用于描述面向对象并发系统的实用图文法模型*

徐建礼 周龙骧

(中国科学院数学研究所,北京 100080)

摘要 面向对象的并发系统与传统的并发系统(如用CSP或CCS所描述的系统)的不同之处在于其进程结构的动态性,系统中的进程以及进程之间的通信链路随着对象的变化而动态地建立或撤消.图文法模型比其他形式化工具更适合描述这种并发和动态的特性.这里我们介绍一个新的用于描述面向对象并发系统的图文法模型,在该模型中为系统的设计开发者提供了一个用来描述系统的静态和动态结构的语言工具——CSDL语言.在面向对象并发系统开发支持环境的支持下,用CSDL语言描述的面向对象并发系统的结构可以转换成对系统运行期进程互联结构的控制机制,用CSDL语言描述的系统中各组成部分之间的依赖关系和通信关系将被动态地映射(binding)到进程间具体的消息链路上.

关键词 并发系统,图文法,形式化方法,多处理机,面向对象.

近年来,并发性与并发程序设计语言已成为计算机科学中最热门的研究领域之一,对各种并发计算模型的研究也吸引了很多人的兴趣.计算机研究与应用领域的下述发展趋势更加突出了并发性在其中的作用[1].首先,单个用户使用交互式多进程系统的情况大大增加,如:工作站甚至PC机上各种多窗口系统的使用等;第二,高速宽带局部网络(LAN)技术已经有了很大发展,用这样的LAN互联起来的工作站和服务器系统已经成为资源共享和分布式问题求解的有效体系结构.例如:工作站网络系统上的分布式数据库管理系统;第三,并行计算机体系结构和多处理器技术的高速发展,使得花费常规的代价就能获得传统的超大型机的计算能力.例如:MPP、SMP以及多计算机系统(multicomputers),多计算机系统目前称为计算机簇(workstation cluster)或机群系统[1].与此同时,在软件工程方面,面向对象的程序设计方法得到了广泛采用,使得数据的抽象性和程序的模块性大大提高.对象通常被定义成将数据以及在数据上的操作封装在一个计算单元中的较为独立的实体.对象之间的相互作用只靠互相通信来完成,因此,我们说并发性是面向对象系统的固有特性.例如第一个面向对象语言Simula在传统的结构上采用并发过程(coroutines)来描述并发性.到目前

* 本文1994-09-30收到,1994-12-29定稿

本研究得到国家自然科学基金、863计划、中国科学院院长基金和中国科学院管理决策与信息系统实验室的资助.

作者**徐建礼**,1962年生,助研,主要研究领域为分布式数据库管理系统,并行系统,多媒体数据库.**周龙骧**,1938年生,研究员,博士导师,主要研究领域为分布式多媒体数据库系统.

本文通讯联系人:徐建礼,北京100080,中国科学院数学研究所

为止,在并发的面向对象程序设计语言(COOP)方面,人们已经做了大量的工作[2],其发展将为多处理体系结构上的并发计算打下坚实的软件基础.

在基于COOP的并发软件系统的开发过程中,仍有两个尚未完全解决的问题.其中一个问题是,面向对象的并发系统与传统的并发系统(用类似CSP这类语言所描述的系统)有以下不同:面向对象并发系统的进程互联结构(process topologies)是动态的,在运行过程中,系统中的进程和进程间的通信关系随着系统中对象的变化而动态地创建或撤消.而在传统的并发系统中,系统的进程互联结构是静态的,是在系统运行前安排好且不随着系统的运行而变化的,如采用CSP或时序逻辑所刻划的并发系统.目前对这种动态的并发性和并发行为还没有很好的形式化方法与工具来描述,也没有基于某种形式化方法的有效的进程互联结构的控制机制.现有的一些支持动态互联结构的并发系统环境,如:Actor systems,Nil和Ada等,对进程互联结构的控制所提供的支持都很差[3].程序员需要在程序中具体地描述和安排系统的进程互联结构及其所有可能的变化,这其中包括:每个进程的创建和撤消、每一条通信链路的建立和撤消等.这样完成的并发系统很难保证其正确性,而且,由于并发系统中各种不确定因素的影响,很难对其进行调试(debug).第二个问题是,在运行在多处理环境下的面向对象并发系统中,进程间的通信成为系统的主要难点.由于面向对象并发系统(以下简称为并发系统)的进程互联结构的动态特性,使得对进程通信的维护和控制变得非常困难,而且,现有的进程通信所提供的服务和支持相对于并发系统的要求来说层次太低,并发系统的开发者必须花费大量的精力来设计他们自己的进程通信协议,以及基于这些协议的复杂的并发控制和同步协议.

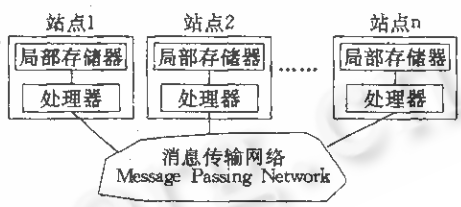


图1 多机系统结构示意图

在本文中,我们给出一个用于描述并发系统的实用的图文法模型,其目的是要解决上述两个难题.该模型及其支持环境能在动态进程互联结构的条件下提供对进程通信的有效支持.以下我们假设并发系统中的进程可以分布到多处理体系结构中的各个处理器上,而且进程间的通信是靠消息传递(message passing)方式进行的.图1是一个多处理体系结构示图.以图文法模型为基础,我们设计了并发系统描述语言CSDL(Concurrent System Description Language),并将其实现在原形系统GRADECS(GRaph GRammar bAsed Development Environment of Concurrent Systems)中[4].CSDL用于形式化地说明并发系统的进程互联结构及其合法的演化规则.根据一个并发系统的CSDL说明,GRADECS可以自动为该系统构造进程控制机制和进程通信机制,并且提供运行期支持.

本文第1章介绍基本图文法模型,第2章将基本图文法模型扩充为一个实用的并发系统描述模型,第3章介绍CSDL语言,最后简要介绍原形系统GRADECS.

1　并发系统的基本图文法模型

1.1　为什么要使用图文法

对图文法的研究始于70年代初,初期的研究重点多集中在图文法自身的理论方

面[5-7]. 从 80 年代末以来,一些研究将重点转向了如何将图文法理论应用到并发程序设计上[3,8,9]. 描述并发系统的难点在于必须清晰地描述出多个进程的控制流与数据流之间的复杂的二维关系[9]. 这个难题在图形化方法的帮助下可以得到解决,在文献[8,9]中比较了一些典型的文字(textual)的和图形(graphical)的形式化工具,例如:Actors, Ada, CCS, Cantor, Petri nets, Graph grammars 等,指出图文法非常适于描述具有动态结构的并发系统. 并发系统的结构和各个组成部分之间的通信关系可以用图的形式来表示,系统运行过程中结构和通信关系的演变可以用图文法的重写规则来描述. 基于这一思想,在文献[3,8]中提出了用于并发程序设计的图文法模型 GRAP.

1.2 基本图文法模型

一个图文法是一个重写(rewriting)系统,它通过引用一个有限图文法规则集中的规则,从一个初始图推出一系列新的图. 图文法在结构和行为上都与串文法相似,不同之处是图文法中的重写是通过子图替换而不是子串替换完成的.

1.2.1 图的形式定义

在并发系统的图文法模型中,图的形式要反映出并发系统的特点. 在我们的图文法模型中,图中包含三种类型的元素:节点(node)、端口(port)和节点与端口之间的有向边(arc),它们分别代表了并发系统中的组成元素(程序模块或进程)、元素上的消息端口,以及元素之间的联系.

定义 1. 一个图 G 定义为五元组 $G=(\Sigma,N,P,f,E)$,其中

$\Sigma=(\Sigma_N,\Sigma_P)$是一对符号集,$\Sigma_N$ 称为节点名集,Σ_P 称为端口名集;

N 是节点集,其中每个节点由 Σ_N 中的符号进行标记;

P 是端口集,其中每个端口由 Σ_P 中的符号进行标记;

$f:N\to p(P)$是节点集到端口集的映射,它为每个节点分配端口,f 要满足以下条件:

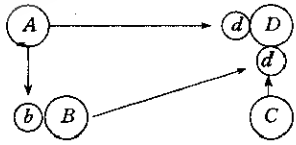
$\forall\ p\in P,\exists\ s\in N$,使得$\exists\ p\in f(s)$.

$P=f(s)$称为节点 s 的端口集;

E 是有向边集,$E\subseteq N\times P$.

注:$p(P)$是 P 的幂集.

图 2 是一个图的例子. 在图的图形表示中,节点由一个中间标有节点名的大圆表示,端口由附在节点周围的小圆表示,每个小圆中标有所代表的端口的端口名,还有一些从节点指向端口的有向边.



$G=(\Sigma,N,P,f,E)$,其中 $\Sigma=(\{A,B,C,D,\ldots\},\{b,d,d',\ldots\})$, $N=\{A,B,C,D\}$, $P=\{b,d,d'\}$, f: $f(A)=\varnothing$, $f(B)=\{b\}$, $f(C)=\varnothing$, $f(D)=\{d,d'\}$, $E=\{(A,b),(A,d),(B,d'),(C,d')\}$

图 2 一个图的例子

这种图形表示法能清楚直观地表示出并发系统的静态结构和动态进程互联结构. 在表示并发系统的静态结构时,图中每个节点代表一个可执行的程序模块,节点上的端口代表程

序模块中的通信点或同步点,节点与端口之间的有向边代表各程序模块之间的各种可能的通信与同步关系.这一静态结构图称为并发系统的原理图(axiom graph).在并发系统运行时,每个进程(某一程序模块的一次执行)被表示成一个节点,每个进程节点有若干个通信端口,从进程节点到通信端口的有向边代表进程间的实际通信链路.运行期某一时刻并发系统的进程互联结构图可通过使用图文法产生式,从原理图出发,经过一系列重写而得到.

1.2.2　重写形式定义

图的重写是通过引用图产生式将一幅图变换成另一幅图.为了避免子图的识别与替换等一类复杂问题(其计算复杂度都是NP完全的),提供实用的图文法形式工具,在这里我们采用节点标识控制的图文法(Node－Label Controlled)[10],简记为NLC图文法.在NLC图文法中,每次重写是用一个子图来置换一个节点,被置换节点的节点标识决定了使用哪一条产生式.我们先给出图文法产生式的定义,然后给出重写过程和子图嵌入的形式定义.

定义 2. 一个产生式具有如下形式:$p:L\rightarrow B,E_m$,其中:

p 为该产生式的唯一标识;

$L\in N$ 是产生式的左部或称为目标(goal);

B 是用来替换 L 的一个图,称为body graph;

E_m 是嵌入说明,它说明 B 替换 L 后如何嵌入到 L 原来所在的图中,E_m 由标有嵌入表达式(Edge－End Embeding Expression)的槽(Sockets),以及从这些sockets到 B 中端口或从 B 中节点到这些sockets的有向边组成;

B 和 E_m 一起称为产生式的右部.

对于图文法产生式我们也采用直观的图形表示法,即,将产生式中的 B 和 E_m 表示成一个特殊的图.在上述定义中,E_m 中的每一个socket(在图形表示中为一个方框)中所标记的嵌入条件表达式(简记为 E^4)表示的是 B 中的节点或端口可与原图中的哪些端口或节点相连.在并发系统中,嵌入说明用以描述新产生的进程与系统中其他进程的通信和同步关系.

E^4 的定义域和值域是产生式目标 L 的相邻元素(neighbourhood),其中包括有边与 L 的端口集中元素相连的节点和从 L 出发的有向边所指向的所有端口.这样的嵌入说明称为邻域控制的嵌入(neighbourhood controlled embeding),简记为NCE.两个节点 A 和 B 称为是相邻的,如果从 A 到 B 的端口集中某个端口有边相连,或从 B 到 A 的端口集中某个端口有边相连.NCE类型的嵌入说明能保证:对两个不相邻的节点进行任意次重写后,这两个节点仍不会相邻.使用NCE的图文法的这类性质称为联系不变性,这一重要性质使图文法成为一个较好的并发系统的形式化描述工具.这一特性将在第2章中讨论.下面给出 E^4 的具体定义.

定义 3. E^4 具有以下的语法、类型和语义:

a. E^4 的语法

$\langle E^4\rangle ::= IN \mid OUT \mid \langle port\ name\rangle \mid \langle node\ name\rangle$	基本集
$\mid filter(\langle E^4\rangle,\langle pred\rangle)$	筛选函数
$\mid \langle E^4\rangle/\langle port\ name\rangle$	转换开关
$\mid \langle E^4\rangle \cdot \langle port\ name\rangle$	从属关系

$\langle E^4\rangle \cup \langle E^4\rangle$	并
$\langle E^4\rangle \cap \langle E^4\rangle$	交 } 集合运算
$\langle E^4\rangle - \langle E^4\rangle$	差

$\langle pred\rangle$ 为一阶谓词

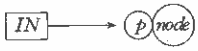
b. E^4 中的运算符类型和操作语义

操作符	类型	操作语义
IN	1→*set*:*node*	所有与 *goal* 所属端口有边相连的节点的集合
OUT	1→*set*:*port*	所有与从 *goal* 出发的边相连的端口的集合
⟨*port name*⟩	1→*set*:*a*	所有与属于 *goal* 且由⟨*port name*⟩指定的端口有边相连的节点的集合，或，与 *goal* 有边相连且由⟨*port name*⟩指定的端口的集合
⟨*node name*⟩	1→*set*:*node*	所有有边指向 *goal* 的某一端口且由⟨*node name*⟩指定的节点的集合
filter	*set*:*a*×*predicate*→*set*:*a*	满足给定谓词的节点或端口的集合
/⟨*port name*⟩	*set*:*port*→*set*:*port*	所有与'/'之前所说明的端口从属于相同节点且由⟨*port name*⟩指定的端口集合
·⟨*port name*⟩	*set*:*port*→*set*:*port*	所有从属于'·'之前所说明的节点且由⟨*port name*⟩指定的端口集合
∪, ∩, −	*set*:*a*×*set*:*a*→*set*:*a*	标准集合运算

注：":*node*"表示节点集，":*port*"表示端口集，":*a*"表示节点集或端口集.

一个 E^4 表达式可由定义 3 中的八类操作构造出来. E^4 是强类型的，一个 E^4 要么表示一个节点集，要么表示一个端口集. 一个 E^4 表达式的类型，可由其所标注的 *socket* 在图文法产生式中的位置来唯一确定：

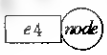
1) *IN*——为一节点集合，它所标记的 *socket* 与产生式右部其他部分的联系方式为：



若 *goal* 出现在产生式右部，则 *IN* 所标记的 *socket* 与 *goal* 节点的联系可简记为：

2) *OUT*——为一端口集合，它所标记的 *socket* 与产生式右部其他部分的联系方式为：

3) 其他形式的 E^4 表达式所标记的 *socket* 与产生式右部其他部分的联系方式，以及所表示的集合类型如下所示.



(a) e4表示节点集合

(b) e4表示节点集合

图文法中最基本的动作是直接的一步推演，称为重写. 在一次重写中，要引用某条图文法产生式来对被重写的图进行操作. 下面给出重写过程的定义：

定义 4. 引用产生式 $p=\langle L, B, E_m\rangle$ 对图 G 中节点 n 的重写(其中 $L=n$)过程定义如下：

a) 对 E_m 中每个 *socket* k，计算其 E^4 表达式，以求出满足 E^4 且与节点 n 的端口有边相连的节点集合，即 n 的相邻节点的子集，或者，属于 n 的相邻节点且有边与节点 n 相连的端口集合，记 *socket* k 的表达式之值为 e_k；

b) 设函数 in：$graph \rightarrow graph$ 是 $G-n$ 和 B 上的同构映射，其中 $G-n$ 是在图 G 中除去节点 n 和 n 的所有端口以及与这两者直接相连的边后得到的图.

c) 设 emb 为所有嵌入边(*embeding edge*)的集合：

• 若 E_m 中有从 B 中节点 t 到 *socket* k 的边，则对 e_k 中的任意端口 s，都有边$\langle in(t), in(s)\rangle$在 emb 中；

• 若 E_m 中有从 *socket* k 到 B 中端口 s 的边，则对 e_k 中的任意节点 t，都有边$\langle in(t), in(s)\rangle$在 emb 中.

d) 令 $H = in(G-n) + in(B) + emb$，$H$ 即为重写后得到的新图.

在图 3 中我们给出一个图文法产生式及引用产生式进行重写的例子. 在这个例子中，产生式 p 的作用是将节点 A 替换成由节点 A 和节点 B 组成的 *bodygraph*，p 一共被引用了两次. 在 p 的嵌入说明部分中一共有两个标有 E^4 表达式的 *sockets*，第 1 个 *socket* 中的 E^4 表达式为"a"，其结果是在原图中与节点 A 的端口 a 有边相连的节点的集合，在这里为$\{D\}$；第 2 个 *socket* 中的 E^4 表达式为"$filter(OUT, x \mid node\_name = C)$"，其结果是一个端口集，它包括所有在原图中属于节点 C 且与节点 A 相连的端口，这里为$\{c\}$.

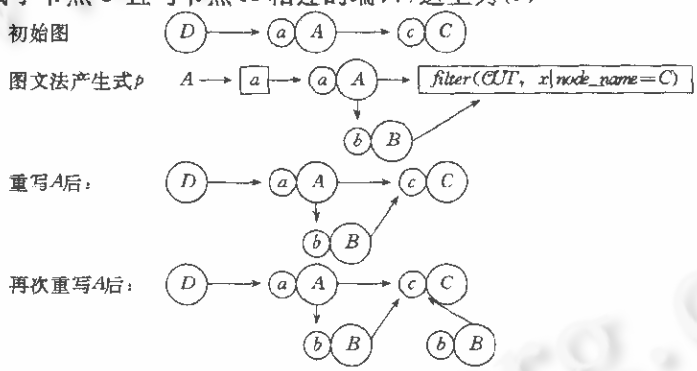


图3 重写的例子

2 扩充的图文法模型

前面所介绍的基本图文法模型有两个明显的缺点：一是产生式是上下文无关的；二是其中没有变量和系统的概念. 文献[3,8]中给出的图文法模型 GARP 便等价于前面的基本图文法模型. 在这一章中我们将对基本图文法模型进行扩充，以克服上述缺点，使之成为一个实用的并发系统的形式化描述工具.

2.1 扩充产生式定义

定义 2 中所构造的重写形式是上下文无关的，但事实上，一条产生式是否能被引用常常是由某些外部条件决定的，如目标节点所处的环境等. 在这里作为对定义 2 的扩充，我们给出一个新的产生式定义.

定义 2'. 一条产生式具有如下形式：$p: L, E_c \rightarrow B, E_m$，其中

p，L，B，E_m 与定义 2 中的相同；

E_c 为产生式左部的条件说明，其形式与 E_m 相同，但其中的边是从 E_c 中的 *socket* 指向目标节点或从目标节点的端口指向 E_c 中的 *socket*. E_c 可以为空.

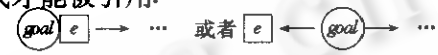
为了描述简单，我们引入一个新的图符[ ]．这是一个*socket*，但其含意为不能有满足其中标记E^4表达式的元素存在．下面给出几个例子来解释这类特殊的*socket*.

例 1： 

只有在不存在与 *goal* 的任一端口相连且满足表达式 *e* 的节点的情况下，这条产生式才能被引用.

例 2： 

只有在 *e* 所表示的端口集为空时，这条产生式才能被引用.

例 3：与前面两个例子类似，我们可以有： 

只有在 *e* 所表示的节点集或端口集为非空时，才能被引用.

2.2 产生式函数

在基本图文法模型中没有变量的概念，因此，在描述实际系统时会产生以下问题：一是，有些产生式在图形表示上是相同的，只是其中的某几个元素上所标记的内容不同，对这些产生式只能逐条进行描述；二是（也是最重要的），由于并发系统的动态特性，产生式中的目标或右部中的某些元素（节点、端口或 socket）上的标记只有在系统运行时的某个时刻才能确定，在基本图文法中对这类问题无法刻划.

为解决上述问题，我们引入了产生式函数的概念．产生式函数可以看成是带有形式参数（变量）的产生式．当一条产生式中的变量被不同的实参替换以后，便会成为不同的产生式．我们将这一替换过程称为产生式函数的实例化．图 4 给出了产生式函数及其实例化的例子.

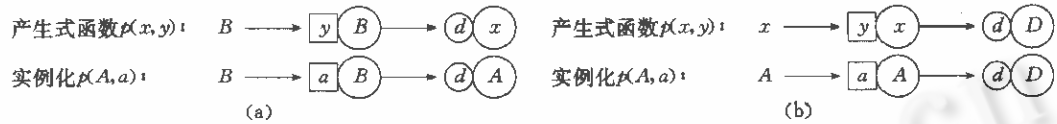


图4 图文法产生式函数及其实例化

2.3 图系统及其联系不变性

在 1.2.2 中，我们曾提到过图文法的联系不变性，下面我们就在图系统的前提下讨论这一性质.

定义 5. 一个图系统是一个四元组 $GS=(\Sigma,A,Prod,G_s)$，其中：

$\Sigma=(\Sigma_N,\Sigma_P)$是一对字符集，$\Sigma_N$ 称为节点名集，Σ_P 称为端口名集，*Prod* 和 G_s 皆由 Σ 中的字符标记；

A 为初始图，称为 axiom graph；

Prod 是图文法产生式集合；

G_s 是图的集合，其中的元素为所有通过引用 *Prod* 中的产生式或产生式序列，从 *A* 出发所导出的图.

在一个图系统中，联系不变性（connection constancy）可以叙述如下：

性质 1. 设图系统 GS 中的 $A=(\Sigma_a,N_a,P_a,f_a,E_a)$，则：

a) 对于$\forall\ G=(\Sigma_g,N_g,P_g,f_g,E_g),G\in G_s$，有 $A\subseteq G$ 成立，并且

b) 对于$\forall\ e\in N_a\times P_a$ 且 $e\notin E_a$，有 $e\notin E_g$ 成立.

所谓联系不变性，是指 axiom graph 中所说明的联系（axiom 图中节点与端口之间的边）在经过重写之后依然保持，并且，在 axiom 图的元素之间不能增加新的联系．由于产生

式嵌入说明中 E^4 表达式的定义域为其目标的邻域,因此性质 1 中的 b)自然得到满足. 为保证 a)得到满足,需要对产生式集合 $Prod$ 中的元素增加以下的限制条件:

条件 1. 在图系统 $GS=(\Sigma, A, Prod, G_r)$ 中,对于任意一条产生式 $p: L, E_c \to B, E_m$, $p \in Prod$,如果 n 为 A 中的节点,则 p 要满足:

a) 如果 $L=n$,则 n 必须也是 B 中节点;

b) 设 E_m 中所有嵌入边的集合为 emb,设 E_n 为 n 在 A 中的所有邻接边的集合,则 $E_n \subseteq emb$ 成立.

其中 n 的邻接边是指从 n 出发的边或指向 n 的端口的边.

联系不变性是扩充的图文法模型中非常重要的性质. 根据这一性质,我们可以用 axiom 图来描述并发系统中各部分之间关键的通信和依赖关系. 如果所使用的产生式满足条件 1,则在系统运行期,无论引用产生式对系统的互联结构带来怎样的变化,在 axiom 图中所说明的关系都不会改变.

3　基于新模型的并发系统描述语言

为了使用图文法模型描述实际的并发系统,我们设计了称为 CSDL(Concurrent System Description Language)的并发系统描述语言. 通过使用 CSDL,我们可以建立并发系统结构的图文法模型描述. 我们先介绍 CSDL 中的一些概念,然后简单介绍 CSDL 语言.

3.1　并发系统的结构

并发系统是要运行在多处理硬件环境上的(多处理器机或各种群机系统),在后面的部分中,我们将多处理硬件环境中的一个处理单元称为一个站点(site).

假设并发系统都是按模块化的方法设计和实现的. 我们知道每个并发系统都有其初始互联结构(静态结构)和动态互联结构(动态结构). 并发系统的静态结构包括一组程序模块和这些模块之间的相互依赖关系. 在面向对象的系统中,这些依赖关系表现为通信关系. 在并发系统的运行期,并发系统可看成是一个动态的进程集合,每个进程是某个程序模块的一次执行,每个程序模块可以有多个进程. 由于每个程序模块都有其特定的功能,因此,我们将同一模块的所有进程所构成的集合称为一个功能簇(function cluster),简记为 f—cluster. 一个运行期的并发系统可看成是若干 f—cluster 所构成的集合. 并发系统的动态互联结构包括其进程结构和进程间的通信关系. 在运行期某一时刻,系统的互联结构可以通过对初始互联结构引用某个产生式或产生式序列而得到. 在描述一个并发系统时,我们既要描述其静态结构也要描述其动态结构.

定义 6. 一个并发系统的静态结构可用四元组 $S=(M, P, f, g)$ 来表示,其中:

M 是可执行的程序模块的集合;

P 是所有形式端口的集合;

$f: M \to p(P)$,映射 f 为每个模块分配形式端口,称为分配映射;

$g: M \to p(P)$,映射 g 给出每个模块与其他模块的形式端口之间的联系,称为互联映射. 若模块 $m1$ 和模块 $m2$ 之间有互联关系,则 $g(m1) \cap f(m2) \neq \varnothing$.

注:$p(P)$ 为 P 的幂集.

可以看出定义 6 与图的定义(定义 1)是等价的,因此可以用图来描述并发系统的静态

结构. 描述并发系统静态结构的图称为并发系统的 axiom 图. axiom 图中的节点和端口分别由程序模块名和形式端口名来标记.

axiom 图中的端口并不是真正的通信端口,而是用来说明各程序模块之间的各种互联关系. 这些互联关系在并发系统运行起来之后,便会演化为各程序模块所对应的进程上的通信端口和进程间的实际通信链路. 因此,将 axiom 图中的端口称为形式端口.

在并发系统的运行期,系统中每个程序模块所对应的活动实体是一个功能簇(f—cluster),因此,对并发系统动态互联结构的说明只需要包括对各个 f—cluster 的描述,以及对同一 f—cluster 中的进程或不同 f—cluster 中的进程之间通信联系的描述. 下面我们给出 f—cluster 的定义.

定义 7. 一个功能簇可定义为六元组 $f-cluster = (m, P, Port, Prod, T, h)$,其中:

m 为功能簇所对应的程序模块(名);

P 是程序模块 m 的所有进程的集合;

$Port$ 是功能簇中所有端口的集合;

$Prod$ 是图文法产生式集合,其中的产生式描述了进程互联结构的合法推演规则;

T 是触发事件的集合;

$h: T \rightarrow p(Prod)$,映射 h 建立了每个触发事件与产生式之间的对应关系,当该事件成立时,其对应的一组产生式将被引用.

注:$p(Prod)$为 $Prod$ 的幂集.

定义 7 中,集合 $Port$ 中包含两部分:一部分是从 axiom 图中对应的程序模块节点上继承下来的形式端口;另一部分为实际的消息通信端口,它们是形式端口的实例化,每个形式端口可有若干个实例端口. T 中的触发事件包括以下几类事件,如:某一特定类型消息的到达、超时、站点故障、消息队列空或溢出,以及其它系统可以感知的事件.

3.2 CSDL

CSDL 是一个基于图文法的、并且能恰当描述并发系统的静态结构和动态结构的工具. 这里我们只简单介绍一下 CSDL 的风格和形式,对 CSDL 的具体定义和应用实例在文献[4]中有详细介绍. 下面给出了并发系统 CSDL 描述的风格和结构,其中的图文法产生式是用图来表示的,在 CSDL 描述中只需给出产生式标识和产生式所在的图文件名即可. 一个并发系统的 CSDL 描述具有如下的结构和形式:

```
SYSTEM system_name                    /* 定义系统名 */
  STATIC                              /* 定义系统的静态结构 */
    MODULE module_name1, SITE site1,
      PARAMETER parameter_file_name;
    MODULE module_name2, ... ;
    ...
    MODULE module_namen, ... ;
      /* 以上是对各系统模块的定义,其中包括模块名、模块所在节点以及启动参数的定义 */
    AXIOM
      〈对系统 axiom 图的描述〉
    AXIOM_END;
  STATIC_END;
```

```
DYNAMIC                    /* 定义系统的动态结构(定义系统中各 f—cluster 的结构) */
  F_CLUSTER f_cluster_name1          /* 定义 f—cluster 名 */
    F_CLUSTER_BEGIN
      MODULE module_name;
      SITE site1, ... , sitej;
          /* 说明该 f—cluster 所对应的程序模块名和该 f—cluster 的进程有可能处于的所有节点 */
      PRODUCTION
        production_name1 IN graph_file_name1,
        production_name2 IN graph_file_name2,
        ...
        production_namek IN graph_file_namek;
          /* 这一部分说明该 f—cluster 所使用的全部图文法产生式(产生式名和产生式所在的图文件名) */
      TRIGGER
        event_name1:INVOKE production_namek,
        event_name2:INVOKE production_name1,
        ....
        event_namep:INVOKE production_nameq;
            /* 此部分说明系统的触发事件,当某事件发生时,系统便引用在此为其指定的图文法产生式来改
变系统的动态互联结构 */
      F_CLUSTER_END;
    F_CLUSTER f_cluster_name2
      F_CLUSTER_BEGIN
      ...
      F_CLUSTER_END;
    ...
    F_CLUSTER f_cluster_namem
      F_CLUSTER_BEGIN
      ...
      F_CLUSTER_END;
                  /* 系统中可有若干个 f—cluster */
  DYNAMIC_END;
SYSTEM_END;
```

CSDL 实现在并发系统开发支撑环境 GRADECS 的顶层,在下一章中我们介绍 GRADECS 的结构和功能.

4　基于图文法的并发系统开发支撑环境 GRADECS

图 5 给出了 GRADECS 的系统结构和系统各部分之间的主要数据流关系. GRADECS 有 4 大模块和两类数据基. CSDL 编译器负责将并发系统的 CSDL 描述转换为系统内部表示形式,当启动并发系统时,通信和进程控制核心(CPC—kernel)会为其生成动态进程互联结构并且提供可靠和高效的进程间消息通信服务. CPC—kernel 是 GRADECS 的核心,其中实现了一个面向 action 的新的进程通信机制[4]. 运行期状态记录器(RS—recorder)负责根据并发系统开发者的要求(用测试要求语言 TDL 来描述)记录并发系统运行期的全部或

某些特定条件下的系统状态. RS－recorder 记录下来的状态测试数据被记入一个分布式数据库中(我们使用的是自行研制的分布式数据库管理系统 C－POREL[11]),或者直接传给运行期状态测试控制器(RST－controller). RST－controller 的功能有:将 TDL 描述的测试要求转换成系统内部表式;在某一时刻或某种条件下从测试状态数据库中收集有关的系统状态数据,并将这些状态数据以图或表格的形式展示给并发系统开发者.

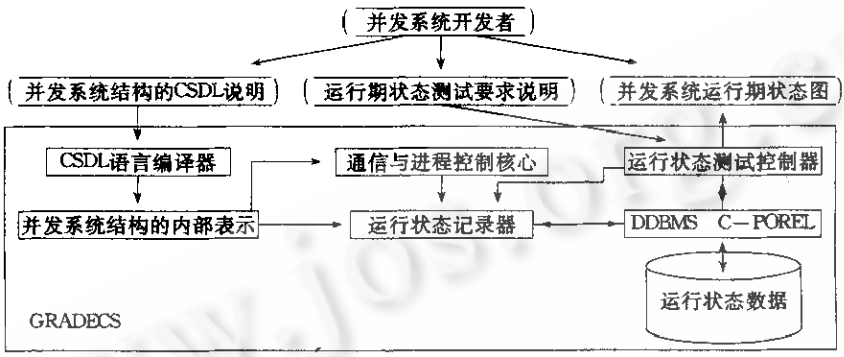


图5 GRADECS系统的结构

GRADECS 的目标机型为多处理体系结构(多处理器机器或机群系统),文献[4]给出了其详细设计与实现方法. 在 GRADECS 的支持下,我们使用 CSDL 语言重新描述并生成了一个现有的大型并发软件系统 C－POREL 的进程互联结构和进程通信功能,将原来 3 个人年的工作量减少到两个人月,其运行效率与原系统大体相同(在 MC68020cpu、UNIX SYSTEM V 操作系统和 10M 以太网环境下). 若各站点机为多处理器机器,可将进程与通信控制核心作为高优先权进程按排在某个处理器上执行. 这样可进一步提高系统运行效率.

5 结束语

在本文中我们介绍了一个新的实用图文法模型,并且给出了使用基于该模型的 CSDL 语言描述并发系统的方法. 这一图文法模型能简洁直观地反映并发系统的本质,其说明性的并发系统描述语言 CSDL 结合了文字描述和图形化说明的特点,使并发系统的规范说明比纯文字或纯图形说明更易读、易于用计算机处理.

在并发系统开发支撑环境 GRADECS 的支持下,图文法的形式化模型工具可以应用于并发系统的结构描述、并发系统运行期的进程互联结构的生成和控制,以及对进程通信的支持. 这样,既保证了并发系统的正确性,又使得并发系统的开发者能够将注意力集中于描述并发系统应有的行为,而不必考虑系统底层的细节,如进程控制、进程间具体通信与同步等.

参考文献

1 Agha G. Concurrent object－oriented programming. CACM, 1990,33(9):125－141.

2 Yonezawa A, Tokoro M. Object－oriented concurrent programming. MIT Press, Cambridge, Mass., 1987.

3 Goering S K, Kaplan S M. Visual concurrent object－based programming in GRAP. LNCS 366, Springer－Verlag, 1989. 165－180.

4 徐建礼. 基于图文法的并发系统开发支撑环境[博士论文]. 中国科学院数学研究所,1991.

5 Ehrig H. Introduction to the algebraic theory of graph grammars. LNCS 73, Springer—Verlag, Heidelberg, 1979. 1—69.

6 Ehrig H, Nagl M, Rozenberg G. Graph grammar and their application to computer science. LNCS 153, Springer—Verlag, Heidelberg, 1982.

7 Ehrig H, Nagl M, Rozenberg G. Graph Grammar and their application to computer science. LNCS 291, Springer—Verlag, Heidelberg, 1987.

8 Kaplan S M, Kaiser G E. GRAP: graph abstractions for concurrent programming. ESOP'88, Spinger—Verlag, March 1988.

9 Kaplan S M, Goering S K, Campbell R H. Specifying concurrent systems with △—grammars. ACM SIGSOFT Engineering Notes, 1989, **14(3)**: 20—27.

10 Janssens D, Rozenberg G. On the structure of node—label controlled graph languages. Information Science, 1980, **20**: 191—216.

11 Zhou Longxiang. C—POREL: a distributed relational database management system on microcomputer network. Scientia Sinica, Series A, 1986, **29(1)**: 78—91.

A PRACTICAL GRAPH GRAMMAR MODEL FOR CONCURRENT OBJECT—ORIENTED SYSTEMS

Xu Jianli　Zhou Longxiang

(*Institute of Mathematics, The Chinese Academy of Sciences, Beijing* 100080)

Abstract　Concurrent object—oriented systems differ from traditional concurrent systems (e.g. systems being described by CSP or CCS) in that the process topologies of them are usually dynamic, and the inter—process communication links among processes are created or withdrawn accordinglywith the change of objects. Graph grammar turns out to be a formal tool much more suitable than other formalisms to specify such kink of concurrency and dynamic characteristics. This paper introduces a new graph grammar model for specifying concurrent object—oriented systems. System developers can specify both the static and dynamic structures of their concurrent object—oriented systems using the CSDL language provided by our model. Via a development and supporting environment for concurrent object—oriented systems (GRADECS), the CSDL specification of concurrent system structure can be transformed to run time process topologies, and the CSDL specification of dependence and communication relations among system components can be dynamically bound to inter—process communication links for message passing.

Key words　Concurrent system, graph grammar, formal method, multiprocessing, object—oriented.