 2 所示.一个并发对象主要两部分组成,一部分是行为控制部分,另一部分是功能实施部分.其中的同步控制由接口控制、名映射和同步控制体协作完成.下面分别从接口控制、名映射和同步控制体这 3 个部分来阐述一个完整的对象响应外界消息的过程.



①并发对象,②行为控制部分,③消息队列,④同步控制体,⑤同步控制空间,⑥功能体,
⑦名映射,⑧顺序对象,⑨顺序对象状态空间,⑩接口控制部分,⑾消息,⑿接口定义部分.

Fig.2 Execution model of concurrent object based on guard trace structure

图 2 基于卫式-路径结构的并发对象执行模型

(1) 接口控制.接口控制部分在编译时刻自动产生,对规约的设计者是透明的.当一个并发对象接收到一个执行方法 m 消息以后,首先由接口控制部分向名映射部分发送“开始执行方法 m”的消息,当且仅当得到回答以后,再调用顺序对象中相应的方法 m,当此方法执行完以后,向名映射部分发送“方法 m 已执行完”的消息,当接到相应的回答后,向调用者返回相应的消息,从而完成一次消息的响应过程.在接口控制部分,每个方法均存在上

述过程,它们可以无限制地被并发调用.

(2) 名映射.名映射对应于类规约中的关联部分,它主要完成名映射的功能.当接收到接口控制发来的消息以后,根据名对应表,将具体方法名(如 m)转换为同步控制中使用的抽象方法名(如 a_m),然后向同步控制体发送“开始抽象方法 a_m 的执行”或“结束抽象方法 a_m 的执行”,当且仅当接收到相应的回答后,向接口控制部分发送回复消息.

(3) 同步控制体.同步控制体对应于类规约的同步部分,在并发对象内部实现为一个线程,是在并发对象创建时自动产生的.当同步控制体接收到新的消息以后,首先查看消息队列中是否存在消息,然后依据同步表达式当前的位置及当前的抽象状态来决定哪些消息可以立即执行.若存在多个可以立即执行的消息,选择其中的一个接受,并释放相应调用者,其他消息仍放回消息队列,等待下次的选择.对于方法执行结束的消息立即执行,以做相应的状态转换工作.

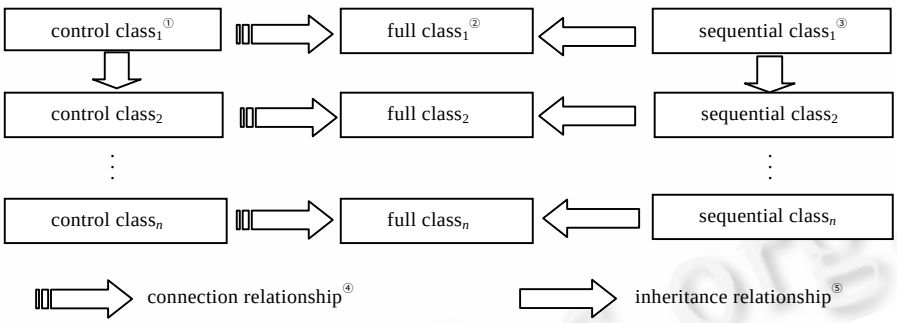
6 同步与关联机制的继承

从行为继承的角度出发,我们规定了如下的同步继承策略:在默认的情况下,子类继承多个父类的同步机制,并使用并行操作符将其联接成一个同步规约.但在下面的几种情况下,需要特殊考虑:

· 当在子类中的同步规约中存在 general guard trace 关键字时,只继承各父类的同步规约,但不对其进行并发操作.子类基于的同步规约由 general guard trace 后面的 guard trace 结构指定.

· 当子类存在与父类同名的 subguard-trace 时,子类中的子卫式路径对父类中的进行重定义,并使用这个重定义的卫式-路径子结构作为子类卫式-路径结构的组成部分.

在当前的继承模型中,不支持关联关系的继承.这是因为继承关联关系可能打破了控制类的封装性,并且不易于实现和维护.因此我们建议采用如图 3 所示的开发模式.



①控制类,②完全类,③顺序类,④关联关系,⑤继承关系.

Fig.3 Inheritance model for synchronization and connection

图 3 同步与关联的继承模式

这种模式的优点在于可以将系统的功能规约与同步规约有效地分离开来,从而可以将开发人员分为 3 组,一组专门研究能够使系统高效的同步模式,一组进行系统功能的开发,另一组通过集成,对另外的两组进行同步.当然,在实际开发中不可能完全严格按照上面的模式.例如,对于某个顺序子类,其同步规约模式可能完成脱离或者很难继承父类的同步模式,这时,为此顺序类设计一个新的同步模式是比较合理的选择.

7 结论与今后的工作

本文基于 VDM++中的 Trace 机制提出了 Guard-Trace 同步机制.一方面它通过并行符和执行条件的引入增强了 Trace 的表达能力,更重要的是,通过卫式路径天然的分离性、卫式路径结构的可分解性以及由于关联机制的引入带来的独立性,对同步代码的复用进行了全方位的支持.另外,表达式基本项的可控制性、分解的灵活性和关联机制的默认规则使得用户在效率和复用性等方面进行取舍成为可能.

我们将在另一篇文章中介绍基于卫式-路径同步机制的实现技术.通过模拟实现,我们发现了并行路径的选择等不确定问题.这些问题表明,关于 Guard-Trace 同步机制,今后的工作应该从以下几方面入手:

· 研究卫式-路径中的不确定性及其如何对其进行解决;

· 研究卫式-路径的形式语义,从而发现并行运算符"||"和"#"产生不确定性问题的一般规律,以及在此语义模型的基础上研究并行面向对象程序的死锁、饿死等安全性问题.

References:

[1] Yang, Da-jun, Zhang, Ming, Lü, Jian. The study on concurrent object oriented programming language. Computer Research and Development, 1998,32(9):769~775 (in Chinese).

[2] Zhang, Ming, Lü, Jian, Yang, Da-jun, *et al*. A two-layered-class method for solving the inheritance anomaly. Journal of Software, 1999,10(6):619~625 (in Chinese).

[3] Durr, E.H., Plat, N. VDM++ Language Reference Manual. Afrodite (esprit-iii project number 6500) document afro/cg/ed/lrm/v9 edition, Utrecht, Netherlands: Cap Volmac, 1994.

[4] Kafura, D.G., Lee, Keung Hae. Inheritance in actor based concurrent object-oriented languages. In: Cook, S., ed., Proceedings of the ECOOP'89. Cambridge, MA: Cambridge University Press, 1989. 131~145.

[5] Tomlinson, C., Singh, V. Inheritance and Synchronization with Enabled-Sets. ACM SIGPLAN Notices, 1989,24(10):103~112.

[6] Neusius, C. Synchronizing actions. Lecture Notes in Computer Science, 1991,512(7):118~132.

[7] Frølund, S. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In: Lehrmann, M.O., ed. Proceedings of the ECOOP'92. LNCS 615, Springer-Verlag, 1992. 185~196.

[8] Matsuoka, S., Yonezawa, A. Analysis of inheritance anomaly in object-oriented concurrent languages. In: Agha, Gul, Wegner, P., Yonezawa, Akinori, ed. Research Directions in Concurrent Object-Oriented Porgramming. Cambridge, MA: MIT Press, 1993. 107~150.

[9] America, Pierre. POOL-T: a parallel object-oriented language. In: Yonezawa, A., Tokoro, M., ed. Object-Oriented Concurrent Programming. Cambridge, MA: MIT Press, 1987. 199~220.

[10] Jan, L.A., van de Snepscheut. Trace theory and VSLI design [Ph.D. Thesis]. Eindhoven, Netherlands: Technische Hogeschool Eindhoven, 1983.

附中文参考文献:

[1] 杨大军,张鸣,吕建.并发面向对象程序设计语言研究与进展.计算机研究与发展,1998,35(9):769~775.

[2] 张鸣,吕建,杨大军,等.基于双层类结构的继承异常处理方法.软件学报,1999,10(6):619~625.

Research on a Concurrent Object-Oriented Synchronization Model*

LÜ Jian, YANG Da-jun, LIAO Yu, TANG Bao

(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093, *China*);

(*Institute of Computer Software, Nanjing University, Nanjing* 210093, *China*)

E-mail: lj@nju.edu.cn

http://moon.nju.edu.cn

Abstract: Synchronization between processes is one of the main features of concurrent programming. However, under the framework of concurrent object-oriented, the existence of synchronization constraints can cause undesired re-definitions of inherited code. Based on the two kinds of synchronization mechanisms in VDM++, a synchronization model, guard trace structure is presented in this paper to be applied to a wide-spectrum concurrent object-oriented specification language. This model can support not only reuse of general code, but also that of synchronization code effectively.

Key words: concurrent object orientation; inheritance; synchronization; inheritance anomaly

* Received March 17, 2000; accepted July 31, 2000

Supported by the National Natural Science Foundation of China under Grant No.69873021; the National High Technology Development 863 Program of China under Grant No.863-02-ZT02-01-4; the National Foundation of China for Palmary Youth under Grant No.61525204; the Natural Science Foundation of Jiangsu Province of China under Grant No.13J99016