第5卷第12期
1994年12月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 5, No. 12
Dec. 1994

一种面向软件工程的时序逻辑语言*

唐稚松　赵　琛

(中国科学院软件研究所,北京 100080)

摘要　XYZ系统由时序逻辑语言XYZ/E及一组基于该语言的CASE工具集组成.XYZ/E语言的目的是欲使逐步求精,描述及验证、快速原型等一些软件工程方法更加有效.特别地,它还能表示实时通信进程中的动态成分.在统一的框架下,不仅能表示不同层次的抽象描述,而且能表示普通高级语言的各种重要性质.本文是关于这一时序逻辑语言最新、最完整的介绍.

关键词　时序逻辑语言,一阶逻辑,描述,验证,证明规则,Hoare逻辑,状态转换,变换,并发性,通信进程,实时,分布式系统,[illegible],约束定,同步,并行语句,选择语句,CASE工具.

时序逻辑语言XYZ/E是XYZ系统的公共基础.XYZ系统是一个基于该语言的CASE工具集,以提高软件可靠性和软件生产率为目标.CASE工具和语言密切相关,相辅相承.XYZ/E语言是面向软件工程设计的,以适应软件工程的以下领域:逐步求精的设计方法;速成原型方法;抽象描述和验证;图形程序设计,包括分布式和基于共享存储器的并发程序设计;并发进程与数据模块相结合意义下的面向对象程序设计;满足各种工程要求,从非形式化描述到形式化描述的平滑过度;在统一的形式框架下表示不同层次的抽象描述和可高效执行的算法过程;作为一个UNCOL式的通用语言,在编译的编译、源代码到源代码的转换中起中间语言的作用.最后,但不是最不重要的,XYZ/E语言能表示普通高级语言的所有重要性质(包括象指针这样的动态机制).事实上,XYZ/E语言是一个语言系列或者用我们的术语说是适应多种程序设计风范的系列化语言族(Xiliehua Yuyan Zu).在XYZ系统中,XYZ/E作为各个CASE工具之间接口的语义基础,因此,这些工具可以可靠地、自由地结合在一起构成一个更复杂的工具或环境.我们认为,不仅程序语言需要CASE工具的支持,辅助以该语言进行的软件开发和维护,而且CASE工具也需要一个形式化的语言作为这些工具公共的语义基础,满足不同用户、不同要求的工具集成,我们认为这是解决困扰软件工程界的软件工具集成性与灵活性相矛盾问题的可行办法.我们同时认为形式语言XYZ/E非常适合作为大型并行机的核心语言.

* 本文1993-08-12收到

本研究在八五计划期间研究经费主要来自国家自然科学基金的支持.此外,电子工业部与863计划提供了部分经费.作者唐稚松,69岁,中国科学院院士,研究员,主要研究领域为计算机科学与软件工程.赵琛,27岁,助研,主要研究领域为软件工具与环境.

本文通讯联系人,唐稚松,北京 100080,中国科学院软件研究所

1 TLL XYZ/E 的基本部分

1.1 逻辑系统

TLL XYZ/E 以 Manna 和 Pnueli[1,2]提出的线性时序逻辑系统为基础. 它既是一个逻辑系统又是一个程序设计语言. XYZ/E 的一个显著特性就是它能在统一的逻辑框架下以其适当的形式表示普通高级语言的基本性质、不同层次的抽象描述以及其它一些程序设计风范.

变量分为全局变量和时序变量两种,定义变量类型的方法与 Pascal 语言相似. 基本类型包括:整型(INT)、字符型(STRING)、浮点数型(FLOAT)、布尔型(BOOL)、栈型(STACK(X)). 它们的语义用非逻辑公理定义. 这些基本类型是可以扩充的. 一个变量 v 声明为类型 X,可用"$v:X$"表示,等价于($X(v)$). 数组和记录的定义简记为:"$v:ARRAY(n,X)$"和"$v:RECORD(m1:X1,\cdots,mk:Xk)$",分别表示"$v(1):X\wedge\cdots\wedge v(n):X$"和"$v\_m1:X1\wedge\cdots\wedge v\_mk:Xk$". 常量声明方法与 Pascal 语言相同. 标号和名字可以作为常量赋给一个具有 NM 类型的变量.

LOGIC型合式公式(wff)以规范形式定义. 除了经典一阶逻辑的连接词、量词(~(非),∧(与),$\vee(或),→(蕴含),==(等于),$A(全称量词),$E(存在量词),$T(真),$F(假)),还有将来时态的时序算子:[](一直),<>(最终),$O(下一时刻),$U(直到),$W(除非). 只有在特殊的子语言 XYZ/PPE 中,过去时态的时序逻辑算子才能出现在产生式规则的条件部分.

在时序逻辑公式中,有一种称为状态转换等式,形式如下:

$$\$Ov=exp \tag{1}$$

这个等式表明下一时刻变量 v 的值等于当前时刻表达式 exp 的值. 显然,等式(1)与普通高级语言中的赋值语句相类似. 事实上,如果等式(1)的左边是"LB"(一个特殊的系统变量,称为控制变量,用于指明程序当前标号),而且如果右边的表达式被一个符号 y 代替,那么等式(1)变成:

$$\$OLB=y \tag{2}$$

这个等式的意义即是"goto y",这里"y"称为转出标号. 与等式(2)相配,另一个等式形式如下:

$$LB=y \tag{3}$$

表明"y 的定义出现",这里"y"称为定义标号. 等式(2)和等式(3)合称控制(或 LB)等式. 以下形式的等式

$$\$O(v1,\cdots,vk)=(e1',\cdots,ek) \tag{4}$$

是下式的一个简记形式.

$$\$Ov1=e1\wedge\cdots\wedge\$Ovk=ek \tag{5}$$

这里等式(4)和等式(5)表示普通意义下的并行赋值.

1.2 条件元和单元

下列形式的任一个蕴含公式称为条件元(ce).

$$LB=yi\wedge P\Rightarrow\$O(v1,\cdots,vk)=(e1,\cdots,ek)\wedge\$OLB=yj \tag{6}$$

$$LB=yi\wedge P\Rightarrow@(Q\wedge LB=yj) \tag{7}$$

这里“⇒”是一个新的记号，表示蕴含；“P”和“Q”是两个一阶逻辑合式公式，分别称为一个条件元的条件部分和动作部分；(7)式中的“@”表示“O”，“〈〉”. 事实上，(6)式和(7)式表示如果在状态 yi 而且条件 P 为真，那么下一时刻(或者将来某时刻，如果(7)式中的“@”是“〈〉”)执行并行赋值动作或动作 Q，然后转到状态 yj((6)式可以看作是(7)式的特殊情形). 这是两种 XYZ/E 中最基本的条件元. (6)式用于表示可执行算法，(7)式用于表示抽象描述，在(7)式中，P 和 Q 分别用于表示前断言和后断言. 事实上，这两种条件元足于表示各种计算中的顺序程序. 时序逻辑算子“$U”(直到)和“$W”(除非)在 XYZ/E 中应用时，主要以扩展的形式：

$$(M)\ \$U(N \wedge \$OLB=z) \tag{8}$$

这里“U”能够用“W”替代. 在经典时序逻辑中，“直到”和“除非”分别表示成“$(M)\ \$U(N)$”和“$(M)\ \$W(N)$”. 但是在 XYZ/E，我们需要一个更复杂的形式：

$$(M)\ \$U(N) \wedge N \rightarrow \$OLB=z$$

简记为：

$$(M)\ \$U(N \wedge \$OLB=z)$$

实际上，“$(M)\ \$U(N)$”仅是更一般情形的一种特殊情况，即“$(M)\ \$U(N \wedge \$OLB=STOP$”. 在 XYZ/E 中，时序逻辑算子“直到”和“除非”可用于两种不同的方式，用两种不同的形式表示.

(1)表示重复执行“M”，等待“N”的出现，在这种方式下，这两个时序逻辑算子用印刷体表示，象以上介绍的，即“$U”和“$W”.

(2)除(1)外，还可以用这两个时序逻辑算子表示中断，例外情况或实时条件. 在这种方式下，这两个时序逻辑算子用带下划线的特殊记法表示，即“$\underline{\$U}$”和“$\underline{\$W}$”.

一个合式公式如果具有以下形式，我们称之为单元.

$$[][ce1;\cdots;cem] \tag{9}$$

$$WHERE\ cons \tag{10}$$

这里“;”是逻辑与的一个记法，$cel, l=1,\cdots,m$，是象(6)式和(7)式的条件元，$cons$ 是一个由逻辑限制、函数定义或操作定义相逻辑与的合式公式，这里的操作定义的范围仅限于(9)式. 这部分称为单元($unit$)的“$WherePart$”，“$WherePart$”可以为空.

单元里有一些特殊的标号；对于可执行的单元总是从“$START$”标号开始，以“$STOP$”标号结束. 有时“$START$”标号可以跟单元的名字作为后缀. 在每一个单元后都假定有如下形式的 ce，但省略不写.

$$LB=STOP \Rightarrow \$OLB \Rightarrow STOP;$$

还有其他一些特殊的标号，如“$RETURN$”、“$NEXT$”、“$EXIT$”等等. 它们的含义与普通高级语言中的相似. 每一个单元只能有一个“$START$”标号作为定义标号，作为开始执行的起点，我们称它为单元的入口.

例 1：阶乘 $f=m!$ 可以用 XYZ/E 以不同方式表示

(1)可执行程序

$[][LB=START\_fact \wedge m>=0 \Rightarrow \$Oz=1 \wedge \$Oj=1 \wedge \$OLB=l1;$

$LB=l1 \wedge j<m+1 \Rightarrow \$Oz=z*j \wedge \$Oj=j+1 \wedge \$OLB=l1;$

$LB=l1 \wedge j>=m+1 \Rightarrow \$Of=z \wedge \$OLB=STOP]$

(2)抽象描述程序

$[][LB=START\_fact \wedge m>=0 \Rightarrow \langle\rangle(LB=STOP \wedge f=m!)]$

$WHERE\ \$Am(m! \leftarrow \text{IF } m=0 \text{ THEN } 1 \text{ ELSE } m*(m-1)!)$

XYZ/E 语言的一个显著特性就是能够分开也能合起来表示执行算法程序(象(6)式的 *ce*)和抽象描述(象(7)式的 *ce*).

例 2:求阶乘和 $S=SUM(i=0,\cdots k)(i!)$

(1)可执行程序

$[][LB=START\_s \Rightarrow \$Oi=0 \wedge \$Or=0 \wedge \$OLB=l1;$

$LB=l1 \wedge i=k+1 \Rightarrow \$Os=r \wedge \$OLB=STOP;$

$LB=l1 \wedge i/=k+1 \Rightarrow \$OLB=l2;$

$LB=l2 \Rightarrow \$Of=1 \wedge \$Oj=j+1 \wedge \$OLB=l3;$

$LB=l3 \wedge j=i+1 \Rightarrow \$OLB=l4;$

$LB=l3 \wedge j/=i+1 \Rightarrow \$Of=f*j \wedge \$Oj=j+1 \wedge \$OLB=l3;$

$LB=l4 \Rightarrow \$Or=r+f \wedge \$i=i+1 \wedge \$OLB=l1]$

(2)抽象描述程序

$[][LB=START\_s \wedge k>=0 \Rightarrow \langle\rangle(LB=STOP \wedge s=SUM(i=0\cdots k)(i!))$

$WHERE\ \$Ai(0! =1 \wedge (i+1)! =i! *(i+1) \wedge \$An(SUM(i=0\cdots 0)(i!)=0!$

$\wedge (SUM(i=0\cdots n+1)(i!)=SUM(i=0\cdots n)(i!)+(i+1)!)$

(3)混合形式的程序

$[][LB=START\_s \Rightarrow \$Oi=0 \wedge \$Or=0 \wedge \$OLB=l1;$

$LB=l1 \wedge i=k+1 \Rightarrow \$Os=r \wedge \$OLB=STOP;$

$LB=l1 \wedge i/=k+1 \Rightarrow \$OLB=l2;$

$LB=l2 \Rightarrow \langle\rangle(LB=l3 \wedge f=i!);$

$LB=l3 \Rightarrow \$Or=r+1 \wedge \$Oi=i+1 \wedge \$OLB=l1]$

$WHERE\ \$Ai(0! =1 \wedge (i+1)! =i! *(i+1))$

能够在统一的程序框架下表示以上三种形式的程序,使得从最抽象描述到一个高效可执行程序的逐步求精过程变得非常平滑,因为最早的求精步骤能用混合的方式表示不同层次的抽象描述.

显然,用";"连接的 *ce* 不是严格顺序的,必须构成一个有意义的程序单元.为了保证它的有效性,必须有以下假定:

(1)时序变量的框架假定

两边具有同一个时序变量标识的状态转换等式是不必要的.比如:"$\$Ov=v$".在 XYZ/E中,一个不必要的等式总是假定等价"$\$T$".所以,我们总是假定每一个时序变量在每个 *ce* 中都被赋值一次(但是它的值并不需要改变).而这种不必要的等式在程序设计中我们是省略不写的.

(2)标号的规范假定

在一个单元中,定义标号和转出标号必须匹配,除非这些标号是"*START*"、"*STOP*"

等.这个假定避免了一个单元中条件元的分离,同时排出了多余的条件元.

(3)条件的完备性和一致性假定

在一个单元中,具有相同的定义标号的条件元称为相关的.这些相关条件元的条件逻辑或的结果必须为真"$T"(完备性),任何两个相关条件元的条件逻辑与结果必须为"$F"(一致性).前者保证无间隔,后者保证无冲突.(作为不确定性,后面将要讨论更严格的逻辑表示方式)

1.3 结构化的高级形式

(6)式和(7)式的 *ce* 用于表示状态转换命令,这种形式在很多方面有它的优点,但同时在程序证明上也有不足.为了保留优势,同时避免在程序验证上的不足,我们从非结构化的开始,将它作为基本形式(我们称状态转换的子语言为基本 XYZ/E 或 XYZ/BE),然后,以 XYZ/BE 为基础,我们定义高级形式的结构化子语言,称之为结构化的 XYZ/E 或 XYZ/SE.在这个子语言中,有几种形式的语句.

最重要的是循环语句,有如下形式:

$$*[LB=yi \wedge P \Rightarrow (\$OLB=w \mid \$OLB=EXIT); \tag{11}$$

$$LB=w\{\mid R\mid\}\ \$OLB=yi] \tag{12}$$

这里"LB=w{|R|}$OLB=yi"意义是"R"是一组 ce(有一个入口和出口,入口和出口分别由"LB=w"和"$OLB=yi"替换,"LB=w{|R|}$OLB=yi"表示循环体.

XYZ/BE 和 XYZ/SE 的主要差别是 case 语句.形式如下:

$$\begin{aligned}&?\ [LB=y \wedge P1 \Rightarrow \$OLB=z1\\&\quad \mid P2 \Rightarrow \$OLB=z2\\&\quad \cdots\\&\quad \mid \sim \$OLB=zk;\\&\quad LB=z1\{\mid Q1\mid\}\ \$OLB=EXIT;\\&\quad LB=z2\{\mid Q2\mid\}\ \$OLB=EXIT;\\&\quad \cdots\\&\quad LB=zk\{\mid Qk\mid\}\ \$OLB=EXIT]\end{aligned} \tag{13}$$

Sconditional 语句有两种形式:

$$LB=y \wedge P \Rightarrow \$O(Q1 \wedge LB=NEXT \mid Q2 \wedge LB=NEXT) \tag{14}$$

$$LB=y \wedge P \Rightarrow \$O(Q \wedge LB=NEXT) \tag{15}$$

事实上,(15)式是(14)式的简单情形,此时 $Q2$ 等于 $\$T$.在 XYZ/SE 中,我们总采用这种简单方式.XYZ/SE 中还有其他几种语句.比如 Sucession 和 Wait 等(也包括在第 2 节要介绍的并发语句).参见文献[3].从(11)式到(15)式中的语句可以进一步简化成不含 LB 的简法形式,参见文献[3].

除这里介绍的这些语句,用于抽象描述的(7)式,也可以用在 XYZ/SE 程序中.

例 3:用 XYZ/AE,XYZ/BE,XYZ/SE 分别表示 $Z=gcd(a,b)$,a,b 是正整数.

(1)抽象描述程序

$$\begin{aligned}&[][LB=START\_gcd \wedge a>0 \wedge b>0\\&\quad \Rightarrow ()(LB=STOP \wedge z \mid a \wedge z \mid b \wedge \$Ax(x \mid a \wedge x \mid b \rightarrow x \mid z))]\end{aligned}$$

$WHERE\ \$Am,n(m|n==\$Er(n=m*r)$

(2)XYZ/BE 程序

$[][LB=START\_gcd\Rightarrow \$Ox=a\wedge \$Oy=b\wedge \$OLB=l1;$

$LB=l1\wedge x=y\Rightarrow \$OLB=l3;$

$LB=l1\wedge x/=y\Rightarrow \$OLB=l2;$

$LB=l2\wedge x>y\Rightarrow \$Ox=x-y\wedge \$OLB=l1;$

$LB=l2\wedge x<y\Rightarrow \$Oy=y-x\wedge \$OLB=l1;$

$LB=l3\Rightarrow \$Oz=x\wedge \$OLB=STOP]$

(3)XYZ/SE 程序

$[][LB=START\_gcd\Rightarrow \$O(x,y)=(a,b)\wedge \$OLB=NEXT;$

$*[LB=l1\wedge x/=y\Rightarrow(LB=l2|\ \$OLB=EXIT);$

$LB=l2\wedge x>y\Rightarrow \$Ox=x-y\wedge \$OLB=NEXT$

$|\ \$Oy=y-x\wedge \$OLB=NEXT;$

$LB=l3\Rightarrow \$OLB=l1];$

$LB=l4\Rightarrow \$Oz=x\wedge \$OLB=STOP]$

循环语句和分支语句的验证规则如下：

$$\frac{NA|-[][LB=w\{|R|\}\ \$OLB=y]\wedge LB=w\wedge INV\wedge P\Rightarrow @(LB=y\Rightarrow INV)}{NA|-[][LB=y\{|X|\}\ \$OLB=EXIT]\wedge LB=y\wedge INV\Rightarrow @(LB=EXIT\wedge\sim P)} \quad (16)$$

$$NA|-[][LB=z1\{|Q1|\}\ \$OLB=EXIT]\wedge LB=z1\wedge PRE\wedge P1\Rightarrow @(LB=EXIT\wedge POST)$$

$$\cdots\cdots$$

$$\frac{NA|-[][LB=zk\{|Qk|\}\ \$OLB=EXIT]\wedge LB=zk\wedge PRE\wedge Pk\Rightarrow @(LB=EXIT\wedge POST)}{NA|-[][LB=y\{|X|\}\ \$OLB=EXIT]\wedge LB=y\wedge PRE\Rightarrow @(LB=EXIT\wedge POST)} \quad (17)$$

这里“NA”表示非逻辑公理；“PRE”，“$POST$”和“INV”分别表示前后断言和不变式；“X”表示整个程序；“@”表示“O”或“〈〉”[3].

赋值语句和 sucession 语句的验证规则很简单，前者只需要替换，后者需要假言推理.

这些验证规则看起来很象 Hoare 逻辑，但是这些规则是在同一个逻辑语言中，而不象 Hoare 逻辑与逻辑语言和非逻辑语言都相关，所以，这些验证规则的正确性和(相对)完备性能在一阶时序逻辑框架下直接证明. 与 Hoare 逻辑的复杂推理过程相比，优势是显然的. 以下两个定理的详细证明在文献[3]给出.

定理 1. 循环语句、分支语句、赋值语句、scondition 语句、succession 语句的验证规则在一阶时序逻辑中是可推导的.

定理 2. 存在一个基于交互方法的 CASE 工具 XYZ/E_SE，能自动地把 XYZ/BE 程序转换成结构化高级形式程序，XYZ/SE 程序.

XYZ/BE 和 XYZ/SE 都有相应的图形表示，XYZ Chart 和 XYZ PAD. 同时有相应的图形编辑器 XYZ/CFC 和 XYZ/PAD.

1.4 声明和程序

显然一个单元不足于构成一个程序，XYZ/E 程序包括更多的项. 程序大致如下：

$$\%PROG\ ProgramName == MainBlock; PackagesPart. \quad (18)$$

这里 $MainBlock$ 是程序的主要部分，包括所有的声明和算法部分. 实际上，所有这些结

构都涉及并发性. *PackagesPart* 是一系列用于表示重用对象的存贮数据模块，程序开始初期可以为空，然后逐步地扩展和修改. 这两个部分以及模块之间的消息是通过 *Import* 和 *Export* 机制实现的. *MainBlock* 有如下的结构：

MainBlock::=[][*ImportDeclPart*;
ExportDeclPart;
LibraryDeclPart;
TypeDeclPart;
GlobalDeclPart;
ShareVarDecPart;
ProcDeclPart;
ProsDeclPart;
ProdDeclPart;
ProBody]
WherePart　　(19)

除 *Probody* 外，其他部分都可以为空. 这些部分都分别用一对方括号括起来，同时有一个前置记号. 这些记号是 *%IMP*，*%EXP*，*%LIB*，*%TYPE*，*%GLOB*，*%VAR*，*%PROC*，*%PROS*，*%PROD*. *Probody* 的记号分为三类，*%ALG*，*%STM*，*%RLE* 分别代表 XYZ/BE，XYZ/SE 和 XYZ/PE，*WherePart* 以保留字 *WHERE* 打头. 这里许多声明与普通高级语言类似. 他们的形式和含义也与普通高级语言相应的结构相似，但是这些结构在 XYZ/E 中都有一个形式化的解释，作为这个语言的一般用户对此可忽略. 有一些结构在 XYZ/E 中是新的. *GlobalVar* 说明不同于普通高级语言的全局变量. 它表明在整个程序中这些变量的值不能被修改. *LibraryDecl* 用于说明库的名字、这些库包含系统提供的函数调用. *ProcDecl*，*ProsDecl*，*ProdDecl* 分别声明形式过程(顺序的)，(并发)进程和产生式单元. *WherePart* 用于定义在它之前整个作用域范围内的一些递归函数、操作、条件或限制，同样也由方括号括起来. 这个部分也可以用于抽象描述. 有时，为了简洁起见，我们把参数和变量的说明也放入 *WherePart*.

过程和进程的定义，如下的方法可以非常简单：

(1)用 *InputParameterDecl*，*OutputParameterDecl* 和 *InputoutputParameterDecl* 替代 *ImportDecl* 和 *ExportDecl*. 它们相应的记号分别是 *%INP*，*%OUTP* 和 *%IOP*.

(2)只需定义局部变量(记号 *%LOC*)代替 *ShareVarDecl*.

(3)过程(进程)可以只包括它们的子过程(子进程)的定义.

(4)在 *ProsDecl*，形式进程的定义必须包括另外一类形式参数. 即通道参数，(形式)通道连接(形式)进程本身(用 * 表示)到其他(形式)进程(用名字表示). 进程(形式)的结构将在下一部分讨论并发性时解释.

以下是递归过程的定义：

$P(n;par)$==[][*Declarations*;
%ALG[$LB=START\_P\Rightarrow\cdots$;
$LB=l1\wedge n=0\Rightarrow\cdots$;

$LB=l1 \wedge n/=0 \Rightarrow \cdots;$

$\cdots$

$LB=li \Rightarrow P(n-1;par);$

$LB=li+1 \Rightarrow \cdots;$

$\cdots$

$LB=lk \Rightarrow RETURN]$

$WHERE\ B$ (20)

这里,"P"是过程名,"par"代表过程参数,与普通高级语言相类似."$RETURN$"是一个特殊的标号,是"$POP(rst)$"的缩写形式."rst"是一个存储着返回标号的递归栈.过程调用用以下形式表示:

$$LB=lj \Rightarrow P(r;par); \tag{20}$$

这实际上是以下两个 ce 的缩写(即宏定义)

$LB=lj \Rightarrow \$O(finps)=(ainps) \wedge \$O(rst)="lj+1" \wedge \$OLB=START\_P;$

$$LB=lj+1 \Rightarrow \$O(aoutps)=(foutps) \wedge \$OLB=NEXT; \tag{22}$$

这里"$finps$"、"$foutps$"分别表示形式输入参数,形式输出参数,"$ainps$"和"$aoutps$"表示实际输入和实际输出.下面的例子说明用 XYZ/E 写递归过程求阶乘.

例 4:求阶乘的递归过程

$fact(n;\%IOPa:INT)==$

$[][\%VAR[w:INT];$

$\%ALG[$

$LB=START\_fact \Rightarrow \$OLB=l1;$

$LB=l1 \wedge n=0 \Rightarrow \$Oa=1 \wedge \$OLB=l3;$

$LB=l1 \wedge n/=0 \Rightarrow fact(n-1;w|a);$

$LB=l2 \Rightarrow \$Oa=n*w \wedge \$OLB=l3;$

$LB=l3 \Rightarrow \$OLB=RETURN]]$

作为存储模块,它的说明是如下形式:

$PackageDecl::=\%PACK[PackageName;$

$ImportDeclPart$

$ExportDeclPart;$

$TypeDeclPart;$

$ShareVarDeclPart;$

$OperationDeclPart\ ]$

$WherePart$ (23)

涉及操作之间关系的限制可以放到 $WherePart$,所有这些操作的前后断言与这些限制的逻辑与可以作为模块的抽象描述,而不需要用代数公理.

2 XYZ/E 表示并发性

2.1 并发性和不确定性

为了表示并发进程，先作以下三个假设：

(a)每一个进程 *Pri*，有自己的控制变量 *Lbi*

(b)进程的动态实例化用下式表示：

$$\$O\ ProcessInstanceName==ProcessCall \tag{24}$$

这里 *ProcessCall* 表示激活那个进程的一个新的拷贝. 在例 6，我们在进程名的后面加上了下标以区分表示不同的拷贝. (24)式能够扩展包括在 *Parallel* 和 *Select* 语句中(见(29)式和(35)式). 在 *ProcessCall* 里，退出标号“*NEXT*”((22)式)被“*STOP*”或“*EXIT*”替代.

(c)为了表示不确定性和并发性，对 *ce* 有如下扩充形式：

$$LB=yi\wedge P\Rightarrow@((Qj1\&LBj1=zj1)\&\cdots\&(Qjk\&LBjk=zjk)) \tag{25}$$

$$LB=yi\wedge P\Rightarrow@(Qj1\&LBj1)\&\cdots\&@(Qjk\&LBjk=zjk) \tag{26}$$

这里，“@”可以是“$O”、“〈〉”或“[]”，“&”可以是“∧”、“$\vee$”或“$\vee'$”(异或). 当然，以上某些组合是不必要或者说没有意义.

假定 *Prm* 代表进程 *prm* 的实例化的名字，*Prm* 是程序说明处相关进程的一个拷贝，(25)式和(26)式能表示成：

$$LB=yi\wedge P\Rightarrow@(Prj1\&\cdots\&Prjk) \tag{27}$$

$$LB=yi\wedge P\Rightarrow@(Prj1)\&\cdots\&@(Prjk) \tag{28}$$

在这些组合之中，从程序设计的观点来看，只有两种是最有意义的. 我们用两种语句来表示这两种逻辑组合.

第一种是“选择语句”，形式如下：

$$LB=yi\wedge P\Rightarrow!!\ [Con1|>ExeAct1,\cdots,Conk|>ExeActk] \tag{29}$$

语义等价于

$$LB=yi\wedge P\Rightarrow\$O[Con1\wedge ExeAct1\ \$\vee'\cdots\$\vee'Conk\wedge ExeActk] \tag{30}$$

这里“*coni*”和“*ExeActi*”分别表示条件元(*ce*)中的条件部分和动作部分. 我们假定所有 *ExeActi* 中的向前标号是 *EXIT*.

第二种是“并行语句”，能在 XYZ/E 中表示普通程序设计中的并发概念，形式如下：

$$LB=yi\wedge P\Rightarrow\$O(Prj1\wedge\cdots\wedge Prjk) \tag{31}$$

$$WHERE\qquad\qquad \|[prj1,\cdots,prjk] \tag{32}$$

这里“$\|[prj1,\cdots,prjk]$”表示

$$[][prj1\ \$\vee\cdots\$\vee prjk] \tag{33}$$

以操作执行的观点可以用更恰当的形式(34)式代替(33)式

$$\$\vee'\{h=1,\cdots,k\}prjh\ \$\vee'\$\vee'\{t,s=1,\cdots,k\}(prjt\wedge prjs) \tag{34}$$

这里 $\$\vee'\{\cdots\}$ 表示 *prjs* 进程序列异或关系.

但是以上语义必须有以下保证：

$$LB=yi\wedge P\Rightarrow()(LBj1=STOP1\wedge\cdots\wedge LBjk=STOPk)\wedge((LBj1=STOP1\wedge\cdots\wedge LBjk=STOPk)\rightarrow\$OLB=STOP) \tag{35}$$

这里最后一个“*STOP*”可以被“*EXIT*”代替.

式(31)表明所有 $Prji's, i=1,\cdots,k$，必须同时开始，式(35)表明它们必须同时终止或退出.

基于这些假定，在满足一定条件下即可得以下精美形式的并行语句的验证规则：

$$\frac{NA,Prj1\mid-prej1\rightarrow\langle\rangle(postj1\wedge\$OLBj1=EXIT1)\quad NA,Prjk\mid-prejk\rightarrow\langle\rangle(postjk\wedge\$OLBjk=EXITk)}{NA,\|[Prj1,\cdots,Prjk]\mid-(prej1\wedge\cdots\wedge prejk)\rightarrow\langle\rangle(postj1\wedge\cdots\wedge postjk)}\quad(36)$$

(36)式的结论是具有可合成(或称可分解)形式的并行语句抽象描述.但是一般来讲，要做到这点，进程中通信命令的输入变量应满足一定的条件，这问题将在2.2中讨论.此外，还应假定，由于通信可能引起的死锁已经排除，本系统已有一工具对此进行判定.选择语句的验证规则较简单，此处略.

2.2 通 信

从一个进程到另一个进程的消息传递是通过两个进程之间的通道实现，在XYZ/E中，通道是可以动态决定的.与CSP相似，有两条与通道相关的消息通信命令，即输出命令和输入命令，即 $ch!\ y$ 与 $ch?\ x$.为了保证抽象描述及相应验证规则的可分解性，我们要求每一输入变量 x 需满足相应条件 $Prex$.因此，其相应的条件元应具有如下的形式：

$$LBr=yr\Rightarrow ch?\ x\wedge\$OLBr=NEXT;LBr=yr'\wedge Prex\Rightarrow\$OLBr=NEXT\mid\$OLBr=yr\quad(37)$$

$$LBs=ys\Rightarrow ch!\ y\wedge\$OLBs=zs\quad(38)$$

这里，“ch”是从输出进程到输入进程之间的通道的名字，“y”是输出进程的输出表达式，“x”是输入进程的输入变量.上面(37)式中如 $Prex$ 为恒真(即 $\$T$)自然后面一条件元即可省去不写.

以上是执行时的形式，在程序验证时先将“$LBr=yr\Rightarrow ch?\ x\wedge\$OLBr=NEXT$”替换成“$LBr=y\wedge P\Rightarrow\$O(Prex\wedge LBr=NEXT)$”，再进行验证.通信语句的验证规则与赋值语句的相同.

为了解释通信命令的语义，我们需要区分两种类型的通信方式，即同步通信和异步通信.分别用“SYN”和“$ASYN$”表示.作为同步通信，输入命令和输出命令的语义可以用以下非逻辑公理表示：

$$\begin{aligned}&\$A\ ch:CHANNEL\ \$A\ x,y:TYPE\\&((ch!\ y\wedge ch?\ x\rightarrow\$Ox=y\wedge\$O(\sim ch!\ y\wedge\sim ch?\ x))\wedge\\&(ch!\ y\wedge\sim ch?\ x\rightarrow\$Och!\ y)\wedge\\&(ch?\ x\wedge\sim ch!\ y\rightarrow\$Och?\ x))\end{aligned}\quad(39)$$

例5：比较两个数大小的并行语句.

$$\begin{aligned}&LB=max\Rightarrow\$OP1==p1\wedge\$OP2==p2\,\$OLB=l1;\\&LB=l1\Rightarrow\$O(P1\wedge P2)\\&WHERE\ \|[p1,p2];\\&p1=[LB1=START\_p1\Rightarrow\$OLB1=l1;\\&\qquad LB1=l1\Rightarrow c?\ x\wedge\$OLB1=l2;\\&\qquad LB1=l2\Rightarrow\\&\qquad !!\ [m>=x\mid>\$OLB1=l3,m=<x\mid>\$OLB1=l4,\$T\mid>\$OLB1=l2];\\&\qquad LB1=l3\Rightarrow d!\ m\wedge\$OLB1=STOP;\\&\qquad LB1=l4\Rightarrow d!\ x\wedge\$OLB1=STOP]\end{aligned}$$

```
p2=[LB2=START_p2⇒$OLB2=k1;
    LB2=k1⇒c! n∧$OLB2=k2;
    LB2=k2⇒d? y∧$OLB2=STOP]
```

例 6:基于消息通信的生产者—消费者问题

```
%PROG example=[][
%VAR[
    i:INT];
%PROS[
producer(%INP s:INT;%CHN ch(*,con:NM);%IOP sum:INT)==[
%LOC [i:INT
    prodmo:INT];
%ALG[
LB1=START⇒$OLB1=l0;
LB1=l0⇒$Oi=0∧$Oprodmo=l2∧$OLB1=l1;
LB1=l1∧(i<sum)⇒$Oi=i+1∧$Oprodemo=prodmo+i*2∧$OSWRITE(s)
∧$OSWRITELN("Producer produce",i,prodmo)∧$OLB1=l11;
LB1=l1∧(i>=sum)⇒$Oprodmo=-1∧$OLB=l2;
LB1=l11⇒ch! prodmo∧$OLB1=l1;
LB1=l2⇒ch! prodmo∧$OLB1=EXIT ]
];
consumer(%CHN ch1(pro:NM,*),ch2(pro1:NM,*)==[
%LOC[j,f1,s1,s2:INT];
%ALG[
LB2=START⇒$Of1=0∧$Of2=0∧$Os2=0∧$OLB2=l0;
LB2=l0⇒!! [
ch1? j|>$Os1=1∧$OLB2=EXIT,ch2? j|>$Os2=1∧$OLB2=EXIT,
(f1=1)∧(f2=1)|>$OLB2=EXIT];
LB2=l01∧(s1=1)⇒$Os1=0∧$OLB2=l1;
LB2=l01∧(s2=1)⇒$Os2=0$OLB2=l21;
LB2=L01∧(s1=0∧s2=0)⇒$OLB2=l12;
LB2=l1∧(j>0)⇒$OSWRITE(""Consumer receive", j)
∧$OSWRITELN("from 1")∧$OLB2=l0;
LB2=l21∧(j>0)⇒$OSWRITE("Consumer receive"j)
LB2=l21∧(j<=0)⇒$Of2=1∧$OLB2=l0;
LB2=l12⇒$OLB2=EXIT]
]];
[
%ALG[
```

$LB=START\Rightarrow \$Oi=5\wedge \$OLB=l0$;

$LB=l0\Rightarrow \$Oa1==producer1(\%INP\ 1|s;\%CHN\ ch(*,b)|ch(*,con);\%INP\ i|sum)$

$\wedge\ \$Oa2==producer2(\%INP\ 2|s;\%CHNs2(*,b)|ch(*,con);\%INP\ i|sum)$

$\wedge\ \$Ob==comsumer(\%CHN\ ch(a1,*)|ch1(pro,*);\%CHN\ c2(a2,*)|ch2(pro1,*))\wedge \$OLB=l1$;

$LB=l1\Rightarrow \$O(a1\wedge a2\wedge b)$;

$LB=l2\Rightarrow \$OLB=STOP]$

$WHERE\ \|[producer1,procedure2,consumer]]$

].

异步通信输入输出命令的语义可以用以下公理表示：

$\$A\ ch:CHANNEL\ \$E\ q(ch):QUEUE\ \$A\ x,y:TYPE$

$(ch?\ x\wedge\sim full(q)\wedge pre\rightarrow \$Oq=enter(q,x)\wedge \$O(\sim ch?\ x)\wedge$

$ch?\ x\wedge(full(q)\$\vee\sim pre)\rightarrow \$Och?\ x\wedge$

$ch!\ y\wedge\sim empty(q)\rightarrow \$Oy=head(q)\wedge \$Oq=tail(q)\wedge \$O(\sim ch!\ y)\wedge$

$ch!\ y\wedge empty(q)\rightarrow \$Och!\ y)$ (40)

3 XYZ/E 表示基于共享存储的并发性

3.1 同步概念

基于消息传递和基于共享存储器模型的并发程序除在进程定义上后者没有通道参数外，两者还有不同的同步概念，前者主要表现为消息传递之间的等待，后者体现为各进程在一组约束条件下交替地对共享变量实施操作. 程序执行的基本操作是对变量赋值，由于赋值语句在计算机硬件级并没有视为不可分割的操作，各进程对共享变量的并发访问，将导致无法预知的结果. 因此各进程对共享变量的访问应该是互斥的，这是其中一类约束条件的来源. 在更广泛的意义上看待程序的操作，对共享变量的操作并不一定由一个赋值语句就可以完成，有可能是若干个操作(若干赋值语句)的组合，若在这个特定操作完成之前，其他进程并发地访问了共享变量，同样无法得到这个特定操作预期的结果. 所以，并发程序语言要为程序员提供定义一段不可分割执行序列的方法. 这个不可分割的执行序列称为程序的临界区，用临界区概念，可以定义一般意义下互斥同步概念，即进程对临界区的操作，导致相关进程的执行延迟，称为互斥同步.

另外一类约束条件来源于程序设计者的特殊要求. 如果共享变量的状态处在不适于作进程将要实施的操作时，进程将在此等待或称延迟，直至其他进程的执行改变了共享变量的状态使之适于该进程等待操作的状态，该进程又继续运行. 对共享变量状态是否适合将要实施操作的判定完全由程序员根据具体的程序设计要求确定. 例如，生产者—消费者问题，对生产者生产条件而言，程序员可以根据实际要求定义为：若生产缓冲区已满，则生产者等待；也可以定义为：若生产缓冲区已满，且生产者已等待了 T 时间，那么用生产者的新产品代替缓冲区中最旧(在缓冲区中滞留时间最长)的产品. 这类由程序员提供的约束条件，也导致了进程的延迟，称为条件同步.

3.2 基于共享存储的并发 XYZ/E 的同步机制

从较低的层次上看,以上讨论的两种同步概念可以统一为一种.互斥同步可以视为一种特殊的条件同步.同步的机制大致有 P、V 操作、信号量、临界区、管程四种.最为基本的是 P、V 操作.目前 XYZ/E 采用 P、V 操作[5].P、V 操作的实质是特殊的过程调用.

例 7:基于共享存储的生产者—消费者问题

```
%PRO prod_cons==[][
%VAR[
buffer: A(INT; 100);
buff_point:INT;
sign_buff:INT;
sign_produ:INT;
    ]
%PROS[
    produce(%INP sum:INT)==[
%LOC[
i,prodmo:INT ; ]
%ALG[
LB=START⇒$Oprodmo=1∧$Oi=1∧$OLB=l0;
LB=l0⇒P(%IOP sign_buff|x);
LB=l1∧(i<sum)⇒$Oi=i+1∧$Obuff[buff_point]=prodmo+i*2∧$Obuff_
point=buff_point+1∧$OLB=l2);
LB=l1∧(i>=sum)⇒$OLB=l4;
LB=l2⇒V(%IOP sign_buff|x);
LB=l3⇒V(%IOP sign_produ|x);
LB=l4⇒$OLB=EXIT;]
    ];
        consumer()=[
%LOC[
  times, j:INT; ]
%ALG [
LB=START⇒$Otimes=0∧$OLB=l0;
LB=l0⇒P(%IOP sign_buff|x);
LB=l01⇒P(%IOP sign_produ|x);
LB=l1⇒$OSWRITELN("Consumer: buff[buff_point]")
        ∧$Obuff_point=buff_point-1∧$Otimes=times+1∧$OLB=l2;
LB=l2⇒V(%IOP sign_buff|x);
LB=l3∧times<10⇒$OLB=l0;
LB=l3∧times>=10⇒$OLB=EXIT;
```

```
]]
[
i:INT;
%ALG[
LB=START⇒ $Oi=5 ∧ $Obuff_point=0 ∧ $Osign_buff=1
          ∧ $Osign_produ=0 ∧ $OLB=l0;
LB=l0⇒ $Oprod1==produce(%INP i|sum)
     ∧ $Oprod2==produce(%INP i|sum)
     ∧ $Oconsu1==consumer()
LB=l1⇒ $O(prod1 ∧ prod2 ∧ consu1) $OLB=NEXT
     WHERE ‖ [produce[1];produce[2];consumer[1]];
LB=l2⇒ $OLB=STOP;
]]
]
```

4 XYZ/E 表示知识

产生式规则是人工智能用于表示知识的常用方法之一，其本质即是一逻辑蕴涵，它与 XYZ/E 语言的条件元(*ce*)形式相似. 在 XYZ/E 语言族中用于进行基于知识的程序设计的子语言是 XYZ/PE. XYZ/PE 语言的知识表示单元前置记号是%RLE，知识表示单元中所有的条件元具有相同的定义标号和转出标号，因此略去不写；这里的条件元仅表示一些事实，因此右边不再是时序逻辑公式，而仅是普通一阶逻辑公式，这时的条件元已蜕变成一般的产生式规则. 所以 XYZ/PE 主要应用产生式规则分模块表示领域知识，一个模块表示领域知识中的一个概念. 用 XYZ/PE 进行基于知识的程序设计时，在程序算法部分能方便地引用领域知识，使描述性知识和过程性算法在统一的框架下得以结合起来[6].

知识表示单元的结构及产生式规则的语法规则如下：

```
Knowledge_Unit ::=RLE [
                        spec
                        ]
```

这里 *spec* 描述如下：

```
spec  ::=rules query
query ::=[? atom]
rules ::={rule}
rule  ::=lhs → rhs ;
lhs   ::=atoms
rhs   ::=atom
atoms ::=atom {∧ atom}
atom  ::=PRE rgs
```

term ::=[(*VRB*|*expr*)]

expr ::=*FUN rgs*

rgs ::=[(*rgl*)]

rgl ::=*term* {, *term*}

这里{*x*}表示空或者 *x*{*x*}；[*x*]表示空或者 *x*；(*x*|*y*)表示 *x* 或者 *y*；*VRB* 表示变量；*FUN* 为函数标志.

5 结束语

本文主要是文献[7]的翻译和改写，扩充了 3，4 两部分. 文献[7]还深入讨论了 XYZ/E 语言的几个重要性质(动态束定、语言模型等)，同时概要介绍了 XYZ 系统的各个 CASE 工具，限于本文的篇幅和介绍语言为主的目的，本文略去了这两部分的内容.

本语言系统已实现.

致谢 本项研究在八五计划期间研究经费主要得自国家自然科学基金委的支持. 此外，电子工业部与 863 计划提供了部分经费，均此致谢意. 在本文的英文稿(发表于 Chinese Journal of Advanced Software Research, Vol. 1, No. 1, 1994: 1－29)中在说明经费来源时有所遗漏，特此更正，并致歉意.

参考文献

1 Mana Z. Verification of sequential programs. In: Broy M, Schmidt G eds. Temporal Axiomatization, Theoretical Foundations of Programming Methodology, North Holland, 1982.

2 Manna Z, Pnueli A. Temporal logic for concurrent and reactive system. Springer Verlag, 1992.

3 Xie Hongliang, Gong Jie, Tang Zhisong. A structured temporal logic language XYZ/SE. J. of Comp. Sci. & Tech., 1991, **6**(1).

4 Tang Zhisong *et al*. The syntax and explanation of the temporal logic language XYZ/E. Inst. Softw., Acad. Sin., Tech. Rep. No. IS－CAS－XYZ－90－1, 1990.

5 赵琛，张健. XYZ 系统中知识的表示和应用. 技术报告 CAS－IS－XYZ－94－5，1994.

6 赵琛. XYZ/CE 编译器的实现[硕士学位论文]. 中国地质大学研究生院，1992.

7 Tang Zhisong. A temporal logic language oriented toward software engineering－an introduction to XYZ system (I). Chinese Journal of Advanced Software Research, 1994, **1**(1): 1－29.

A TEMPORAL LOGIC LANGUAGE ORIENTED TOWARD SOFTWARE ENGINEERING

Tang Zhisong and Zhao Chen

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

Abstract XYZ system consists of a temporal logic language XYZ/E and a group of

CASE tools based on it. The language XYZ/E is designed to facilitate the software engineering methodologies such as stepwise refinement, specification and verification, rapid—prototyping, and in particular, to be able to represent the dynamic aspects of the real time communicating process. With a uniform framework, it can represent not only the specifications of different abstract levels but also almost every kind of significant features in conventional imperative languages. This paper is the most updated and detailed introduction of this temporal logic language in Chinese.

Key words Temporal logic language, first order logic, specification, verification, proof rules, Hoare—logic, state—transition, transformation, concurrency, communicating process, real time, distributed system, channels, dynamic binding, synchronization, parallel statement, select statement, CASE tools.