国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

红 皮 书

卷Ⅵ.11

功能规格和描述语言（SDL）

建议Z.100—Z.104的附件

第 八 次 全 体 会 议

1984年10月8—19日 马拉加—托雷莫里诺斯

1987年 北京

国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

红 皮 书

卷Ⅵ.11

功能规格和描述语言（SDL）

建议Z.100—Z.104的附件

第八次全体会议

1984年10月8—19日　马拉加-托雷莫里诺斯

1987年　北京

ISBN 92—61—02245—6

CCITT 图书目录

适用于第八次全体会议（1984年）以后

红皮书

卷 Ⅵ.8 - 七号信号系统技术规程。建议Q.721—Q.795(第11研究组)。
卷 Ⅵ.9 - 数字入口信号系统建议Q.920—Q.931(第11研究组)。
卷 Ⅵ.10 - 功能规格和描述语言（SDL）。建议Z.101—Z.104(第11研究组)。
卷 Ⅵ.11 - 功能规格和描述语言（SDL）。建议Z.101—Z.104的附件（第11研究组）。
卷 Ⅵ.12 - CCITT高级语言（CHILL）。建议Z.200(第11研究组)。
卷 Ⅵ.13 - 人机语言（MML）。建议Z.301—Z.341(第11研究组)。
卷Ⅶ - （3 个分册，按册出售）
卷 Ⅶ.1 - 电报传输。R 系列建议（第 9 研究组）。电报业务终端设备。S 系列建议（第 9 研究组）。
卷 Ⅶ.2 - 电报交换。U 系列建议（第 9 研究组）。
卷 Ⅶ.3 - 远程信息处理业务的终端设备和协议。T 系列建议（第 8 研究组）。
卷Ⅷ - （7 个分册，按册出售）
卷 Ⅷ.1 - 电话网上的数据通信。V 系列建议（第17研究组）。
卷 Ⅷ.2 - 数据通信网：业务和设施。建议X.1—X.15（第 7 研究组）。
卷 Ⅷ.3 - 数据通信网：接口。建议X.20—X.32（第 7 研究组）。
卷 Ⅷ.4 - 数据通信网：传输、信号和交换；网路问题；维护和行政安排。建议X.40—X.181(第 7 研究组)。
卷 Ⅷ.5 - 数据通信网：开放系统的相互连接（OSI）；系统描述技术。建议X.200—X.250（第 7 研究组）。
卷 Ⅷ.6 - 数据通信网：网路间的互通；移动数据传输系统。建议X.300—X.353(第 7 研究组)。
卷 Ⅷ.7 - 数据通信网：信息处理系统。建议X.400—X.430(第 6 研究组)。
卷Ⅸ - 干扰的防护，K 系列建议（第 5 研究组）。电缆的建筑、安装和防护以及外线设备的其他组成部分。L 系列建议（第 6 研究组）。
卷Ⅹ - （2 个分册，按册出售）
卷 Ⅹ.1 - 术语和定义。
卷Ⅹ.2 - 红皮书索引。

红 皮 书 卷 Ⅵ.11 目 录

建议Z.100至Z.104的附件

功能规格和描
述语言（SDL）

卷 首 说 明

1 在1985—1988研究期内委托给各研究组的研究课题可查阅该研究组的第一号文献。

2 本卷中的“主管部门”一词是电信主管部门和经认可的私营机构两者的简称。

建议Z.100至Z.104的附件

功能规格和描述语言（SDL）

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

附 件 A

（属于建议Z.100至Z.104）

SDL词汇表

在SDL建议Z.100至Z.104中出现的、用楷体（英文用斜体）字印刷並出现在本词汇表中的术语，都是严格地按本词汇表中定义的意义来使用的。

如果某一用楷体字印刷的词组，例如过程标识符（*procedure identifier*），未出现在此词汇表中，则它可能是两个术语的链接，在此例中就是术语过程后面跟术语标识符。当一个英文单词是斜体字但在词汇表中查不到时，则此单词可能是一词汇表术语的导出词，例如*exported*就是*export* 的过去式。

除一个术语是另一个术语的同义词外，在每个术语的定义后面给出在Z.100系列建议中使用该术语的主要引用处（括在方括号〔 〕之内）。例如〔建议Z.100§3.2〕是指主要引用处在建议Z.100§3.2中。由于常常没有对术语作出定义，这些引用处仅仅是一种参考。

附件C已包含了符号的图形表示和图，包含了正文短语表示法的关键字和这些关键字的用法，因此在本词汇表中不再重复这些内容。

abstract data type　抽象数据类型

抽象数据类型是由该类型的值与对这些值的运算之间的代数关系来定义的一种类型。〔建议Z.100，§2.3，建议Z.104，§1。〕

abstract syntax　抽象语法

SDL的抽象语法在建议Z.101、Z.102、Z.103和Z.104中给出，並概括在附件B中。SDL的抽象语法用来描述SDL定义的概念性结构，以与SDL的每种形式即SDL/GR、SDL/PR和SDL/PE的具体语法相对照。〔建议Z.100，§3.1。〕

access　访问

它是所有的数据类型都隐含有的预定义操作符，用于使一个变量获得与之相关联的值。〔建议Z.104，§4.5。〕

active operator　主动运算符

该运算符要有一个或多个变量作参量，因为它可能改变这些变量所取的值。〔建议Z.104，§4.5。〕

actual parameter　实在参数

实在参数是当创建（或调用）一个进程（或过程）时，传递给该进程或过程的与形式参数相对应的一个值。〔建议Z.101，§2.3，建议Z.103，§2.1。〕

actual parameter list 实在参数表

实在参数表是与某一创建请求或某一过程调用相关联的值列成的表。〔建议Z.101,§2.3,建议Z.103,§2.1。〕

additional-save-set 附加保存组

附加保存组是保存在一个过程中的一组附加信号。〔建议Z.103,§2.1。〕

allocation symbol 分配符号

是在SDL/GR 进程树图中的符号，它附属于某一进程符号。把与该进程符号相关联的进程或子进程分配给某一功能块，该功能块的名字包含在该分配符号中。〔建议Z.102,§2.3。〕

annotation 注解

SDL/GR 中的注解是一种注释，SDL/PR 中的注解是一种注释或注。注解不改变SDL的意义。〔建议Z.101,§3.3,§4.3。〕

arc 弧

弧是进程流图的节点之间的有向连接线。〔建议Z.101,§§2.2。〕

array 数组

是用来引进数组概念的预定义生成元。〔建议Z.104,§5.3。〕

assignment statement 赋值语句

一条赋值语句是一个动作。它把一个值和一个变量关联起来。〔建议Z.101,§2.2,建议Z.104,§4.11。〕

axiom 公理

对表达式中所用变量的所有可能的值都为真的一个布尔表达式。〔建议Z.104,§4.6。〕

B'

是SDL/PR 的关键字，用来引进一个数值的二进制表示。〔建议Z.104,§4.8。〕

basic SDL 基本SDL

由建议Z.101定义的SDL的最小子集。

behaviour　行为

在SDL中，一个系统的行为或功能行为指的是对离散激励的离散响应。〔建议Z.101，§ 1。〕

block　功能块

功能块是功能块实例的同义词。

block definition　功能块定义

功能块定义规定了一个标有名字的功能块类型的结构的性质和连接的性质。〔建议Z.101，§ 2.2。〕

block interaction diagram　功能块交互作用图

SDL/GR 的功能块交互作用图用来表示功能块、信道、信号表、数据表、进程以及进程间信号的流动等的结构。功能块交互作用图也可用来表示把一个系统划分成多个功能块、子功能块、信道和子信道。〔建议Z.101，§ 3.1，建议Z.102，§ 3.2。〕

block substructure 功能块子结构

一个功能块的功能块子结构就是将该功能块划分为在较低抽象层次上的子功能块和新的信道。〔建议Z.102，§ 2.2。〕

block substructure definition　功能块子结构定义

功能块子结构定义是一个功能块定义的任选部分，它定义一个功能块子结构。〔建议Z.102，§ 2.2。〕

block symbol　功能块符号

功能块符号表示在SDL/GR的一个功能块交互作用图中一个功能块的概念。〔建议Z.101，§ 3.1。〕

block tree diagram　功能块树图

功能块树图是系统划分的SDL/GR 表示。它用倒立的树图将一个系统划分成较低抽象层次上的多个功能块。〔建议Z.102，§ 3.1。〕

Boolean　布尔

是逻辑数据类型，具有TRUE、FALSE 两种值和常规的逻辑运算符集。〔建议Z.104，§ 5.4。〕

call node　调用节点

调用节点这个术语是过程调用节点的同义词。

call symbol　调用符号

调用符号这个术语是过程调用符号的同义词。

channel　信道

信道是一类实体，它将信号从一个功能块传递到另一功能块。信道也传递来自和到达环境的信号。信道是单向的，也就是说，它们只向一个方向传递信号。〔建议Z.101，§2.1。〕

channel definition　信道定义

信道定义定义一个标有名字的信道的性质。这些性质是始发功能块、目的功能块、信道可传递的信号集合，以及一可任选的信道子结构定义。目的功能块可以是系统的环境。〔建议Z.101，§2.2，建议Z.102，§2.2。〕

channel substructure信道子结构

信道子结构是对信道进行的一种划分，将信道划分成在较低抽象层次上的一组信道和功能块。〔建议Z.102，§2.2。〕

channel substructure definition　信道子结构定义

信道子结构定义是信道定义的一个任选部分，它定义信道子结构。〔建议Z.102，§2.2。〕

channel substructure diagram　信道子结构图

信道子结构图是一个信道子结构的SDL/GR表示。〔建议Z.102，§3.4。〕

channel symbol　信道符号

在SDL/GR的功能块交互作用图上，该符号表示信道或子信道。符号的箭头指向目的功能块，而符号的另一端标识始发功能块。通过信道传递的信号集合可用一相关联的信号表符号来表示。〔建议Z.101，§3.1。〕

character　字符

它是一种预定义数据类型，其字面值是长度为1的字符串字面值，其运算符是等于、不等于、以及可形成一个字符串值的并置。〔建议Z.104，§5.5。〕

charging in progress PE　进行计费PE

它是一个图形元素，指示当前正进行计费。〔建议Z.103，§6.1。〕

charstring 字符串

它是一种预定义数据类型，其字面值是CCITT 5号字母表中的字符所组成的串，其运算符是为字符而产生的串预定义生成元的那些运算符。〔建议Z.104，§5.13。〕

combined signalling sender and receiver PE 信号收发器PE

它是一个图形元素，相当于信号发送器和信号接收器的组合。〔建议Z.103，§6.1。〕

comment 注释

加到SDL图上或对图进行阐明的信息。在SDL/GR中，注释可用一对方括号括起来，通过虚线连接附于进程图的一条流线上，或者附于任一符号上。在SDL/PR中，注释可用关键字COMMENT引入。〔建议Z.101，§3.3。〕

composite operations 组合操作

表示初等SDL概念的一种标准组合的简化符号。组合操作可用一种系统的方法映射到SDL抽象语法的概念中去。〔建议Z.103，§3。〕

concrete syntactical form 具体语法形式

SDL/GR、SDL/PR和SDL/PE是SDL的具体语法形式，通过考虑概念性基础数学的SDL流图，它们可以互相映射。〔建议Z.100，§3.1。〕

concrete syntax 具体语法

各种SDL表示的具体语法是用来表示SDL的，采用SDL/GR、SDL/PR或SDL/PE形式的实际符号。〔建议Z.100，§3.1。〕

conditional expression 条件表达式

它是在IF后面跟一布尔表达式的一种表达式。它控制机器去解释THEN后面的表达式，或是跟在ELSE后面的另一表达式。〔建议Z.104，§4.10。〕

connected switching path PE 连接交换通路PE

指示终端设备和（或）信号部件之间的连接的一种图形元素。〔建议Z.103，§6.1。〕

connector 连接符

连接符是用于SDL/GR的一种符号。某一连接符（一个圆圈）或为入连接符，或为出连接符。一条流线可用一对相关联的连接符断开，其流向设定为从出连接符到它所关联的入连接符。〔建议Z.101，§3.1。〕

constant value 常量值

它或者是一个字面值，或者是一个同义词。〔建议Z.104，§2.2。〕

constant expression 常量表达式

它是一个常量值，或是一个只带有常量表达式作为参数的运算符。〔建议Z.104，§2.2。〕

continuous signal 连续信号

连续信号是一种含有条件的组合操作。当条件变为真时，进程就离开附有该连续信号的状态。〔建议Z.103，§3.3。〕

convergence 汇聚

在SDL/GR的进程图中，当两个或多个符号后面只跟一个符号时，流线就汇聚于该符号。汇聚可表现为一条流线流入另一条流线，或一个以上的出连接符和一个入连接符相关联，或几条分开的流线流进同一个符号。〔建议Z.101，§3.3。〕

create request action 创建请求动作

它是一个动作，根据一特定的进程定义，创建和启动一个新的进程实例。〔建议Z.101，§2.3。〕

create request node 创建请求节点

在进程流图或过程流图中一次跃迁中的一个节点，在这里进行创建请求动作。〔建议Z.101，§2.2。〕

create symbol 创建符号

在SDL/GR的功能块交互作用图中的一个符号，它把创建进程的进程符号和被创建进程的进程符号连接起来。箭头方向指明子孙进程，而创建请求符号的另一端指明父本进程。〔建议Z.101，§3.1。〕

create request symbol 创建请求符号

在SDL/GR的进程图上表示一个创建请求的符号。〔建议Z.101，§3.3。〕

D'

为SDL/PR关键字，引入一数值的十进制表示。由于它是缺省数制表示，所以D'也可省略。〔建议Z.104，§4.8。〕

data 数据

数据这一术语是数据项的同义词。

data item 数据项

一个数据项或者是一个变量，或者是一个值。

data type 数据类型

数据项的数据类型决定数据的范围、在该范围内值的意义、以及可作用于该数据类型的项的有效运算符集合。（亦见预定义数据类型。）〔建议Z．104，§ 1。〕

data type definition 数据类型定义

它定义一个数据类型的名字、数据值和运算符。〔建议Z．104，§2.2。〕

decision 判定

判定是在一次跃迁中一个判定节点上的一个动作。它提出一个问题，对此问题可立即得到回答，並从该节点选择若干条走出弧中的一条来继续该跃迁。〔建议Z．101，§2.3。〕

decision node 判定节点

判定节点是进程流图或过程流图中一次跃迁中的一个节点，在此节点处进行一次判定。〔建议Z．101，§2.2 。〕

decision name 判定名

判定名是与判定相关联的名字。判定的名字必须是一个易被解释者理解的问题。从判定引出的诸判定弧的名字必须构成该问题的全部可能的结果。〔建议Z．101，§2.3。〕

decision symbol 判定符号

在SDL / GR 进程图中表示SDL判定概念的符号。〔建议Z．101，§3.2。〕

declare！ 声明!

它是一个预定义运算符，隐含与产生变量实例相关联的所有数据类型。〔建议Z．104，§4.5。〕

definition 定义

定义这一术语是类型定义的同义词。

description 描述

对系统要求的具体实现应在该系统描述中予以描述。描述包括该系统的具体实现的一般参数以及其实际行为的功能描述（FD）。〔建议Z．100，§1.1。〕

divergence 发散

在SDL/ GR中，凡是一个符号后面跟有两个或多个符号的地方，一条引向该符号的流线可发散成两条或多条流线。〔建议Z.101，§3.3。〕

Duration 持续时间

表示两个时刻之间的时间间隔的预定义数据类型。〔建议Z.104，§5.11。〕

enabling condition 允许条件

允许条件是一种组合操作，用于有条件地接受输入，或根据一个条件有选择地保存某个信号。〔建议Z.103，§3.2。〕

enabling condition symbol 允许条件符号

在SDL/ GR的进程图或过程图中的符号，用来表示允许条件（当它跟随一输入符号时）或连续信号（当它跟随一状态符号时）。〔建议Z.103，§3.2。〕

environment 环境

环境这一术语是系统的环境的同义词。〔建议Z.101，§2.1。〕

environment symbol 环境符号

在SDL/ GR进程图中表示系统的环境的符号。〔建议Z.101，§§3.1.1，3.1.2。〕

environment of a system 系统的环境

系统的环境是系统的一部分，其行为不在SDL中示出，但通过向系统发送信号实例而与该系统的其余部分发生交互作用。〔建议Z.101，§2.1。〕

equivalent behaviour 等价行为

等价行为这一术语是等价功能行为的同义词。

equivalent functional behaviour 等价功能行为

从系统（或功能块，或进程）外部看来，若两个系统（相应地或功能块，或进程）对一给定的信号实例序列具有相同的响应，则它们具有等价功能行为。〔建议Z.100，§1.1。〕

EXPORT

语法结构EXPORT（变量名）用于SDL/ GR和SDL/ PR两者，以表示一变量的值的出口。〔建议Z.103，§3.1。

export 出口

出口这一术语是出口操作的同义词。

EXPORTED

它是一个SDL/ PR 关键字，在变量定义中指明该变量是出口的。〔建议Z.103，§3.1。〕

exporter 出口者

变量的出口者是一个进程实例，该进程实例拥有其值可以出口的变量。〔建议Z.103，§3.1。〕

export operation 出口操作

出口操作是组合操作，它允许一个进程出口诸数据项的值，使另一进程能访问这些值。〔建议Z.103，§3.1。〕

expression 表达式

表达式或者是一个值的名字、一个同义词名字、一个变量名字、一个条件表达式，或者是施加于一个或多个表达式的一个运算符。〔建议Z.104，§4.7。〕

extract！ 抽出!

它是在一变量后面紧跟带括号的表达式时，隐含在公理之外的运算符（除非带括号的表达式后面跟以：=，在这种情况下隐含insert！）。〔建议Z.104，§4.5。〕

flow line 流线

流线在SDL/ GR 的进程图上将每一个符号连接到它前面的符号（或诸符号）上去。〔建议Z.101，§3.3。〕

formal parameter 形式参数

形式参数是包含在进程定义或过程定义中的变量名。定义规定，当该进程被创建或相应地该过程被调用时，要将实在参数赋予诸变量。〔建议Z.101，§2.2，建议Z.103，§2.1。〕

formal parameter list 形式参数表

它是在进程定义或过程定义中诸形式参数的列表。实在参数按它们在各自表中的位置而与形式参数一一对应。〔建议Z.101，§2.2，建议Z.103，§2.1。〕

frame(1) 框(1)

在SDL / GR 的功能块交互作用图中，一个框表示一个功能块。功能块的划分由该功能块交互作用图来定义。〔建议Z.102，§3.2 。〕

frame（2） 框（2）

在SDL/GR的信道子结构图中，一个框表示一个信道。信道的划分由该信道子结构图来定义。〔建议Z.102，§3.4。〕

functional behaviour 功能行为

（见行为。）

functional block 功能块

一个功能块是一个客体，其大小便于管理，其内部有紧密关系。它有一个名字，有一组将它和别的功能块（或环境）连接起来的信道，以及有一些内部进程（或者有功能块定义）。〔建议Z.100，§2.1，Z.102，§2.1。〕

general parameters 一般参数

在系统的规格和描述中，一般参数是指诸如温度界限、结构、交换机容量、服务等级等一类事情。〔建议Z.100，§1.1。〕

generator 生成元

生成元这一术语是数据类型生成元的同义词。

graphic syntax 图形语法

（见SDL/GR。）

H′

是SDL/PR关键字，一数值的十六进制表示。〔建议Z.104，§4.8。〕

identifier 标识符

标识符是类型或类型实例的唯一的名字，由限定部分和名字部分组成。〔建议Z.100，§2.1。〕

implicit transition 隐式跃迁

在SDL/GR进程图中，对有效输入信号中的某一信号，若既没有显式的输入符号，又没有显式的保存符号附加于一状态符号，则该处存在一隐式跃迁，也就是一个输入节点被直接地连接、回到同一个状态，因此这些信号被丢失。〔建议Z.101，§3.3。〕

IMPORT

语法结构IMPORT（变量名，进程实例）用于SDL/GR和SDL/PR，以表示从某一进程实例进口值。〔建议Z.103，§3.1。〕

import 进口

进口这一术语是进口操作的同义词。

IMPORTED

它是一个SDL/PR关键字，在变量定义中指明用该变量来包含被进口的值。〔建议Z.103，§3.1。〕

importer 进口者

一个进口值的进口者是进口该值的进程实例。〔建议Z.103，§3.1。〕

Import operation 进口操作

进口操作是组合操作，它允许一个进程进口和访问由另一进程拥有的诸变量的值。〔建议Z.103，§3.1。〕

imported value 进口值

被进程看到的进口数据项的值。〔建议Z.103，§3.1。〕

IN

形式参数属性，指的是当一个值传递给一个过程时的情况。〔建议Z.103，§2.3。〕

IN/OUT

形式参数属性，指的是当一个形式参数名字作为变量的一个同义词使用时的情况。〔建议Z.103，§2.3。〕

in-connector 入连接符

一条流线可以用一对相关联的连接符断开，其流向设定为从出连接符到它所关联的入连接符。〔建议Z.101，§3.3。〕

incoming channel 进入信道

一条进入信道是在一信道被划分时形成的一条新信道。进入信道向由信道划分形成的新功能块传递由原被划分信道传递的全部信号。〔建议Z.102，§2.1。〕

infix operator 中缀运算符

为SDL/PR的预定义二元运算符(=> OR XOR AND IN /= => < <= >= = + - // # / MOD REM) 之一，它置于两个参数之间，而不是在带括号的参数前面；或为一元前缀运算符 (+ - NOT)之一。〔建议Z.104，§4.5。〕

inlet 入口处

宏的入口处是一个点，一条线在该处进入该宏。〔建议Z .103，§ 4.1。〕

input 输入

输入这一术语是输入动作的同义词。

input action 输入动作

输入动作根据信号名来接收和消耗一个输入信号，并允许进程实例解释该输入动作，来访问该输入信号中所包含的信息。〔建议Z .101，§ 2.3。〕

input node 输入节点

输入节点是进程流图或过程流图中的一个节点，一个输入动作在该处进行，且具有该输入动作所消耗的信号的相同名字。〔建议Z .101，§ 2.2。〕

input port 输入端口

进程的输入端口按信号到达的先后次序接收和保留信号，直到它们被一个输入动作所消耗。〔建议Z .101，§ 2.3。〕

input signal 输入信号

进程的一个输入信号是该进程能在其任一状态节点接收的有名字的信号集合中的一个信号。进程所能接收的有效输入信号集合是出现在该进程任一输入节点中的所有信号名字的集合。〔建议Z .101，§ 2.3。〕

input symbol 输入符号

在SDL/GR进程图中表示输入的SDL概念的符号。〔建议Z .101，§ 3.3。〕

insert! 插入!

它是在一变量后面紧跟带括号的表达式并再跟:= 时，隐含在公理之外的运算符。〔建议Z .104，§ 4.5。〕

instance 实例

一个类型的实例是一个值，它具有该类型的全部性质，且能用一个标识符和同一类型的其它实例区别开来。〔建议Z .100，§ 2.1。〕

instantiation 实例产生

实例产生就是从一个类型创建一个实量的实例。〔建议Z .100，§ 2.1。〕

Integer 整数

整数类型由值的集合-（无穷大）、…、-2、-1、0、+1、+2、…、+（无穷大）及其常规的数学运算符即+、-、乘、除等来定义。〔建议Z.104，§5.1。〕

label 标号

标号是一个任选的名字，附加于宏的一个入口处或出口处。〔建议Z.103，§4.2。〕

level 层次

层次这一术语是抽象层次的同义词。

level of abstraction 抽象层次

一个抽象层次是一个功能块树图的层次之一。一个系统的描述就是在最高抽象层次上的一个功能块，并表示为在功能块树图中顶部的一个功能块。〔建议Z.102，§3.1。〕

literal 字面值

一个字面值是指一个值本身。字面值12、B′1100、O′14和H′C全都指同样的整数值。〔建议Z.104，§4.4。〕

macro 宏

宏是多个语法项目的一个标以名字的集合，由SDL用户定义。在考虑该SDL表示的意义之前要用这些语法项目去代替所使用的宏名字。〔建议Z.103，§4。〕

macro definition 宏定义

SDL/GR中的一个宏定义是任一SDL/GR图中一个标有名字的部分。它具有用线条表示的入口处和出口处，（分别地）指明是流向和流自该部分。这些线条可加标号。〔建议Z.103，§4.2。〕

macro symbol 宏符号

它是一个用于SDL/GR中的符号，用名字来指出对宏定义的引用。〔建议Z.103，§4.2。〕

name 名字

名字这一术语是一个标识符的名字部分的同义词。

name part 名字部分

一个标识符的名字部分是自然语言中一个有意义的词组，它和该标识符的限定部分合在一起，可用来标识一个实量的类型或实例。〔建议Z.100，§2.1。〕

Natural　自然数

自然数类型是整数类型中的一个范围类型，具有值0、1、2、…直至无穷大。〔建议Z.104，§5.6。〕

newtype　新类型

一个新类型引入了字面值和运算符集合，这些字面值和运算符集合是不同于任何其它字面值和运算符的（它们可有相同的名字，因而不同的标识必须通过限定符来加以分辨）。新类型的诸性质要用公理中的字面值和运算符来定义。〔建议Z.104，§4.3。〕

node　节点

一个节点是进程流图上一个标有名字的地方，它通过弧和其它节点相连。SDL中节点的种类有状态节点、输入节点、任务节点、输出节点、判定节点、起动节点、停止节点、过程启动节点、过程调用节点、过程返回节点和创建请求节点。〔建议Z．101，§§2.2。〕

note　注

是SDL/ PR 中的一个注解，它只与SDL/ PR 表示有关。一个注是由/ * 和* / 括起来的一个正文串。〔建议Z．101，§4.3。〕

O′

是一个SDL/ PR 关键字，一数值的八进制表示。〔建议Z．104，§4.8。〕

OFFSPRING

进行创建的进程的OFFSPRING 是一个数据项，它与被此创建进程最近所创建的进程的SELF数据项具有相同的值。若一进程尚未创建任何进程，则其OFFSPRING数据项是未定义的。〔建议Z.101，§2.3。〕

operator　运算符

当一个运算符施加于一个或多个值时，它得出一个由该运算符在公理中的用法所决定的值。符号+ - * / 是算术运算符。〔建议Z.104，§4.5。〕

operator typing　运算符定类

它定义运算符所作用的数据项的数据类型及运算结果值（若有的话）的数据类型。〔建议Z.104，§4.5。〕

option　任选

任选是在进程定义中的一个具体语法结构，它允许在该进程流图被解释之前选择不同的行为。〔建议Z.103，§5.1。〕

option expression　任选表达式

包含在任选中的一个表达式，通过对它的求值来确定要选择哪一个行为。〔建议Z．103，§5.2。〕

option symbol　任选符号

该符号在SDL/ GR进程图或过程图上表示任选。〔建议Z.103，§5.2。〕

out-connector　出连接符

一条流线可以用一对相关联的连接符断开，其流向设定为从出连接符到它所关联的入连接符。〔建议Z.101，§3.3。〕

outgoing channel　走出信道

一条走出信道是在一信道被划分时形成的一条新信道。信道划分形成了一些新的功能块，走出信道从这些新功能块传递由原被划分信道传递的全部信号。〔建议Z.102，§2.1。〕

outlet　出口处

宏的出口处是一个点，一条线在该处离开该宏。〔建议Z.103，§4.2。〕

output　输出

输出这一术语本身是输出动作的同义词。

output action　输出动作

输出是一次跃迁中的一个动作，它产生一个信号。该信号在别的地方又作为输入信号而动作。〔建议Z.101，§2.3。〕

output node　输出节点

进程流图上的一个节点，一个输出动作在该处进行，且具有它所产生的信号的同一名字。〔建议Z．101，§2.2。〕

output symbol　输出符号

在SDL/ GR进程图中表示输出的SDL概念的符号。〔建议Z.101，§3.3。〕

PARENT

进程的PARENT数据项就是其父本进程的SELF数据项。父本进程是指解释启动该进程的创建请求动作的那个进程。〔建议Z.101，§2.3。〕

partitioning 划分

划分是将复杂和（或）庞大系统的行为精心地分成子系统，以提供该系统行为的逻辑划分和提供同一系统的不同的抽象观察角度。〔建议Z.102，§2.1。〕

passive operator 被动运算符

该运算符只要求有作为参数的值，並产生一个值作为结果。一个被动运算符不能改变与变量相关联的诸值。〔建议Z.104，§4.5。〕

pictorial element (PE) 图形元素(PE)

它是在SDL/ PE 的状态图形中使用的许多标准化图形形式中的一种图形形式，用来表示交换系统的概念。〔建议Z.103，§ 6 。〕

pId

用来标识进程实例的预定义数据类型。〔建议Z .104，§5.8。〕

predefined data type 预定义数据类型

为了简化描述，预定义数据类型既用于数据类型的预定义名字，又用于数据类型生成元的预定义名字。布尔、字符、字符串、持续时间、整数、自然数、PId 、实数、时刻和计时器是预定义的数据类型名字，数组、幂集、和串是预定义的数据类型生成元名字。〔建议Z.104，§ 5 。〕

Powerset 幂集

它是一个预定义数据类型生成元，它所产生的诸数据类型所具有的值，是值的数学上的有序集合。每一幂集值是该数据类型的值的集合，用来使幂集生成元参数化。〔建议Z.104，§5.7。〕

procedure 过程

进程流图的一段，可以看成是独立的。过程在一个地方予以定义，但可多次加以引用，甚至可在不同的进程中引用。受过程解释影响的信号和变量要由参数传递来控制。〔建议Z.103，§ 2 。〕

procedure call 过程调用

过程调用是引用一标有名字的过程的手段，以便对该过程进行解释，並向它递交参数。〔建议Z.103，§2.1。〕

procedure call node 过程调用节点

过程调用节点是在进程流图或过程流图中的一个节点，在该处进行一次过程调用。〔建议Z . 103，§2.1。〕

procedure call symbol　过程调用符号

表示一次过程调用的SDL / GR 符号。〔建议 Z.103, § 2.2 。〕

procedure definition　过程定义

过程定义定义了进程流图的一段。该定义以下列内容与过程流图相关联：过程名、形式参数表、附加保存组，以及任选的、更进一层的过程定义和数据定义。〔建议 Z.103, § 2.1。〕

procedure diagram　过程图

过程图是过程流图的SDL / GR 表示。〔建议 Z.103, § 2.2。〕

procedure graph　过程流图

过程流图是由有向弧连接起来的一个流图，用以描述一个过程的行为，並能形成进程流图的一段。〔建议 Z.103, § 2.1。〕

procedure return　过程返回

（见返回。）

procedure start node　过程启动节点

过程启动节点是一个节点，並且是在过程流图中，在这里通过对该过程的调用而开始解释该过程。〔建议 Z.103, § 2.1。〕

procedure start symbol　过程启动符号

在SDL / GR 过程图中表示一过程启动节点的符号。〔建议Z .103, § 2.2。〕

process　进程

一个进程执行一项功能。它要求各项信息来执行各种子功能。执行诸子功能的次序取决于该信息对进程生效时间的先后。从SDL的角度来说，一个进程是一类实体，其诸实例或者处于某个状态等待着输入，或者处于一次跃迁中。进程这一术语本身是进程实例的同义词。〔建议Z .101, § 2.1。〕

process definition　进程定义

进程类的一个类型的行为在进程定义中描述，表现为一个由一些输入、保存、状态、跃迁、判定、任务和输出组成的闭合的有向流图。〔建议Z . 101, § 2.1。〕

process diagram　进程图

进程图是进程流图的SDL/ GR 表示。〔建议 Z.101, § 3.2。〕

process graph 进程流图

进程流图是一个由有向弧连接起来的，用以描述进程行为的流图。〔建议Z.101，§2.2。〕

process instance 进程实例

进程实例是一个动态地创建的进程的实例。〔建议Z.101，§2.3。〕

process substructure 进程子结构

一个进程的进程子结构是将该进程划分成诸子进程，並将这些子进程分配到子功能块中。〔建议Z．102，§2.2。〕

process substructure definition 进程子结构定义

进程子结构定义是进程定义中的一个任选部分，它定义一个进程的进程子结构。〔建议Z．102，§2.2。〕

process symbol 进程符号

它是在SDL/GR的功能块交互作用图或进程树图上的一种符号，用来表示零个或多个进程实例。进程符号含有标识该进程定义的进程名字和一个形式参数表。〔建议Z.101，§3.1，建议Z.102，§3.3。〕

process tree diagram 进程树图

SDL/GR进程树图表示将功能块中的一个进程划分成子进程。将子进程分配给子功能块则由进程树图上的分配符号示出。该图的形状象一棵倒立的树。〔建议 Z.102，§3.3。〕

qualifiers 限定符

限定符这一术语是一个标识符的限定部分的同义词。

qualifying part 限定部分

一个标识符的限定部分是一种信息，必须把它加到该标识符的名字部分才能形成一个唯一的名字。标识符的限定部分可以从名字部分用法的上下文关系导出。〔建议 Z.100，§2.1。〕

Real 实数

实数类型由－（无穷大）和＋（无穷大）之间全部值的集合以及常规的数学运算即＋、－、乘、乘方等来定义。〔建议Z.104，§5.2。〕

reserved switching path PE　保留交换通路PE

它是一个图形元素，表示终端设备和（或）信号部件之间保留的连接通路。〔建议Z.103，§6.1。〕

RESET

该运算符是为计时器数据类型定义的，它使计时器清零。〔建议Z.104，§5.12。〕

retained signal　保留信号

当一信号到达一进程时，它应被该进程接收并保留（它处在进程之外，因此还没有被它消耗掉）。〔建议Z.101，§2.1。〕

return　返回

过程的返回就是破坏在过程启动时创建的诸变量和同义词，接着结束该过程启动的解释。〔建议Z.103，§2.1。〕

return node　返回节点

返回节点是过程流图中的一个节点，在这里进行从该过程的返回。〔建议Z.103，§2.1。〕

return symbol　返回符号

在过程图中，该符号表示从该过程的一次返回。〔建议Z.103，§2.2。〕

reveal attribute　透露属性

被进程所拥有的变量可以有一种透露属性，在此情况下同一功能块中的另一进程能视见与该变量相关联的值。〔建议Z.100，§3.1。〕

save　保存

保存是当进程处于不能输入某个信号的某一特定状态时，推迟对该信号的确认。〔建议Z.101，§2.2。〕

save-signal-set　保存信号组

进程的某一状态的保存信号组是该状态诸保存信号名字的集合。〔建议Z.100，§2.2。〕

save symbol　保存符号

SDL/GR进程图中的一个符号，它表示保存的SDL概念。〔建议Z.101，§3.3。〕

SDL/GR

SDL 的作图表示法。〔建议Z.100，§3.1。〕

SDL/PR

SDL 的正文短语表示法。〔建议Z.100，§3.1。〕

SDL/PE

SDL 的图形元素形式，它是一种面向状态的SDL/GR的扩充。〔建议Z.100，§3.5。〕

SELF

进程的SELF数据项是唯一的进程实例值，它区别于任何其它进程实例。〔建议Z.101，§2.3。〕

SENDER

进程的SENDER数据项等于最近被消耗的信号的始发进程数据项。〔建议Z.101，§2.3。〕

SET

该运算符系为计时器数据类型而定义，用它来设置计时器。〔建议Z.104，§5.12。〕

shared value 共享值

共享值是关联于某一变量的值，它在一个进程中具有透露属性，因而可被另一进程视见。〔建议Z.101，§2.3。〕

SIGNAL

过程的一个信号参数的形式参数属性。〔建议Z.103，§2.3。〕

signal 信号

信号这一术语是信号实例的同义词。

signal definition 信号定义

信号定义定义一个名字作为信号名，並把一个具有零个或多个数据类型的表与该信号名相关联。〔建议Z.101，§2.2。〕

signal instance 信号实例

信号实例是一个信号的实例，它从另一进程实例的输出动作或从环境传送信息到某一进程实例。另外，信号实例也用于从一个进程的输出动作传送信息到环境。〔建议Z.101，§2.3。〕

signalling receiver PE　信号接收器PE

一个表示信号接收器的图形元素。〔建议Z.103，§6.1。〕

signalling sender PE　信号发送器PE

一个表示信号发送器的图形元素。〔建议Z.103，§6.1。〕

signal list　信号表

能被一个信道传递或能在一个功能块内部从一个进程传递到另一进程的所有信号的名字表。〔建议Z.101，§2.2。〕

signal list symbol　信号表符号

SDL/GR功能块交互作用图上的符号，表示一个信号表，並有一信道或信号路径符号与之相关联。〔建议Z.101，§3.1。〕

signal route symbol　信号路径符号

SDL/GR交互作用图上的符号，指明在一个进程和同一功能块内的另一进程之间，或者和连接于该功能块的诸信道之间的信号的流动。〔建议Z.101，§3.1。〕

specification　规格

一个系统的要求在该系统的规格中定义。规格由系统所要求的一般参数和它所要求的行为的功能规格（F-S）组成。〔建议Z.100，§1.1。〕

specification and description language （SDL）　规格与描述语言（SDL）

用来表示存储程序控制（SPC）交换系统内部逻辑过程的功能规格和功能描述的CCITT语言。〔建议Z.100，§1.1。〕

start action　启动动作

进程的启动动作在任何其它动作之前首先得到解释。启动动作给该进程的形式参数赋以初始值。〔建议Z.101，§2.3。〕

start node　起动节点

进程流图中的起动节点是唯一不跟随其它节点的节点。在一个进程流图中只有一个起动节点，而进程的解释也在此节点开始。起动节点是输入动作进行的地方。〔建议Z.101，§2.2。〕

start symbol　起动符号

SDL/GR进程图中的一个符号，表示一个起动节点。〔建议Z.101，§3.3。〕

state 状态

状态是一种状况，在这里进程的动作暂停，而等待一个输入信号。〔建议Z.101，§2.1。〕

state node 状态节点

进程流图或过程流图中的一个节点，该进程在此处进入一个状态。〔建议Z.101，§2.2。〕

state picture 状态图形

状态图形是由一些图形元素组成的状态符号，用来把SDL/GR扩充到SDL/PE。〔建议Z.103，§6。〕

state symbol 状态符号

SDL/GR进程图中的一个符号，表示一个或多个状态的SDL概念。〔建议Z.101，§3.3。〕

stop 停止

它是终止进程实例的一个动作。〔建议Z.101，2.3。〕

stop node 停止节点

进程流图中的一个节点，在这里进程停止了。〔建议Z.101，§2.2。〕

stop symbol 停止符号

SDL/GR进程图上的一个符号，表示停止。〔建议Z.101，§3.1。〕

String 串

是一种预定义数据类型生成元。它产生的数据类型所取的值，是用来把串生成元参数化的数据类型的项目表。〔建议Z.104，§5.9。〕

Struct 构件

带构件的数据类型定义隐含地引入域名字的数据类型及隐含的公理，后者用extract!（抽出）和insert!（插入）结构的部分值来定义域名字的用法。〔建议Z.104，§4.3。〕

sub-block 子功能块

子功能块是包含在另一功能块中的功能块。子功能块是在一功能块被划分时形成的。〔建议Z.102，§2.1。〕

sub-block definition　子功能块定义

子功能块定义定义一功能块，并形成功能块子结构定义的一部分。〔建议Z.102，§2.2。〕

sub-block symbol　子功能块符号

SDL/GR功能块交互作用图或功能块树图中的一个符号，它表示一个子功能块，并完全等同于一个功能块符号。〔建议Z.102，§3.2。〕

sub-channel　子信道

子信道是在功能块被划分时形成的信道。〔建议Z.102，§2.1。〕

sub-process　子进程

子进程是在进程被划分时形成的进程。〔建议Z.102，§2.3。〕

subscriber line PE　用户线P E

表示一条用户线的图形元素。〔建议Z.103，§6.1。〕

switchboard PE　电键板P E

表示一终端设备的电键板的图形元素。〔建议Z.103，§6.1。〕

switching module P E　交换模块P E

表示一定换模板的图形元素，它关联于一条连接好的或保留的交换通路。〔建议Z.103，§6.1。〕

synonym definition 同义定义

它是对数据值的一个名字的定义。〔建议Z.104，§4.12。〕

syntax diagram 语法图

语法图是用来定义SDL/PR具体语法的图。〔建议Z.100，§3.4。〕

syntype　同义类型

同义类型引入一组值，它们对应于父本新类型的值的一个子集。Access!（访问）、declare!（声明）和assign!（赋值）是用于同义类型的仅有的运算符，因为在赋值之前和从同义类型变量抽出之后的值总是其父本新类型的值。〔建议Z.104，§4.3。〕

system 系统

系统是功能块的一个集合，这些功能块用信道彼此连接起来，并用信道与环境相连接。系统这个术语本身是系统实例的同义词。〔建议Z .101，§2.1。〕

system boundary 系统边界

系统边界是处在用SDL定义的诸功能块和环境之间的分界线。〔建议Z .101，§2.3。〕

system definition 系统定义

系统定义定义一个系统的性质。系统的性质是指该系统的诸功能块、信道、与信道相关联的信号、数据类型和同义词。〔建议Z .101，§2.2。〕

task 任务

任务是跃迁中的一个动作，它或者含有一系列的赋值语句、设置语句或复位语句，或者含有非形式正文。任务的解释取决于系统所掌握的信息，且能作用于该信息。〔建议Z .101，§2.2。〕

task node 任务节点

它是进程流图或过程流图中的一个节点，在这里执行一次任务。〔建议Z .101，§2.2。〕

task symbol 任务符号

SDL/GR进程图中的一个符号，表示一个任务的SDL概念。〔建议Z .101，§3.3。〕

terminal equipment PE终端设备PE

已有的六个图形元素中的一个，这些图形元素表示下列类型的终端设备：挂机的电话机、摘机的电话机、中继线、用户线、电键板（或控制台）、其它。〔建议Z .103，§6.1。〕

text extension symbol 正文扩展符号

此符号所关联的正文被认为是属于SDL/GR中该正文扩展符号所依附的那个符号。〔建议Z .101，§ 3。〕

Time 时刻

它属于预定义数据类型，表示绝对时间。〔建议Z .104，§5.10。〕

Timer 计时器

它属于预定义数据类型，用于计时器，并为计时器定义运算符SET和RESET。〔建议Z .104，§5.12。〕

time supervision of a process P E　进程时间监视P E

表示一监视计时器运行的图形元素。〔建议Z .103, § 6.1。〕

transition　跃迁

跃迁是一个动作序列。它出现在当一进程实例响应一输入而从一个状态变到另一个状态的时候。〔建议Z .101, § 2.2。〕

transition string　跃迁串

跃迁串是在一输入节点和其后面的状态节点、停止节点或过程返回节点之间的零个或多个动作的序列。〔建议Z .101, § 2.2。〕

trunk P E　中继线P E

表示一中继线接口的图形元素。〔建议Z .103, § 6.1。〕

type　类型

一个类型是实体的一组性质。SDL 中类型的类有：功能块、信道、数据项、过程、进程、信号和系统。一个类型可由同一类的各实体组成，在这种情况下这些实量就是子类型(例如，一个功能块由诸子功能块组成)。〔建议Z .100, § 2.1。〕

type definition　类型定义

类型定义定义一个类型的性质。〔建议Z .100, § 2.1。〕

valid input signal　有效输入信号

在进程的每个状态上，对能够输入的诸信号都有一个信号名集合。有效输入信号是一个信号名，它是某个集合中的一个成员。〔建议Z .101, § 2.3。〕

value　值

一个数据类型的数据值是关联于该数据类型一个变量的诸值之一，并能和一个要求具有该数据类型的值的运算符一起使用。〔建议Z .104, § 1。〕

variable　变量

变量是由进程拥有的一个实量，它能被赋予某个值。当一变量被访问时，它给出已赋予它的最新的值。〔建议Z .101, § 2.3。〕

variable definition　变量定义

变量定义定义一个数据类型的实例。〔建议Z .101, § 2.2。〕

view 视见

如果一变量的值被拥有该变量的进程所透露且另一进程能访问该值，则该变量能被视见。〔建议Z.101，§2.3。〕

view definition 视见定义

视见定义是进程定义的一部分，它定义一些变量。这些变量是由另一进程所拥有，且只被含有该视见定义的进程所视见。〔建议Z.101，§2.2。〕

viewing expression 视见表达式

视见表达式用于一表达式之内，以指明得到了一视见变量的最新值。〔建议Z.101，§2.3。〕

附 件 B

（属建议Z.100至Z.104）

抽象语法概要

这个概要包含了建议Z.101、Z.102、Z.103和Z.104中所定义的全部抽象语法和有关的好的格式规则。语法用一种扩充的BNF[1]范式来说明，如建议Z.200中所定义的；而规则用中文给出。由于这是一个概要，如果要了解准确的定义，请参考各个建议。

B.1 系统

（1）＜系统定义＞::=
(Z.101) 系统名字
(Z.101) 〔＜功能块定义＞〕+
(Z.101) 〔＜信道定义＞〕*
(Z.101) 〔＜信号定义＞〕*
(Z.104) 〔＜数据定义＞〕*
(Z.103) 〔＜过程定义＞〕*

好的格式（Wellformedness）

对于每个信道定义中信号表中的每个信号名字，结构定义中必须含有其对应的信号定义（建议Z.101）。

1） BNF = 巴科斯－诺尔范式。

B．2　功能块

（2）＜功能块定义＞::=
(Z．101) 功能块名字
(Z．101) 〔＜进程定义＞〕*
(Z．101) 〔＜信号定义＞〕*
(Z．104) 〔＜数据定义＞〕*
(Z．103) 〔＜过程定义＞〕*
(Z．102) 〔＜功能块子结构定义＞! 〕

好的格式

在所包含的各进程中，对与输入和输出操作有关的每一信号名，应在该功能块中含有其信号定义，或者在其包围结构中含有其信号定义。

如果一功能块定义含有功能块子结构定义，它就不一定要含有进程定义（建议Z .102）。

B．3　功能块子结构

（3）　＜功能块子结构定义＞::=
(Z．102) 〔＜子功能块定义＞〕+
(Z．102) 〔＜子信道定义＞〕
(Z．102) 〔＜信道定义＞〕+
(Z．102) 〔＜进程子结构定义＞〕*
(Z．102) 〔＜信号定义＞〕*
(Z．104) 〔＜数据定义＞〕*
(Z．103) 〔＜过程定义＞〕*
(3.1)　＜子功能块定义＞::=
(Z．102) ＜功能块定义＞
(3.2) ＜子信道定义＞::=
(Z．102) ＜信道定义＞

好的格式

对包围功能块中的每一进程定义，该功能块子结构定义必须含有一进程子结构定义。

对通向包围功能块诸信道的每一终止端点，必须至少有一个以该端点为始发端点的子信道定义；而反过来，对该包围功能块的所有始发信道端点来说，必须至少有一个以该端点为终止端点的子信道定义。当一信道通向（或发自）该包围功能块时，具有相同端点的诸子信道定义的信号表的并集必须与该信道的信号表完全一致。此外，始发自一终止端点的诸信道定义的各信号表必须互不重迭（建议Z .102）。

包含在子结构中的信道定义必须使各子功能块互相连接起来，所有子信道必须将包围功能块的信道端点连接到各子功能块。

B．4　信道

（4）　＜信道定义＞::=
(Z．101) 信道名
(Z．101) 〔始发功能块标识符! ENVIRONMENT〕
(Z．101) 〔目的功能块标识符! ENVIRONMENT〕
(Z．101) ＜信号表＞
(Z．102) 〔＜信道子结构定义＞! 〕

(4.1) ＜信号表＞::=
(Z．101) 〔信号 名字〕+

好的格式

诸始发功能块标识符或目的功能块标识符中，只有一个标识符可引用环境来代替。

与一个信道相关联的始发功能块标识符和目的功能块标识符必须不同，且每一个标识符必须是该系统内某一功能块的标识符，或是该功能块内某一子功能块的标识符，或必须是ENVIRONMENT（建议Z.101）。

B．5 信道子结构

(5) ＜信道子结构定义＞::=
(Z．102) ＜进入信道定义＞
(Z．102) ＜走出信道定义＞
(Z．102) 〔＜信道定义＞〕*
(Z．102) 〔＜功能块定义＞〕+
(Z．102) 〔＜信号定义＞〕*
(Z．104) 〔＜数据定义＞〕*
(Z．103) 〔＜过程定义＞〕*
(5.1) ＜进入信道定义＞::=
(Z．102) ＜信道定义＞
(5.2) ＜走出信道定义＞::=
(Z．102) ＜信道定义＞

好的格式

进入信道定义具有与包围信道定义相同的始发端点，走出信道具有与包围信道定义相同的终止端点。进入信道定义和走出信道定义所具有的信号表与包围信道定义的信号表完全一样（建议Z.102）。

B．6 信号

(6) ＜信号定义＞::=
(Z．101) 信号 名字
(Z．101) 〔数据类型 标识符〕*

好的格式

所用的数据类型标识符必须是预定义的，或者是在该包围结构中数据类型定义的名字。

B．7 进程

(7) ＜进程定义＞::=
(Z．101) 进程 名字
(Z．101) ＜实例的数目＞
(Z．101) 〔＜形式参数＞〕*
(Z．104) 〔＜数据定义＞〕*
(Z．101) 〔＜视见定义＞〕*
(Z．103) 〔＜过程定义＞〕*
(Z．101) ＜进程流图＞

(7.1) ＜实例的数目＞::=
　　＜整数值标识符＞＜整数值标识符＞
(7.2) ＜视见定义＞::=
(Z．101) ＜变量标识符＞
　　＜类型标识符＞
　　＜进程定义标识符＞

好的格式

所引用的全部类型名字必须是预定义的，或是在进程定义中定义的，或是在包围结构的概念中定义的。

B．8　进程子结构定义

(8) ＜进程子结构定义＞::=
(Z．102) 进程名字
(Z．102) 〔子进程名字子功能块名字〕*

好的格式

进程名字必须是包含在包围功能块中的某一进程定义的名字。所包含的子进程名字必须是包含在（具有所关联的子功能块名字的）子功能块中的诸子进程的名字。

在进程的有效输入信号集中的每一信号名必须只出现在诸子进程的一个有效输入信号集中。附属于进程的一个输出节点的每一信号名必须至少附属于一个子进程中的一个输出节点。

B．9　进程流图

(9)　＜进程流图＞::=
(Z．101) ＜进程启动节点＞＜进程跃迁＞
(9.1) ＜进程跃迁＞::=
　　＜跃迁串＞＜状态节点＞
　　！＜跃迁串＞＜停止节点＞

B．10　跃迁串

(10)　＜跃迁串＞::=
(Z．101) ＜任务节点＞＜跃迁串＞
(Z．101) ！＜输出节点＞＜跃迁串＞
(Z．101) ！＜判定节点＞
(Z．101) ！＜创建请求节点＞＜跃迁串＞
(Z．103) ！＜过程调用节点＞＜跃迁串＞
(Z．101) ！

B．11　起动节点

(11)　＜起动节点＞::=

B．12　状态节点

(12)　＜状态节点＞::=
(Z．101) 状态名字
(Z．101) ＜保存信号组＞
(Z．101) 〔＜输入节点＞〕+

(12.1) ＜保存组＞ :: =
(Z．101) 〔信号标识符〕 *

B．13 停止节点

(13)＜停止节点＞:: =

B．14 任务节点

(14)＜任务节点＞:: =
(Z．101) 〔〔＜语句＞〕 +! 〔＜非形式正文＞〕 * 〕

B．14．1 语句

(14.1)＜语句＞::>=
(Z．101)＜设置语句＞
(Z．101)！＜复位语句＞
(Z．101)！＜赋值语句＞

(14.1.1)＜设置语句＞:: =
(Z．101)＜时间表达式＞计时器标识符
(14.1.2)＜复位语句＞:: =
(Z．101) 计时器标识符
(14.1.3)＜赋值语句＞:: =
(Z．101)＜赋值运算符＞
(Z．101)＜变量标识符＞
(Z．101)＜表达式＞

B．15 输出节点

(15) ＜输出节点＞:: =
(Z．101)＜信号标识符＞
(Z．101) 〔＜表达式＞〕 *
(Z．101)＜目的块＞
(15.1)＜目的块＞:: =
(Z．101)＜*PId*表达式＞

B．16 输入节点

(16)＜输入节点＞:: =
(Z．101) 输入信号标识符＜进程跃迁＞
(Z．103)！输入信号标识符＜过程跃迁＞

好的格式

进程跃迁只可出现在进程定义中，而过程跃迁只可出现在过程定义中。

B．17 判定节点

(17)＜判定节点＞:: =
(Z．101)＜问题＞
(Z．101)＜回答＞〔＜回答＞〕 +

(17.1)＜问题＞:: =
（Z．101）＜表达式＞
　！＜非形式正文＞
(17.2)＜回答＞:: =
（Z．101）〔＜值标识符＞〕+
　〔＜过程跃迁＞!＜进程跃迁＞〕

好的格式

判定节点必须跟有两个或多个回答（建议Z．101）。
判定名字中的表达式和各回答中的表达式必须具有相同的类型。

B．18　过程调用节点

(18)＜过程调用节点＞:: =
（Z．103）过程标识符
（Z．103）〔＜表达式＞〕*
（Z．103）〔信号标识符〕*
（Z．103）＜附加保存组＞
(18.1)＜附加保存组＞:: =
　〔＜信号名字＞〕*

好的格式

每个表达式必须与（具有该过程名字的）过程定义中相应的形式参数是同一类型。
对过程定义中非类型信号的每一形式参数必须有一个表达式，而对类型信号的每一形式参数则必须有一个信号标识符。

B．19　创建请求节点

(19)＜创建请求节点＞:: =
（Z．101）进程标识符
（Z．101）〔＜表达式＞〕*

好的格式

对于由进程标识符所引用的进程定义中的每一形式参数，必须要有一个表达式，而且每个表达式必须在类型上与所关联的形式参数一致（建议Z．101）。

B．20　过程

(20)＜过程定义＞:: =
（Z．103）过程名字
（Z．103）〔＜数据定义＞〕*
（Z．104）〔＜变量定义＞〕*
（Z．103）〔＜过程定义＞〕*
（Z．103）〔＜形式参数＞〕*
（Z．103）＜过程流图＞

好的格式

出现在过程流图中的每一信号标识符必须也作为类型信号的一个形式参数出现。

B．21 过程流图

(21) ＜过程流图＞::=
(Z．103)＜过程启动节点＞＜过程跃迁＞
(21.1)＜过程跃迁＞::=
(Z.103)＜跃迁串＞＜状态节点＞
(Z．103)！＜跃迁串＞＜返回节点＞

B．22 过程启动节点

(22) ＜过程启动节点＞::=

B．23 返回节点

(23) ＜返回节点＞::=

B．24 数据定义

(24)＜数据定义＞::=
(Z.104)＜数据类型定义＞
(Z.104)！＜同义定义＞

B．25 变量定义

(25) ＜变量定义＞::=
(Z.101)变量名字
(Z.101)类型标识符
(Z.101)〔＜透露属性＞！〕

B．26 标识符

(26) ＜标识符＞::=
(Z.100)＜限定符＞名字

B．27 限定符

(27) ＜限定符＞::=
(Z.100)＜结构名字＞＜实量类型名字＞

B．28 结构名字

(28)＜结构名字＞
(Z.100)系统名字
(Z.100)〔功能块名字〕*〔信道名字〕*
(Z.100)〔信号名字〕*〔进程名字〕*
(Z.100)〔过程名字〕*〔类型名字〕*

B．29 实体类型名字

(29) ＜实体类型名字＞::=
(Z .100) 系统! 功能块! 进程
(Z .100) ! 过程! 信号! 信道
(Z .100) ! 任务! 判定! 起动
(Z .100) ! 停止! 创建请求
(Z .100) ! 返回! 过程启动
(Z .100) ! 调用! 变量! 数据类型
(Z .100) ! 运算符! 值

B．30 数据类型定义

(30) ＜数据类型定义＞::=
(Z .104) 数据类型 名字＜数据类型描述＞
(Z .104) ! 数据类型 名字＜同义类型描述＞

B．31 数据类型描述

(31) ＜数据类型描述＞::=
(Z .104) 〔值 名字〕 * *
(Z .104) 〔＜运算符引入＞〕 +
(Z .104) 〔类型公理〕 *

注 1 - 符号〔〕 * *表示项目数为零个或多个且项目数可为无穷多个的表。除了表可以是无穷的这一性质以外，它和〔〕 *是一样的。

注 2 一类型公理在本抽象语法中不进一步定义，但对公理是有具体语法的。

注 3 - 加号（+）表示该组群必须出现，且可再重复任意次。如语法元素用方括号（〔和〕）来组群，则该组群是任选的。

B．32 运算符引入

(32) ＜运算符引入＞::=
(Z .104) ＜通用运算符＞
(Z .104) ! 运算符名字＜运算符定类＞

B．33 通用运算符

(33) ＜通用运算符＞::=
(Z .104) ＜变量运算符＞
(Z .104) ! ＜比较符＞

B．34 变量运算符

(34) ＜变量运算符＞::=
(Z .104) 赋值
(Z .104) ! 访问
(Z .104) ! 声明

B．35 比较符

(35) ＜比较符＞::=
(Z .104) 小于
(Z .104) ! 大于
(Z .104) ! 等于

B．36 运算符定类

(36) ＜运算符定类＞::=
(Z .104) ＜数据类型表＞＜结果类型标识符＞

B．37 数据类型表

(37) ＜数据类型表＞::=
(Z .104) 〔＜数据类型标识符＞〕+

好的格式 (对于30－37)

所有的值名字在类型描述中都必须是互斥的。
所有的运算符名字在类型描述中都必须是互斥的。
每次运算符定类时，在数据类型表中必须有一个数据类型标识符是该被定义类型的数据类型标识符。

B．38 同义类型抽象语法

(38) ＜同义类型描述＞::=
(Z .104) ＜父本类型标识符＞
〔＜值标识符＞〕*

好的格式

值标识符必须全部是父本数据类型标识符中的值标识符集合中的成员。

B．39 非形式正文

(39) ＜非形式正文＞::=
(Z .104) 易懂的名字

B．40 表达式

(40) ＜表达式＞::=
(Z .104) ＜初等量＞
(Z .104) ！＜运算＞
！＜条件表达式＞

B．40.1 运算

(40.1) ＜运算＞::=
＜运算符＞〔＜表达式＞〕*

B．40.2 条件表达式

(40.2) ＜条件表达式＞::=
(Z .104) ＜布尔表达式＞
(Z .104) ＜表达式＞＜表达式＞

B．41 初等量

(41) ＜初等量＞::=
(Z .104) 同义标识符
(Z .104) ！值标识符
(Z .104) ！变量标识符

B．42 运算符

(42) ＜运算符＞::=
(Z .104) 运算符标识符
(Z .104)！＜通用运算符＞类型 标识符

B．43 同义定义

(43) ＜同义定义＞::=
(Z .104) 同义 名字＜常量表达式＞

B．44 常量表达式

(44) ＜常量表达式＞::=
(Z .104) ＜常量值＞
(Z .104)！＜运算符＞〔＜常量表达式＞〕+

B．45 常量值

(45) ＜常量值＞::=
(Z .104) 值 标识符
(Z .104)！同义 标识符

附件C1

（属于建议Z .100－Z .104）

SDL /GR概要

在SDL /GR 中，一个系统包括：
— 功能块交互作用图（BID），
— 信道子结构图，
— 功能块树，
— 进程树，
— 进程图，
— 状态概览图。

这些文件中的某一些对于提供系统规格（或描述）是基本的，而另外一些是辅助文件，辅助文件有助于理解该规格（或描述）。

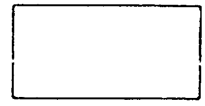
C 1.1 功能块交互作用图（BID）

C 1.1.1 符号

BID含有一个系统名字、一组功能块符号、环境符号、一组信道符号，並且可以含有进程符号。

C.1.1.1.1 框边线

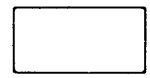
它围绕该图，代表所划出的诸功能块的边界线。



CCITT-74040

C1.1.1.2 功能块符号

它含有该功能块名字（见建议Z.101，§3.1.1和用户指南§D.4.3.2）。



CCITT-74040

C1.1.1.3 进程符号

它含有该进程名字（见建议Z.101，§3.1.1和用户指南§D.4.3.4），且可有一对括号，括号中含有形式参数表。

CCITT-74040

C1.1.1.4 环境符号

它仅用一个关键字来表示（见建议Z.101，§3.1.1）。

ENVIRONMENT

C1.1.1.5 信道符号

它含有一个信道名字。信道符号具有一个与一功能块符号相连接的始发端，和一个与另一功能块符号相连接的目的端。作为代替，或者是始发端或者是目的端（但不是它们两者）可不连接到一个功能块符号，而接到一个环境符号。

信道符号含有一个箭头，画在线条中间，用来表示信号流动的方向。

CCITT-74040

C1.1.1.6 信号路径符号

功能块中的信号路径符号可以附一个信号表符号。

信号路径符号可从一个进程引到另一个进程，或从一个进程引到一个信道的始发端（在功能块边界线上），或从一个信道的目的端（在功能块边界线上）引到一个进程。

信号路径符号可在一个信道的始发端（在功能块边界线上）汇聚，并可从一个信道的目的端（在功能块边界线上）发散。信号路径符号在一端有一个箭头，用来表示信号流动的方向。

CCITT-74040

C1.1.1.7 信号表符号

CCITT-74040

信号表符号含有一系列名字。信号表本身可以有一个名字，写在符号的上方；表中的项目是各个信号的名字，或者是其它信号表的名字，用逗号分隔，并且排成列或行。在表内，每个信号表名字还要再用一对方括号括起来，用这样的办法，将信号表名字与其它信号名字加以区分。

C1.1.1.8 创建符号

CCITT-74040

它用在进程符号之间，来表示在箭头端的进程实例被虚线始端的进程实例所创建。创建符号的两端可连接到同一个进程，以表示该进程的一个实例系由同一进程的另一个实例所创建。

C1.1.2 规则

每个功能块有一定数目的有向线将它连接到其它功能块和（或）环境。

如在功能块交互作用图中出现进程，则进程之间的信号线必须只处在同一功能块中彼此通信的进程之间。

C1.1.3 惯例

信道符号和信号线最好应以90度角分别连接到功能块符号边界线和进程符号边界线。如有必要，信道符号可含有多个90度的拐弯。

线条相交没有意义。

C1.2 信道子结构图

由于其组成部分是一些功能块和信道，该图与功能块子结构图类似。

C1.2.1 符号

在这种图中所用的符号与用于功能块子结构图中的符号相同。

C1.2.2 规则

图的连接规则与功能块子结构图相同。

C1.2.3 惯例

所介绍的FBID（功能块交互作用图）作图惯例也适用于信道子结构图。

C1.3 功能块树

功能块树含有框和“划分”线。

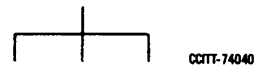
C1.3.1 符号

C1.3.1.1 框

框代表一个系统或一个功能块。被代表对象的名字应出现在符号里面。

C1.3.1.2 “划分”线

它们代表这样的关系：上面的功能块被划分成为下面的多个子功能块。



C1.3.2 规则

把这些符号连接起来以构成一棵层次结构的树。图的连接规则是：

— 有且只有一个根框（顶框）。
— 任何其它框必须连在“划分”线的一条支线后面。
— 任何一个框都可以跟有一“划分”线。
— 一“划分”线必须连在一个框后面，且在其所有支线上都必须跟有框。

C1.3.3 惯例

树应画成这样：一个划分层在这种表示中作为一致层出现，即图中的“划分”线向下的长度应相等。

C1.4 进程树

进程树含有一些进程符号和“划分”线。

C1.4.1 符号

C1.4.1.1 进程符号

如在建议Z .101（§3.1.2)中所定义的，所指的进程的名字应出现在符号里面。用注释语法来表示该进程被配置在哪个功能块里。

C1.4.1.2 “划分”线

它们代表这样的关系：上面的进程被划分成下面的一些子进程，且可以由这些子进程来代替上面的进程。

C1.4.2 规则

把这些符号连接起来以形成一棵层次结构的树。图的连接规则是：

— 有且只有一个根进程符号（顶进程），它跟有一“划分”线。

— “划分”线在其每一条支线上都跟有一个进程符号。

C1.4.3 惯例

如果一个进程树图较大，可适当地把该图分成几个图。这种分法应该是这样：把原先的图截断，使一组要再划分的进程看来象是没有被划分，在下面的图中，这些进程就作为根。

C1.5 进程图

进程图是一组由流线连接起来的符号。

C1.5.1 符号

C1.5.1.1 起动符号

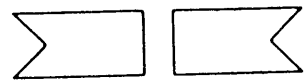
它含有它所描述的进程的进程名字（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。

C1.5.1.2 状态符号

它含有状态名字或一个星号*，或星号后再跟括在方括号内的诸状态名字（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。

C1.5.1.3 输入符号

它含有用逗号分隔的信号名字或一个星号*（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。

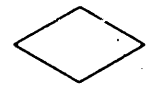


C1.5.1.4 保存符号

它含有用逗号分隔的信号名字或一个星号*（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。

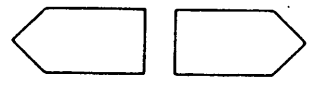
C1.5.1.5 判定符号

它含有判定名(任选的)和一个形式的或非形式的正文短语(见建议Z.101，§3.3.2及用户指南§§D.4.3和D.6.3)。



C1.5.1.6 输出符号

它含有用逗号分开的信号名字（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。



C1.5.1.7 任务符号

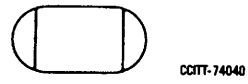
它含有任务名字（任选的）和一个形式的或非形式的正文短语（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。

C1.5.1.8 过程调用符号

它含有过程名字和括在圆括号内的实在参数（见建议Z.103，§2.2及用户指南§§D.4.3和D.6.3）。

C1.5.1.9 过程启动符号

它含有过程名字和括在圆括号内的形式参数（见建议Z.103，§2.2及用户指南§§D.4.3和D.6.3)。

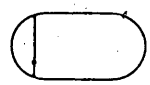


C1.5.1.10 返回符号

它是一个圆圈，里面有一个乘号叉（见建议Z.103，§2.2及用户指南§§D.4.3和D.6.3）。

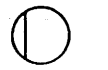
C1.5.1.11　宏入口处符号

它含有宏名字（见建议Z.103，§4.2及用户指南§§D.4.3和D.6.3）。



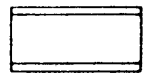
C1.5.1.12　宏出口处符号

它是一个圆圈，里面有一竖线（见建议Z.103，§4.2及用户指南§§D.4.3和D.6.3）。



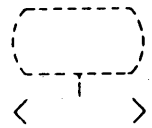
C1.5.1.13　创建符号

它含有进程名字和括在括号内的实在参数（见建议Z.101，§3.3.1及用户指南§§D.4.3和D.6.3）。



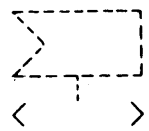
C1.5.1.14　连续信号符号

它含有条件正文，然后是关键字PRIORITY，后跟有关的优先数（见建议Z.103，§3.3.3及用户指南§§D.4）。



C1.5.1.15　允许条件符号

它含有条件正文（见用户指南§§D.4）。



C1.5.1.16　替换符号

它含有替换正文（见建议Z.103，§5.2及用户指南§D.4）。

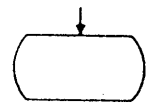
C1.5.1.17　连接符号

它含有连接名字（见建议Z.101，§3.3.1及用户指南§D.4）。

CCITT-74040

C1.5.1.18　下一状态符号

它含有状态名字或一条短横线（见建议Z.101，§3.3.1及用户指南§D.4）。



CCITT-74040

C1.5.1.19　停止符号

它是一个乘号叉（见建议Z.101，§3.3.1及用户指南§D.4）。

CCITT-74040

C1.5.2　规则

— 一个状态符号必须和一个或多个输入符号或保存符号相连接（和其它符号的任何连接都是错误的）。
— 过程调用节点不可跟在状态节点、停止节点或返回节点后面，而可跟在任一其它节点后面。
— 调用节点后面不可跟有起动节点、过程启动节点、输入节点、返回节点或保存节点，而可跟有这些节点以外的任一节点。
— 允许条件可附于任何输入符号。
— 连续信号附于一条来自状态符号但无输入符号的流线。
— 宏调用符号可以插入到任何图中（进程图、功能块交互作用图、状态一览图），且可插入到图的任何地方。它可以有一个或多个入口处，在有多个入口处的情况下，每个入口处应附有标号。它能有零个、一个或多个出口处，在最后一种情况下每个出口处应附有标号。入口处用指向宏符号的箭头表示，出口处用从宏符号发出的箭头表示。宏符号的入口处和出口处应通过适当类型的流线(根据它们的意义)连接到其它符号。
— 过程和宏的停止符号或返回符号后面不能跟任何符号。
— 凡是可以安排任务的地方，都可以放创建符号。
— 只有在各替换行为不包括状态且终结于某点时，替换符号才能直接插入到一个跃迁之中。
— 一条实流线可用一对相配的连接符断开，其流向设定为从出连接符到与其相配的入连接符。
— 在两个或更多个符号后面只跟有一个符号的场合，通向该符号的流线就汇聚，这种汇聚的表现形式可以是一条流线流入另一条流线，或一个以上的出连接符和一个入连接符相联系，或几条分开的流线进入同一个符号。

例：汇聚

CCITT-74040

— 出连接符后面不跟任何别的符号，且无流线从它那里发出。
— 在一个符号后面跟有两个或更多个其它符号的场合，一条从该符号引出的流线可发散成两条或两条以上的流线。

例：发散

CCITT-74040

— 每当两条流线汇聚、以及每当一条流线进入一出连接符或状态符号时，都要求有箭头。进入输入符号的流线上禁止有箭头。
— 连接符号要连接到一个符号或连接到一条实流线。

C1.5.3 惯例

— 在任何一个图中，同一类型的所有符号最好都大小一样。
— 符号的取向最好是水平的，符号的宽高比最好为 2：1。

C1.5.4 通用符号

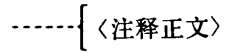
C1.5.4.1 正文扩展符号

它能附于所有的SDL/GR符号。在这个符号中所包含的正文应被看成是包含在正文扩展符号所依附的那个符号之中。

CCITT-74040

C1.5.4.2 注释符号

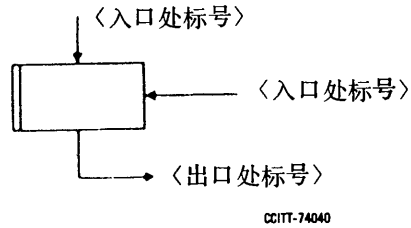
注释符号含有注释正文。



CCITT-74040

C1.5.4.3 宏、入口处和出口处符号

宏符号是对宏定义的引用。宏的入口处和宏的出口处分别由指向宏和发自宏的流线表示，流线可以任选地附上标号。宏符号含有宏定义的名字。



CCITT-74040

C1.6 状态概览图

这种图表示进程中的状态的序列，能够看出从哪些状态可以达到某一个状态（见用户指南§§D.4.2和D.4.3）。

C1.6.1 符号

C1.6.1.1 状态符号

在这种图中，状态符号是一个圆圈，其中含有该状态的名字（见用户指南§§D.4.2和D.4.3）。

C1.6.2 规则

— 诸状态用弧线连接，可以把多个输入信号名字任选地写在弧线上面。

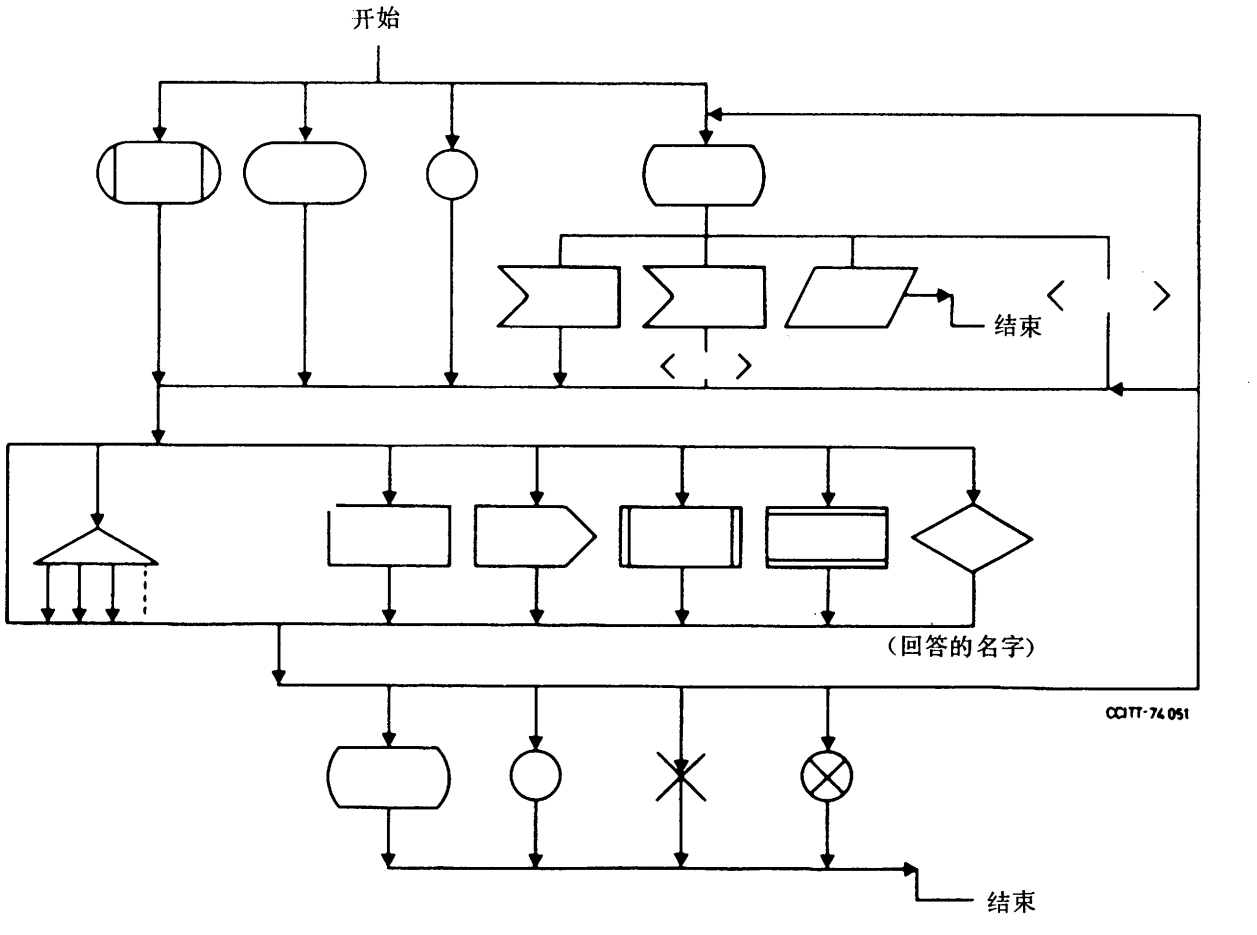
C1.7 通用惯例

在图形语法中有一些作图惯例，适用于所有类型的图：

— 符号的长宽比和大小可以不同，由用户随意处理；

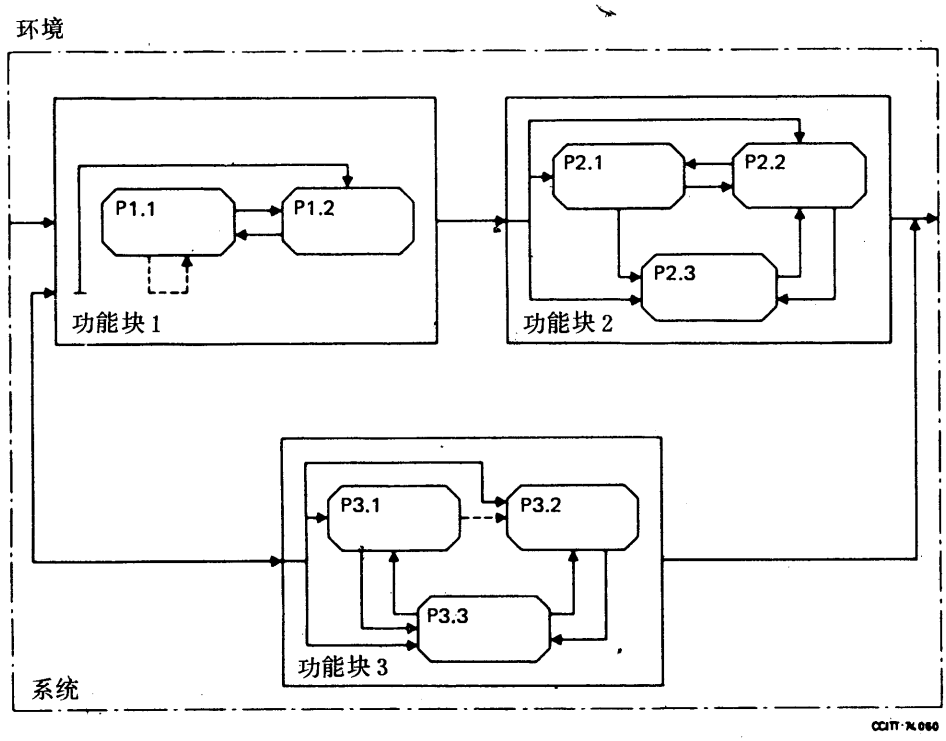
— 符号的边界线不得互相重叠或相交。这条规则不适用于信道符号和信号流符号，它们可以相交。相交的诸信道或信号流符号之间没有逻辑关系。

C1.8 在SDL/GR进程图中允许的用流线连接符号的方法

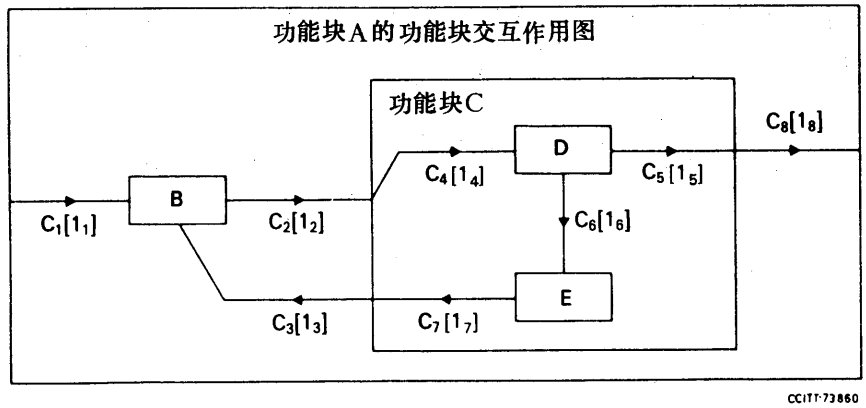


C1.9 举例

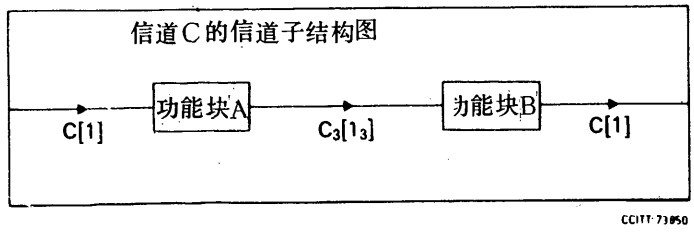
C1.9.1 功能块交互作用图



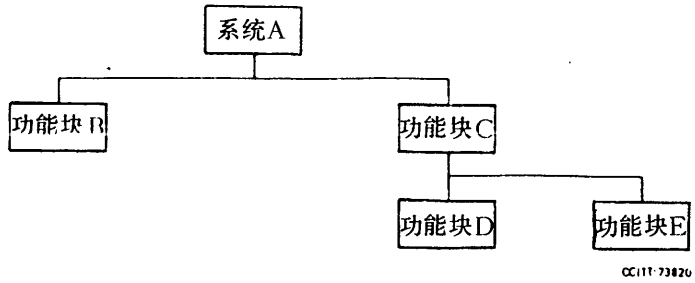
C1.9.2 功能块交互作用图



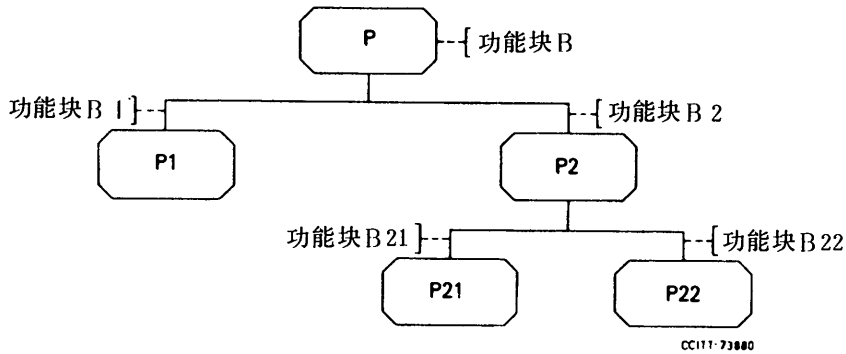
C1.9.3 信道子结构图



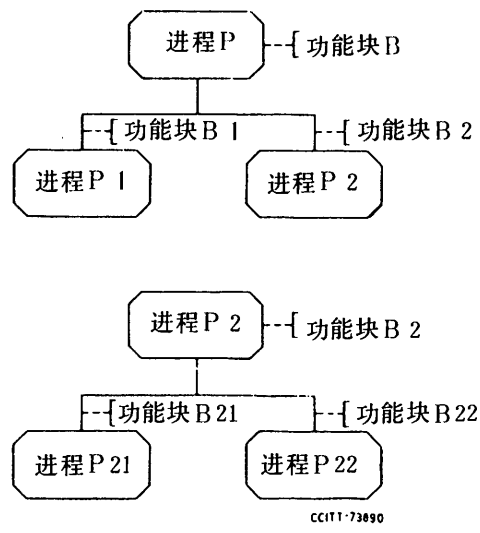
C1.9.4 功能块树



C1.9.5 进程树



同一进程树的另一种表示法（当进程树较大时此表示法较有用）。



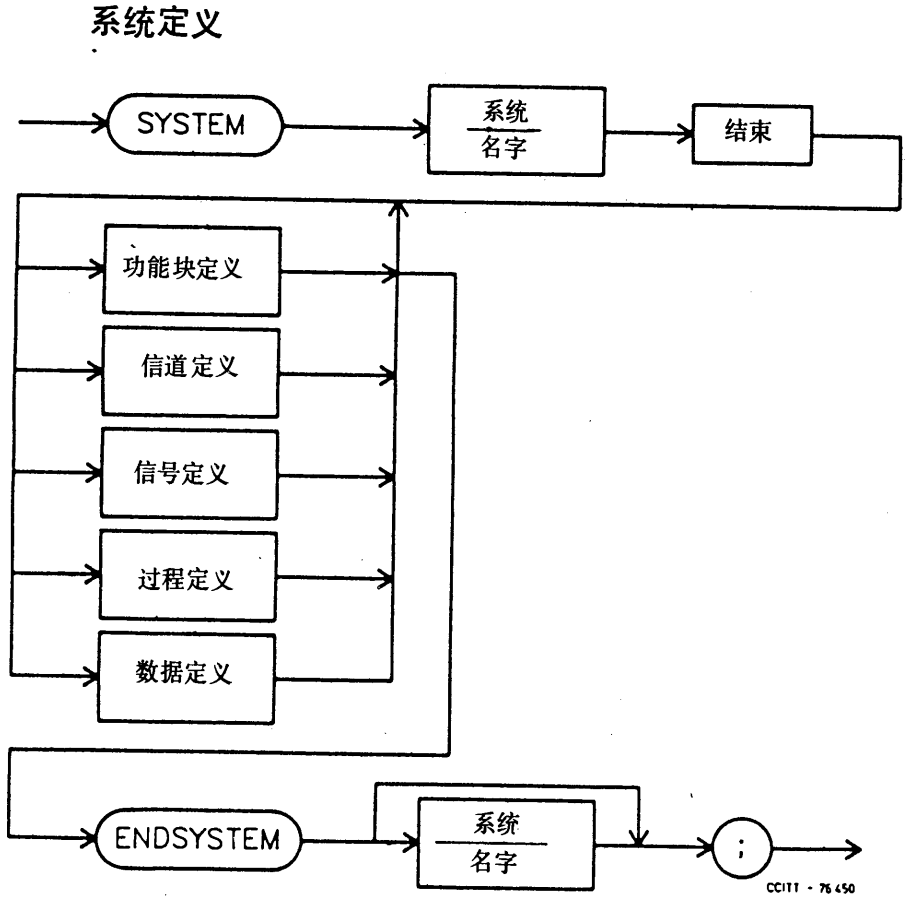
附件C 2

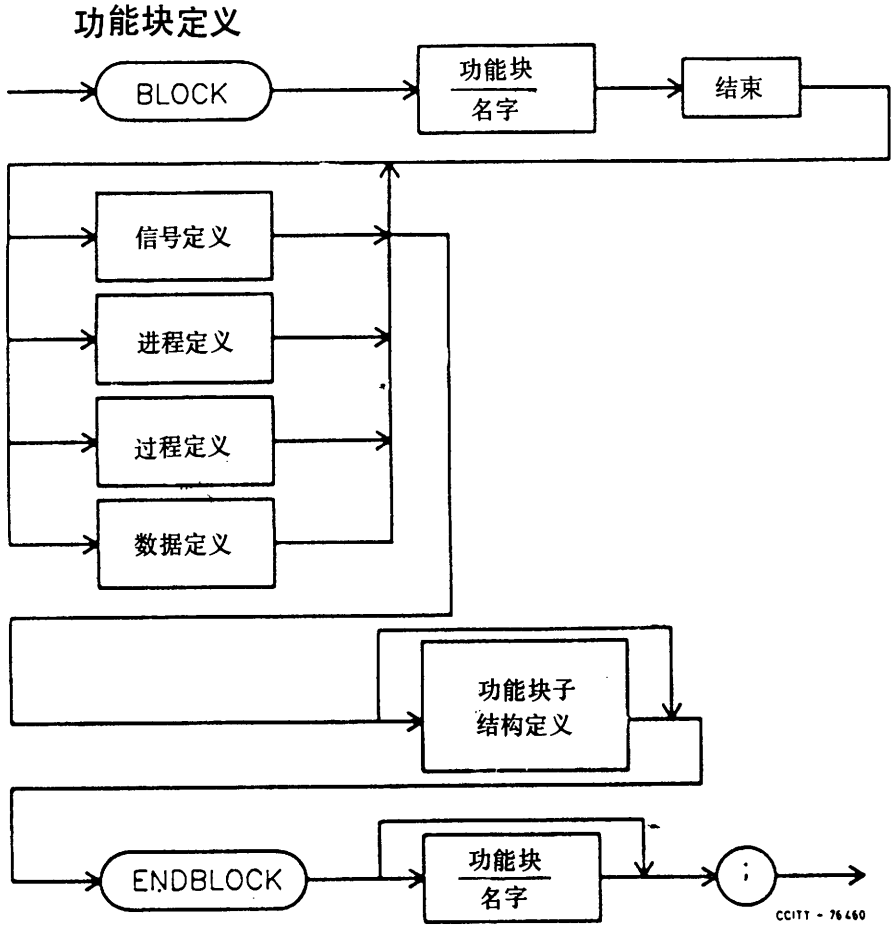
（属于建议Z .100至Z .104）

SDL/PR 概要

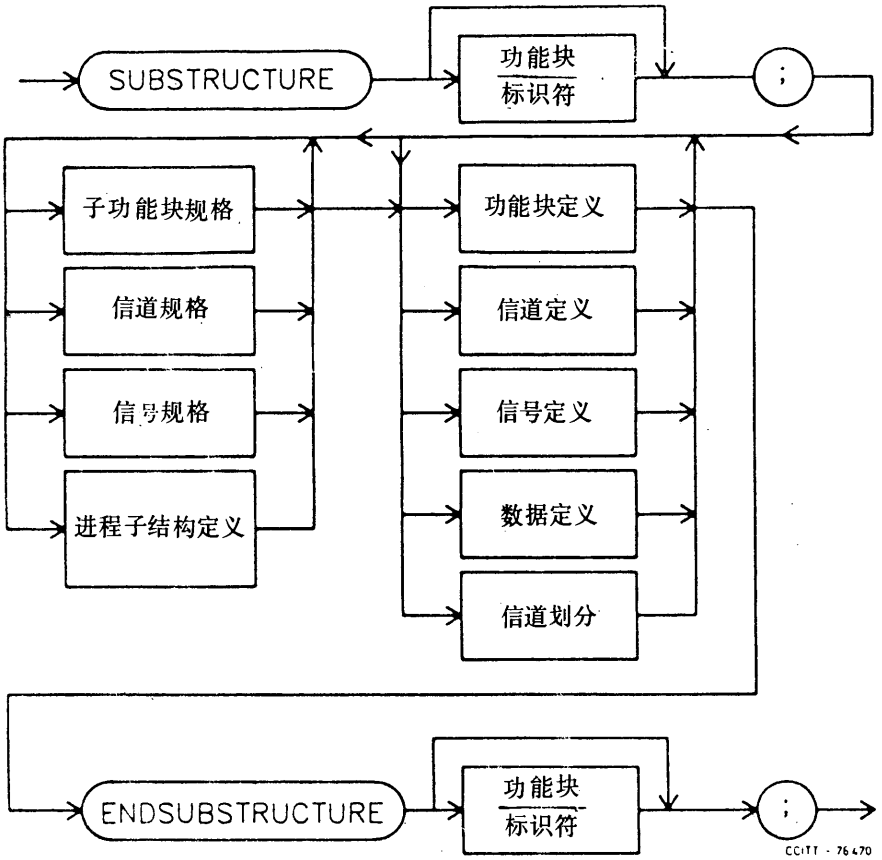
C2.1 语法

语法图

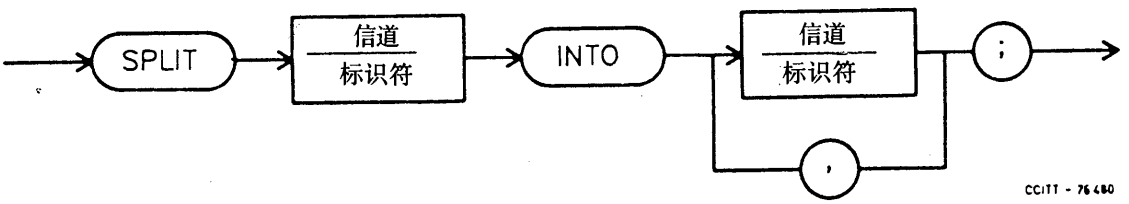



功能块定义
BLOCK
功能块
名字
结束
信号定义
进程定义
过程定义
数据定义
功能块子
结构定义
ENDBLOCK
功能块
名字
;
CCITT - 76460

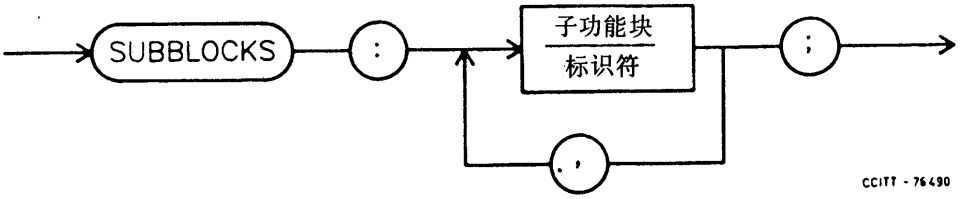
功能块子结构定义



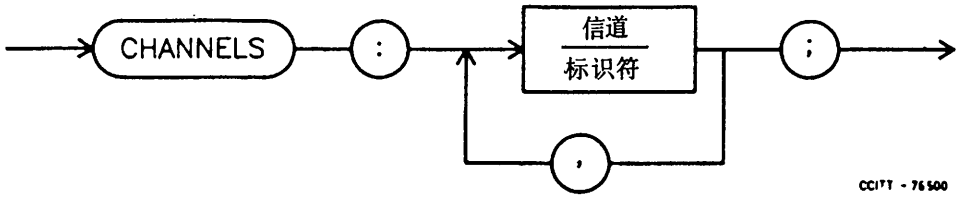
信道划分



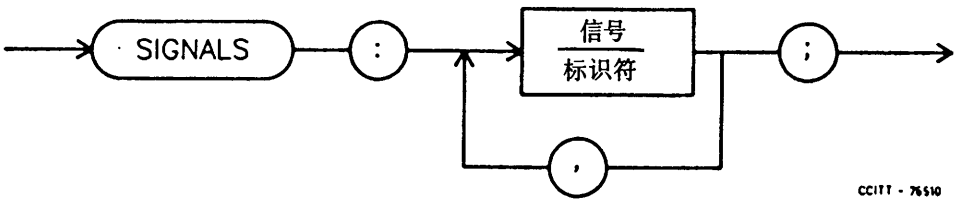
子功能块规格



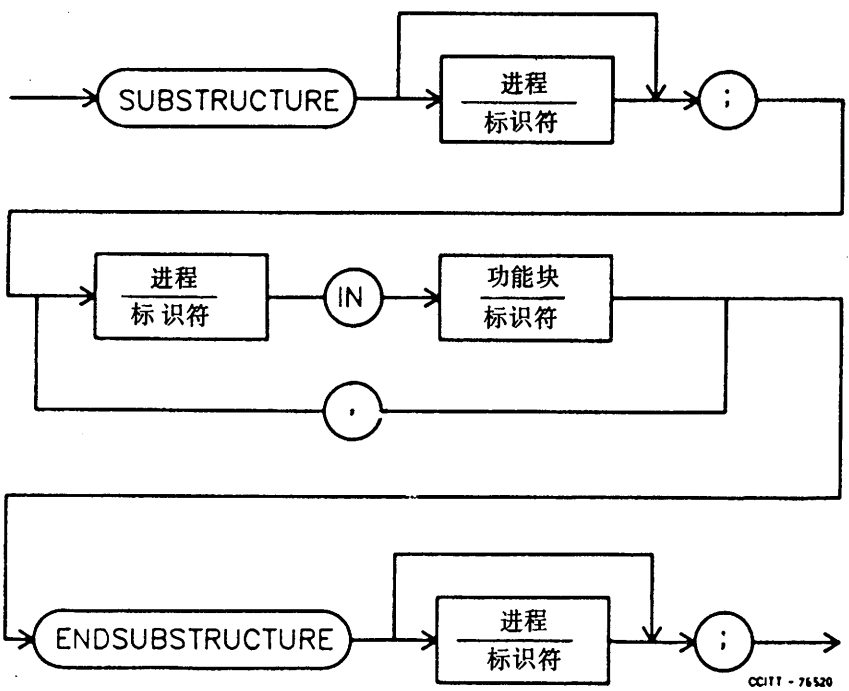
信道规格



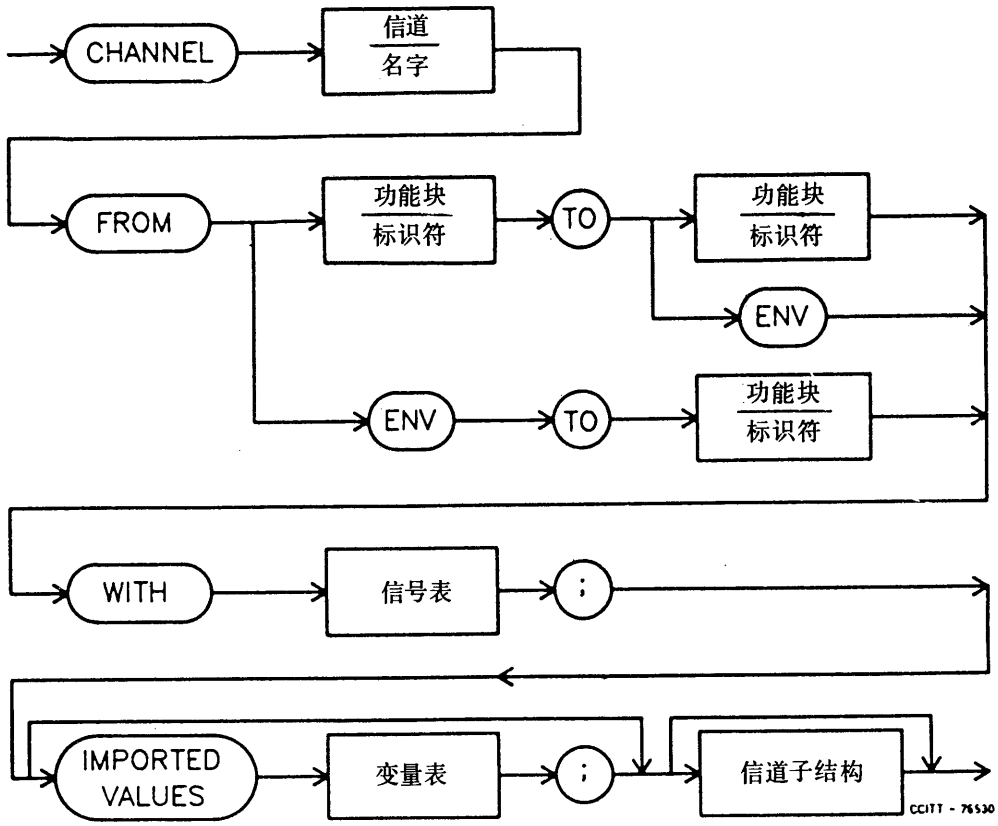
信号规格



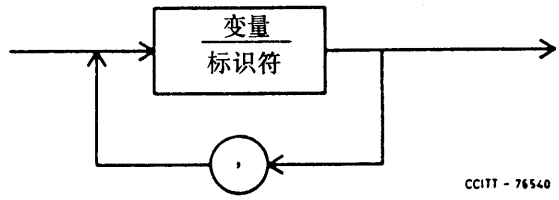
进程子结构定义



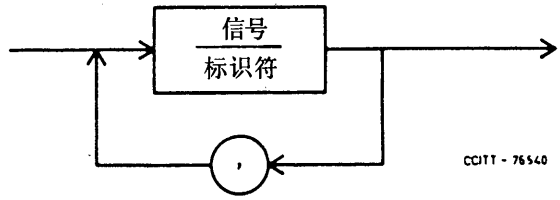
信道定义


CHANNEL
信道
名字
FROM
功能块
标识符
TO
功能块
标识符
ENV
ENV
TO
功能块
标识符
WITH
信号表
;
IMPORTED
VALUES
变量表
;
信道子结构
CCITT - 76530

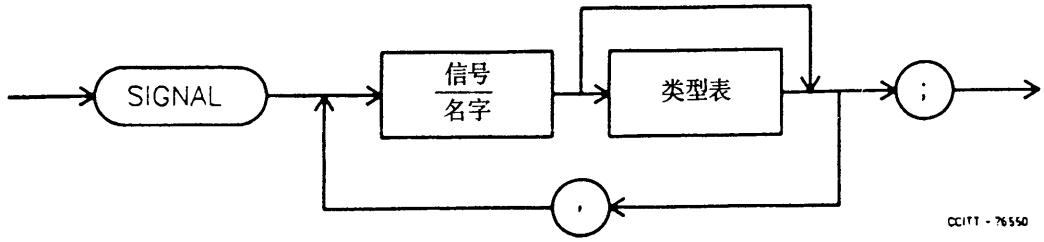
变量表


变量
标识符
,
CCITT - 76540

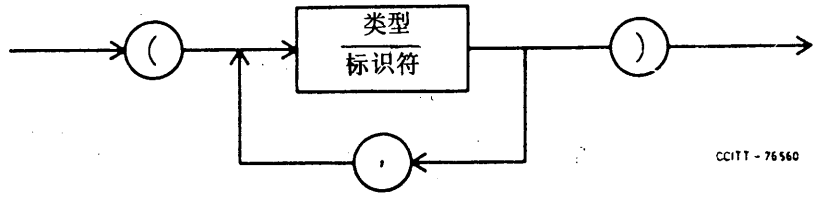
信号表


信号
标识符
,
CCITT - 76540

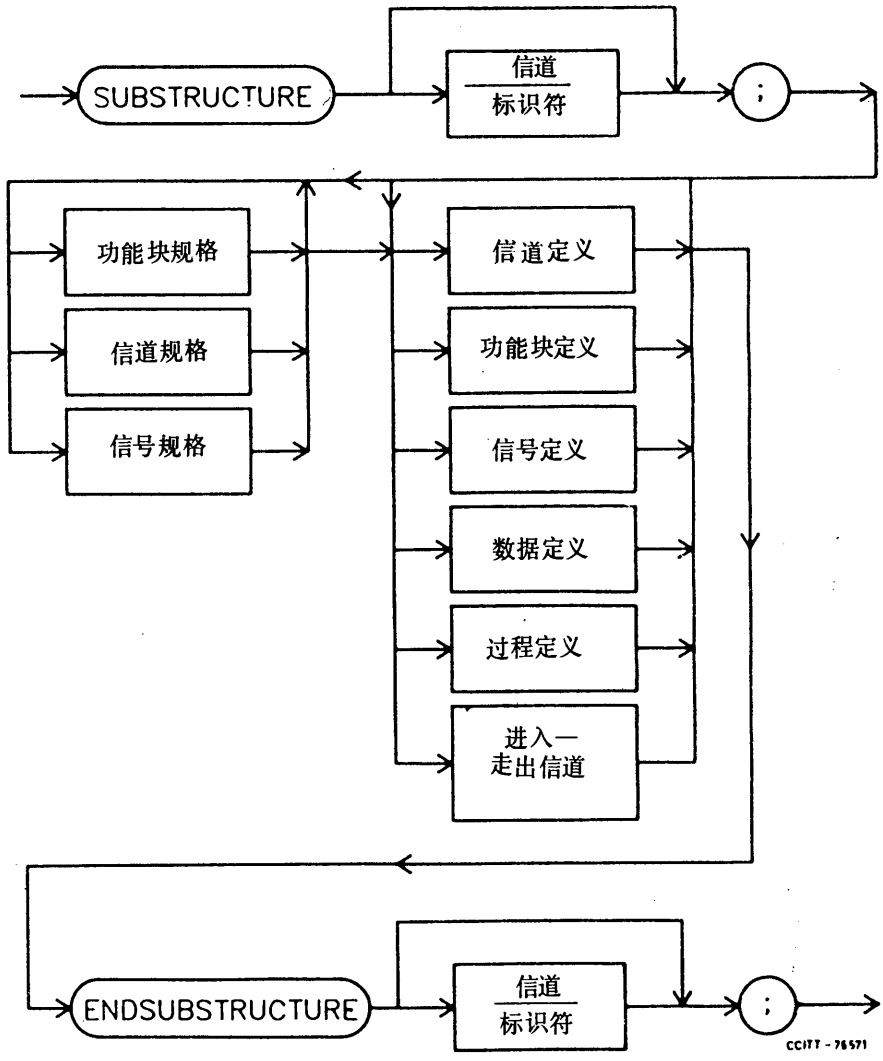
信号定义



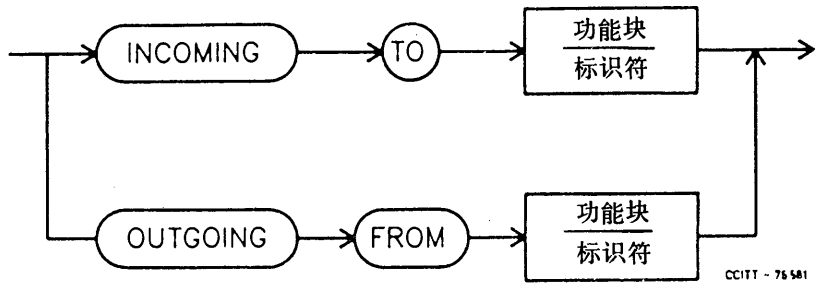
类型表



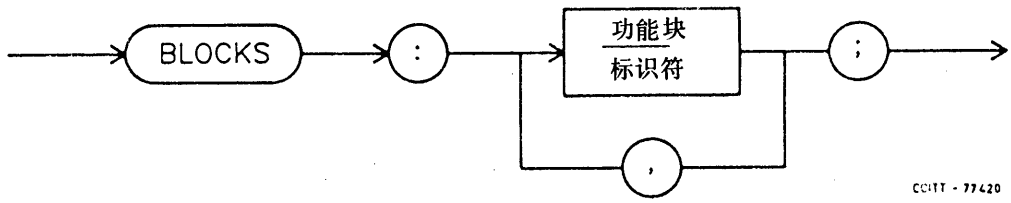
信道子结构定义



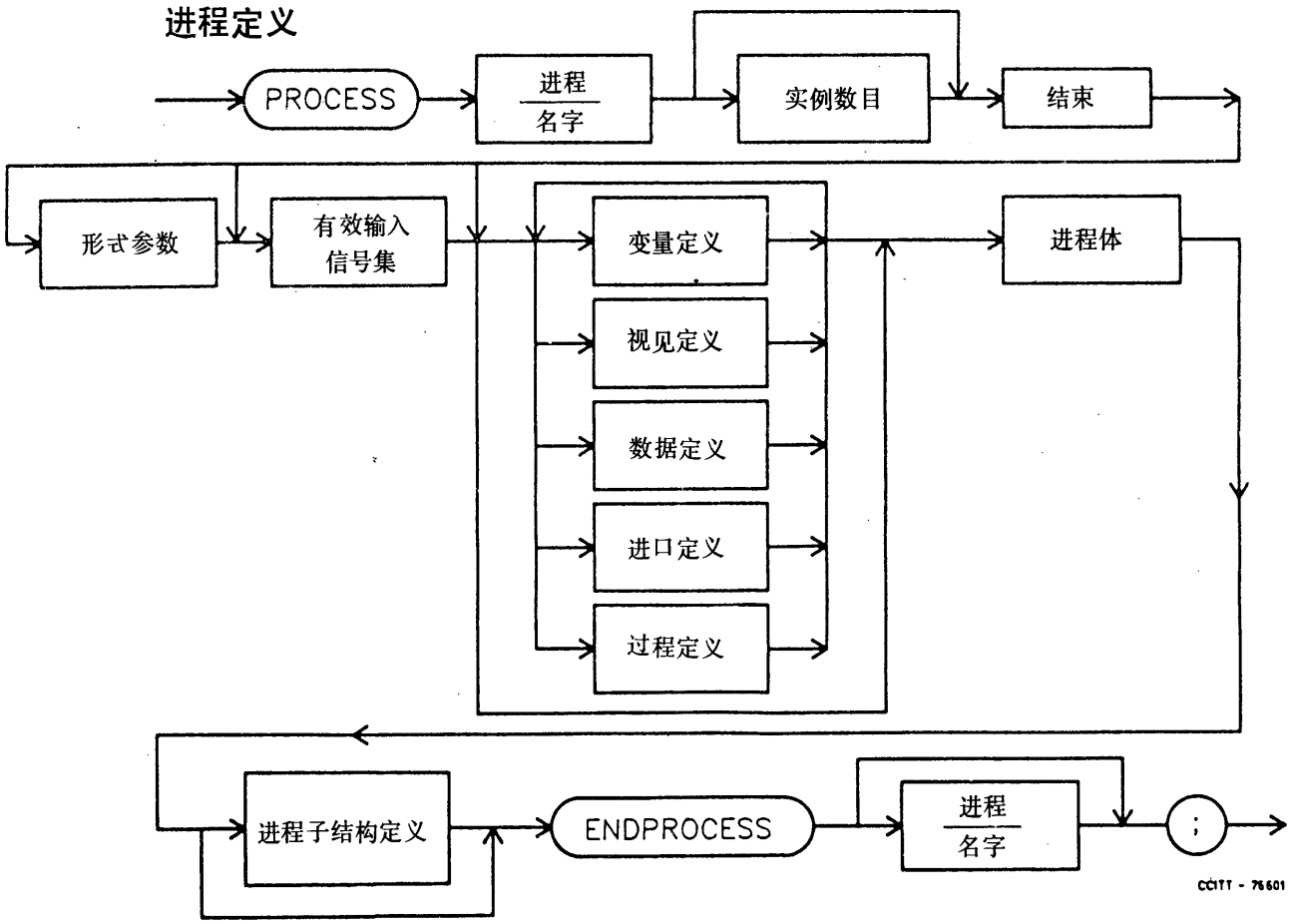
进入—走出信道



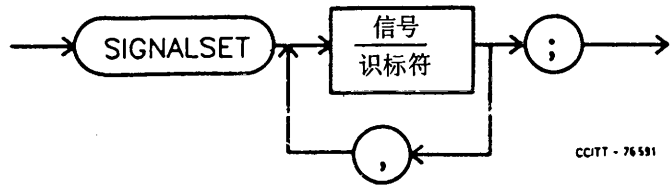
功能块规格



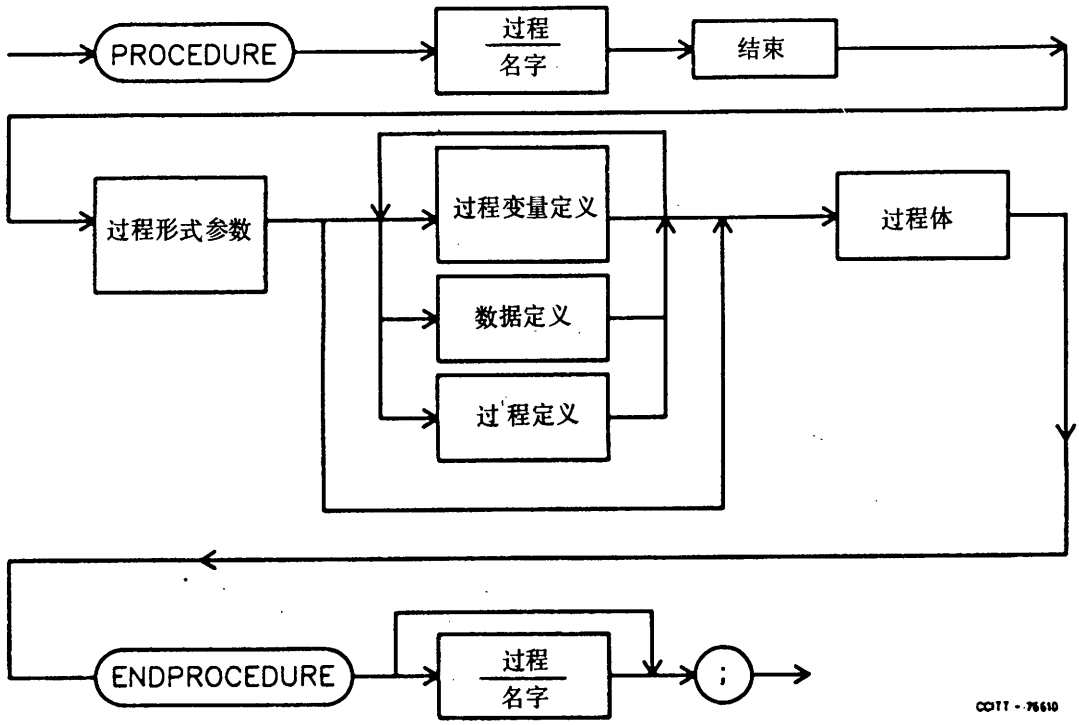
进程定义



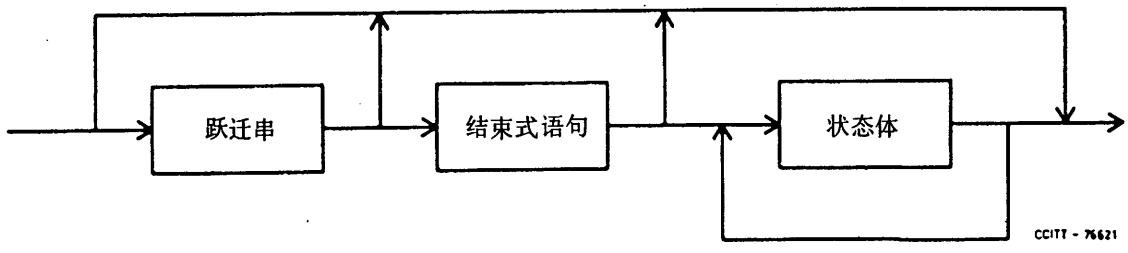
有效输入信号集



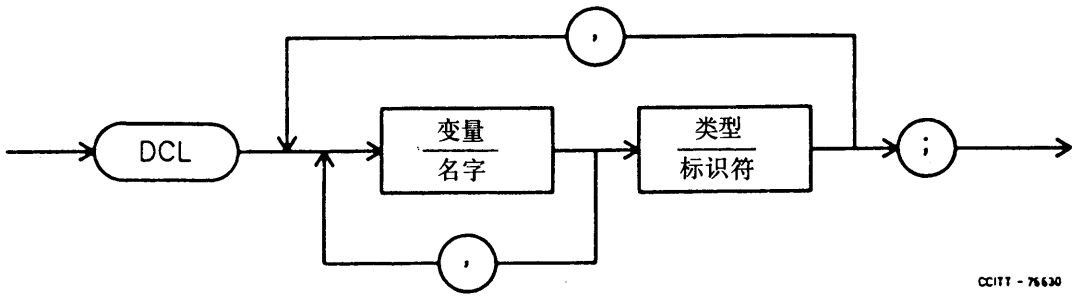
过程定义



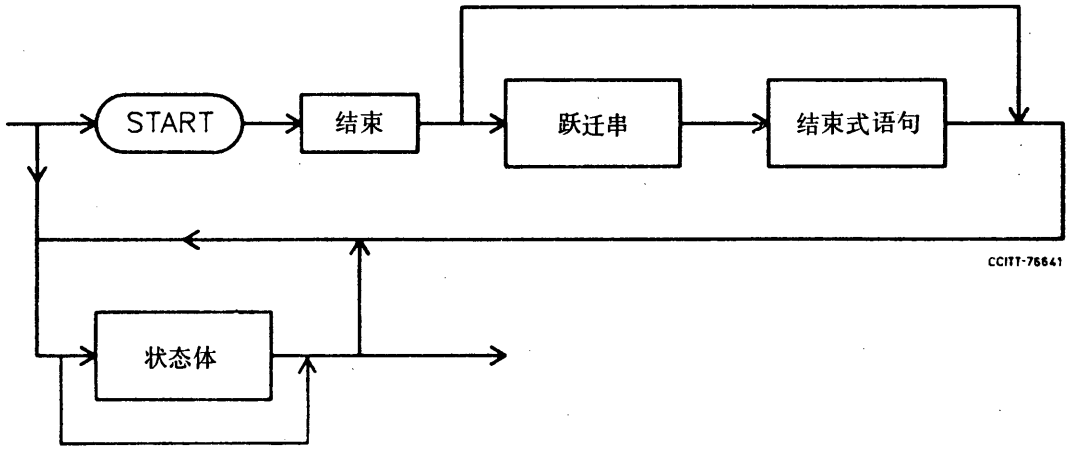
过程体



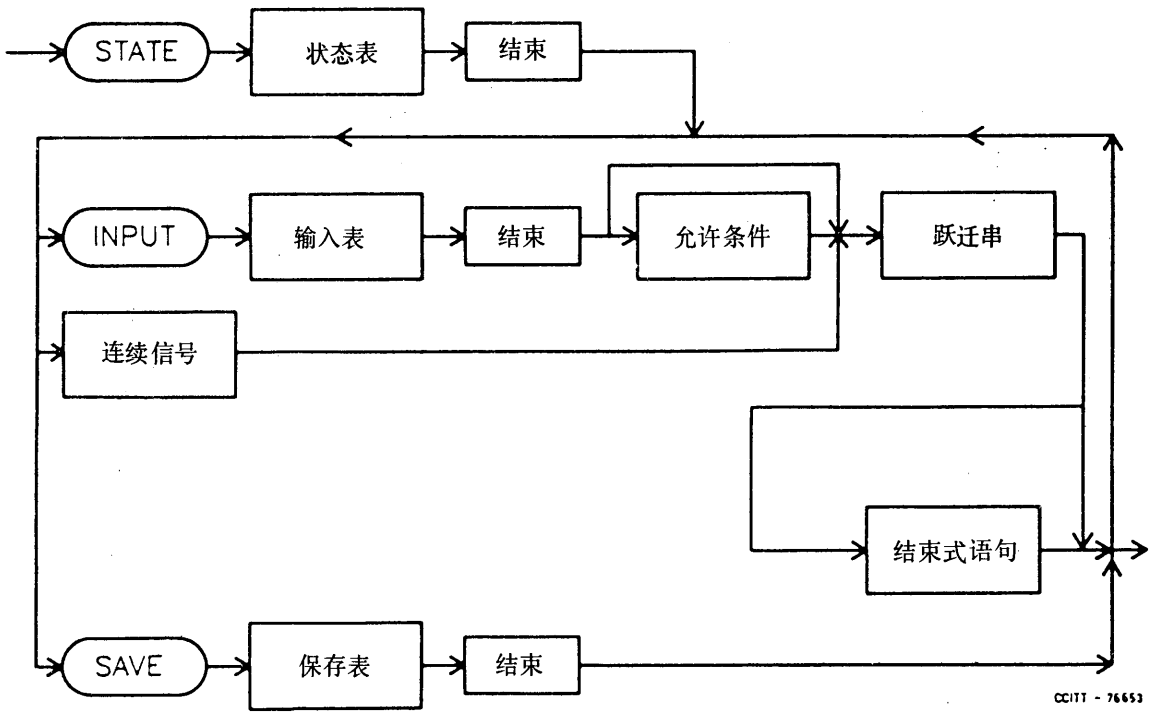
过程变量定义



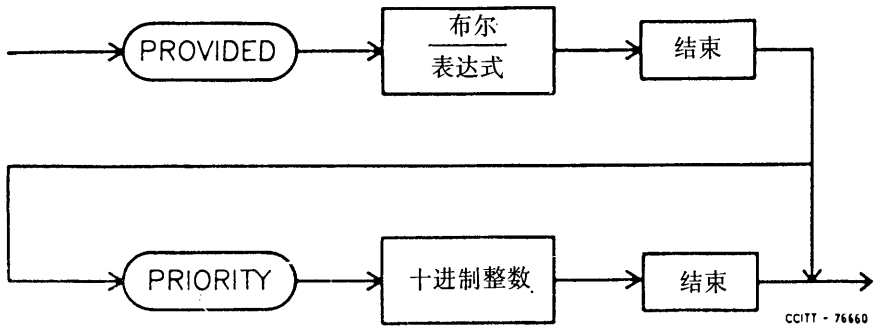
进程体



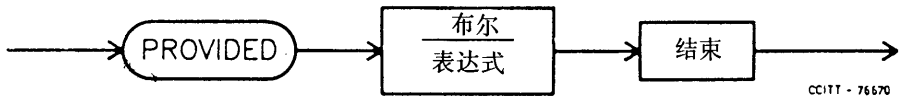
状态体



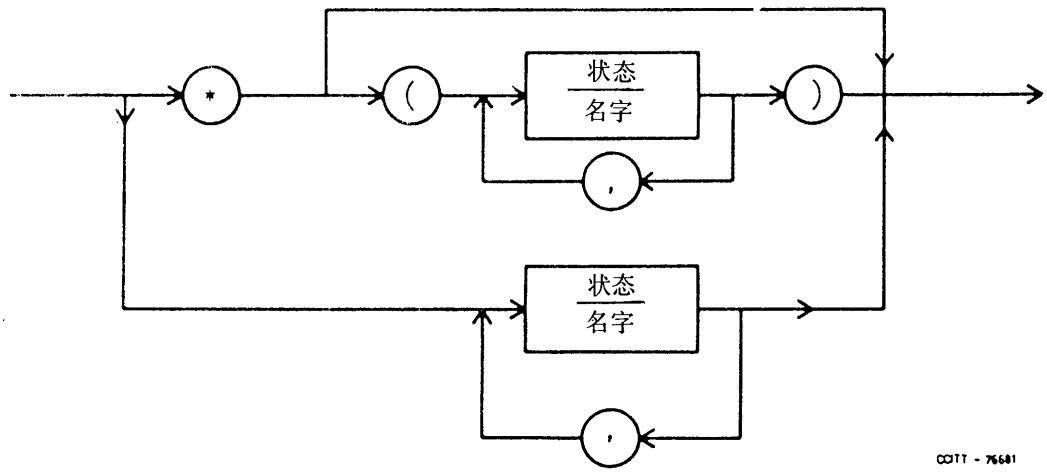
连续信号



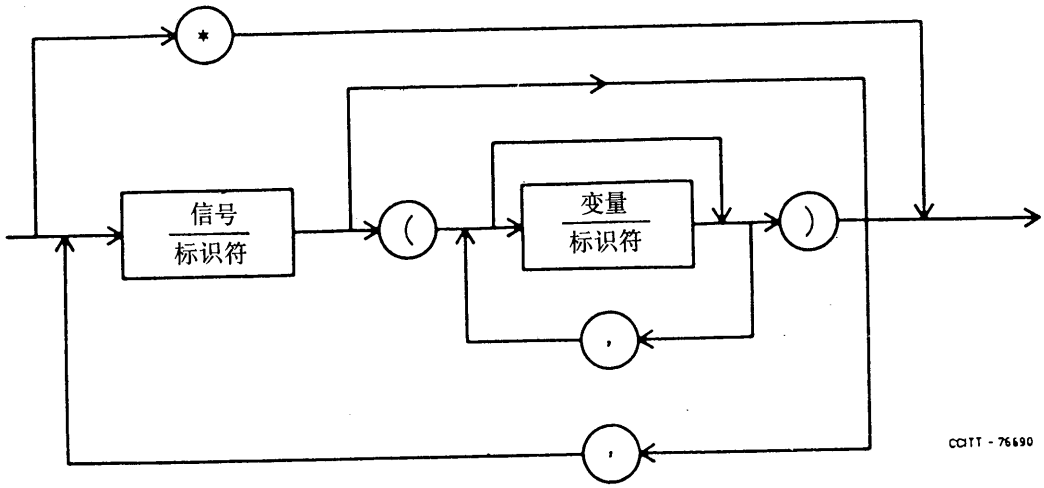
允许条件



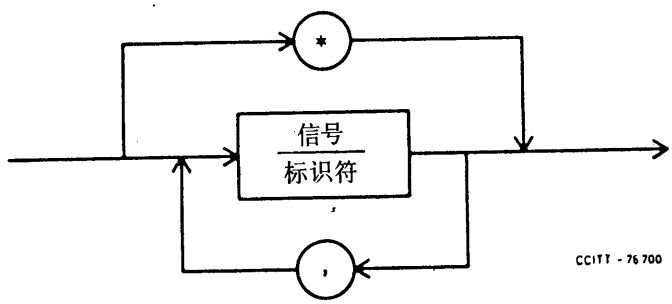
状态表



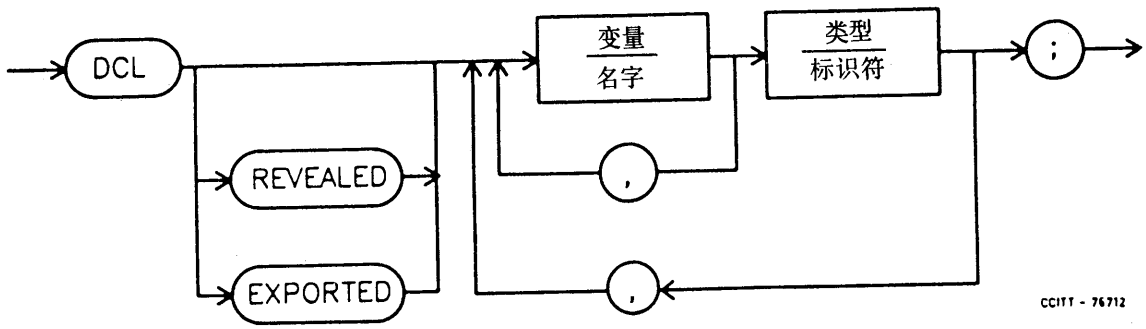
输入表


*
信号
标识符
(
变量
标识符
)
,
,
CCITT - 76690

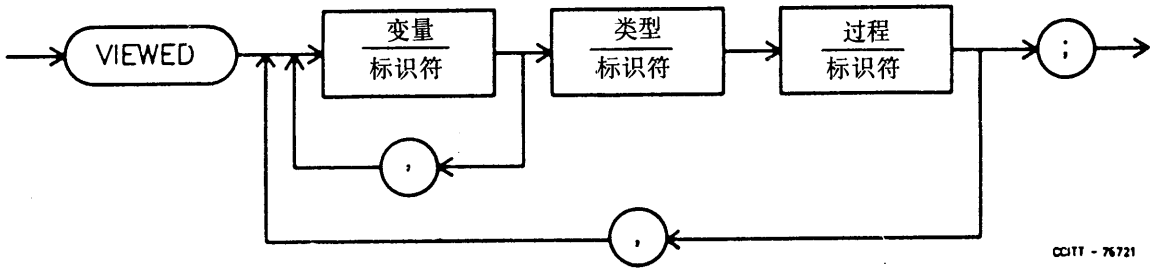
保存表


*
信号
标识符
,
CCITT - 76700

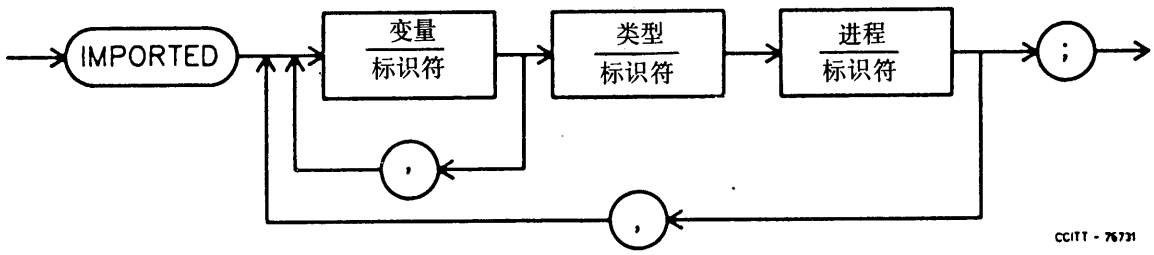
变量定义


DCL
REVEALED
EXPORTED
变量
名字
类型
标识符
;
,
,
CCITT - 76712

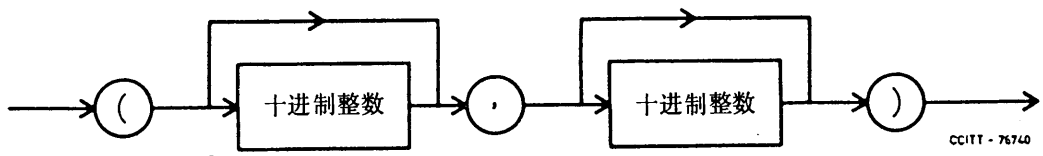
视见定义



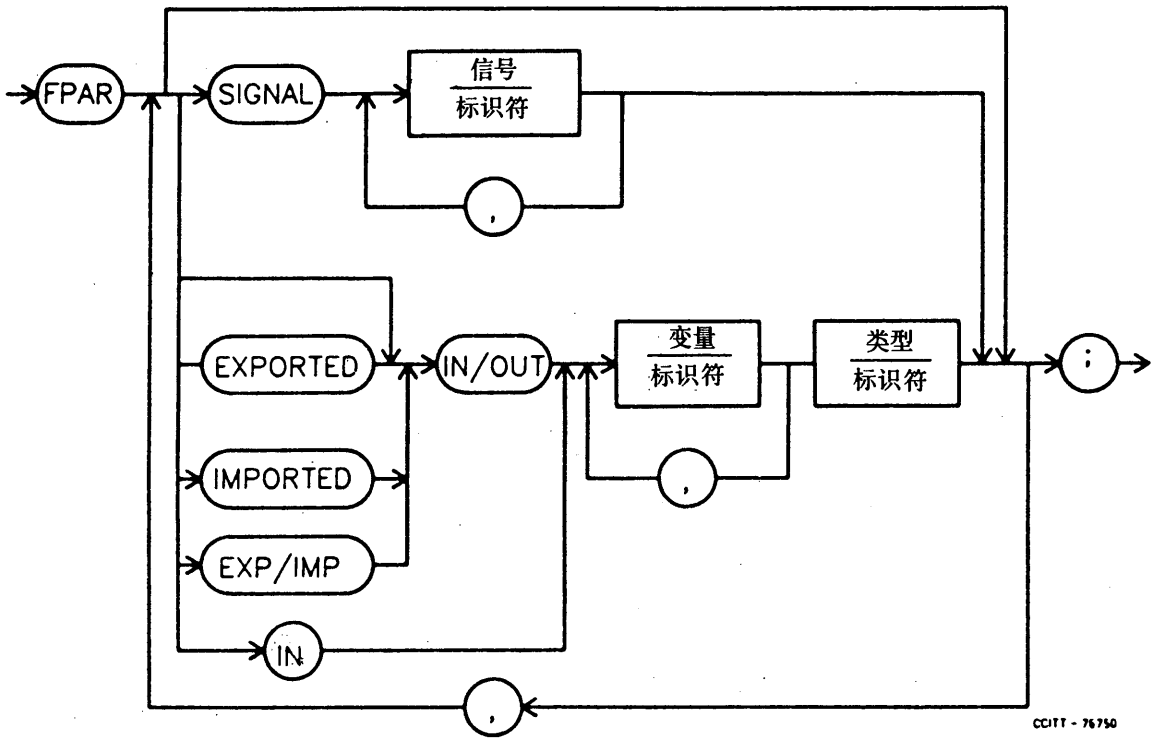
进口定义



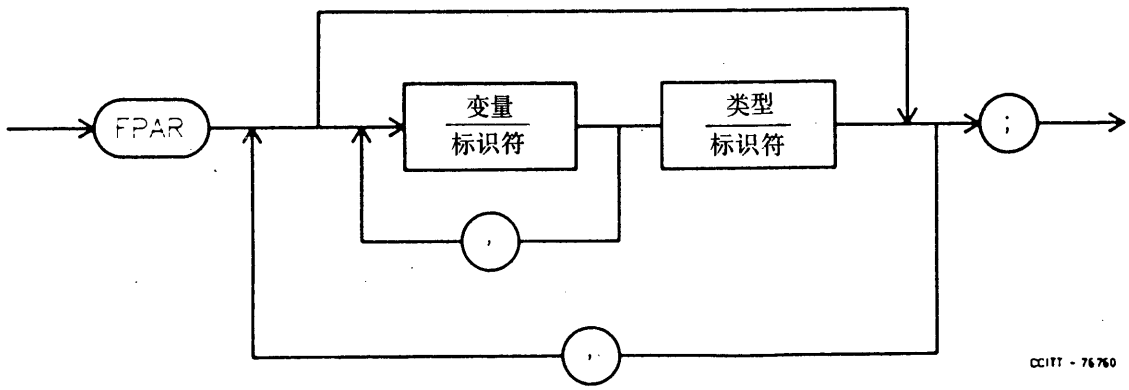
实例数目



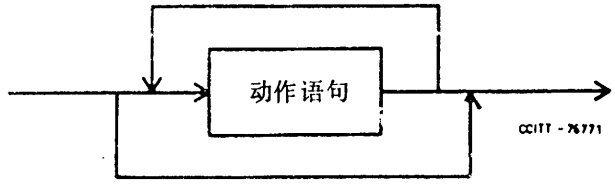
过程形式参数


FPAR
SIGNAL
信号
标识符
,
EXPORTED
IN/OUT
变量
标识符
类型
标识符
;
IMPORTED
EXP/IMP
IN
,
,
CCITT - 76750

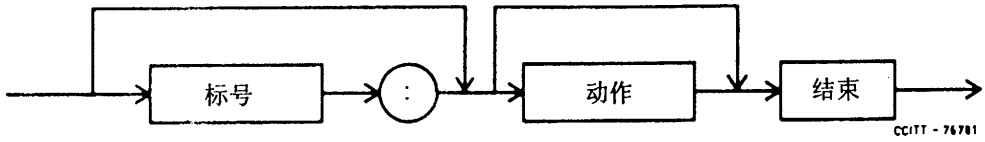
形式参数


FPAR
变量
标识符
类型
标识符
;
,
,
CCITT - 76760

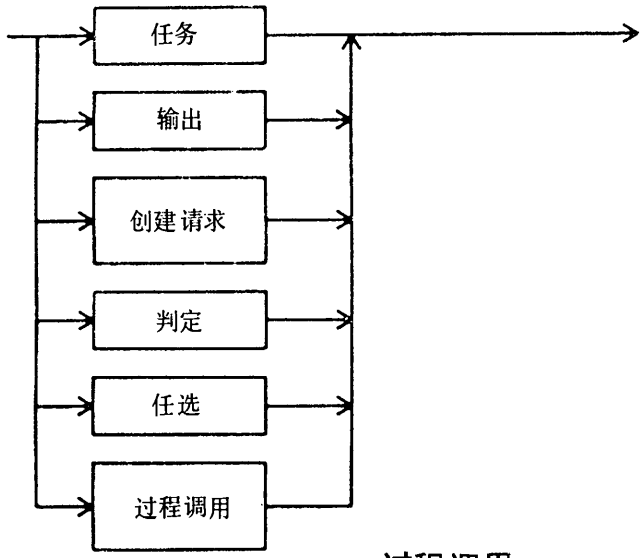
跃迁串



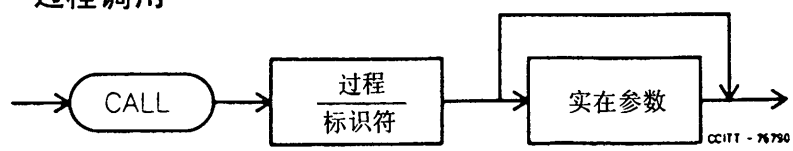
动作语句



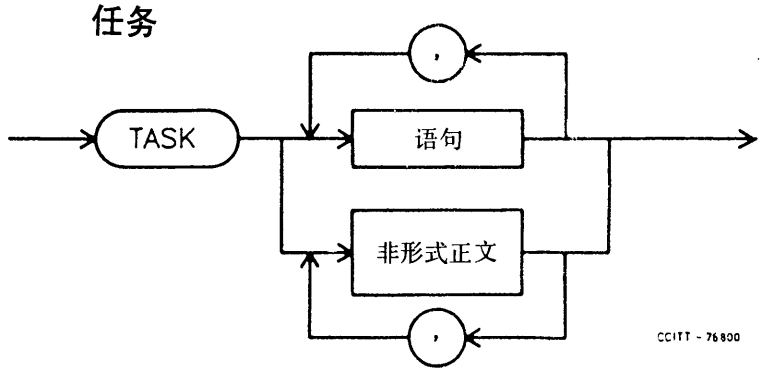
动作



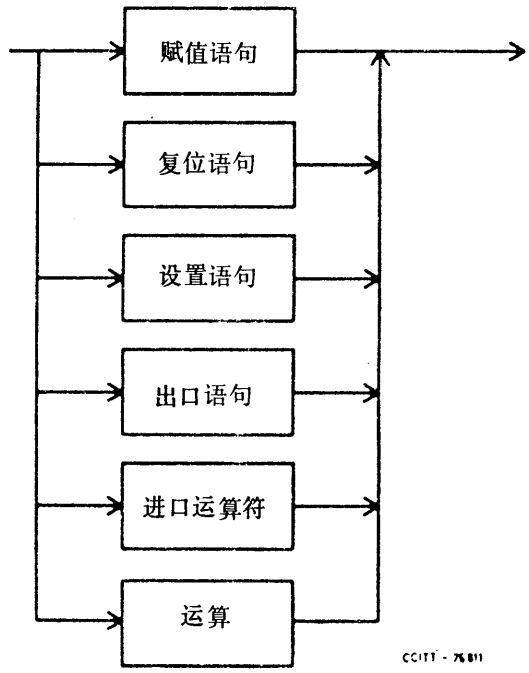
过程调用



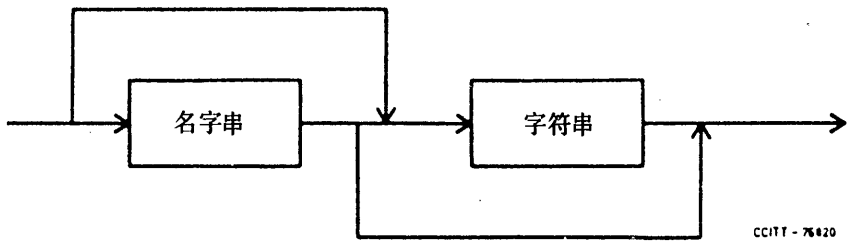
任务


TASK
语句
非形式正文
,
,
CCITT - 76800

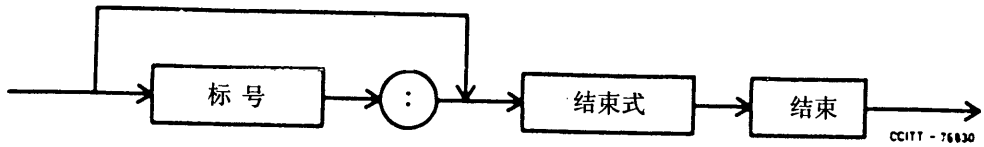
语句


赋值语句
复位语句
设置语句
出口语句
进口运算符
运算
CCITT - 76811

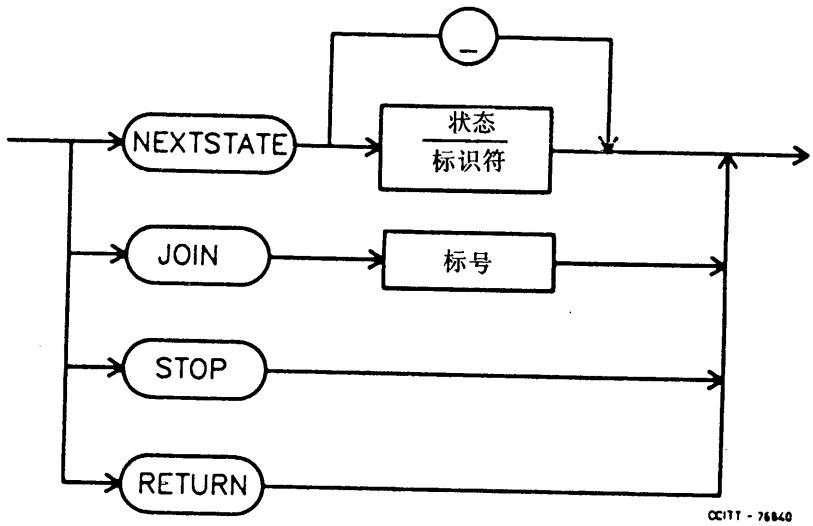
非形式正文


名字串
字符串
CCITT - 76820

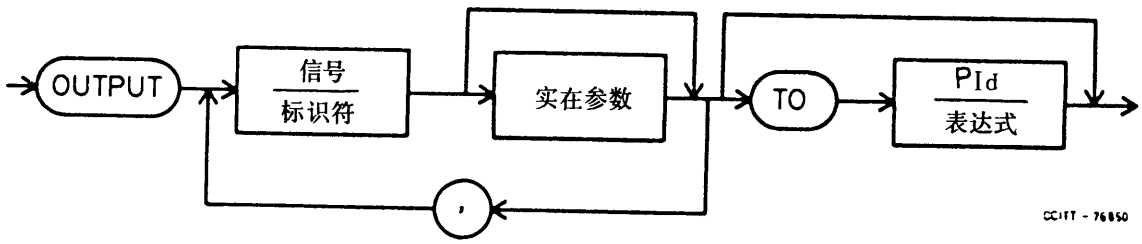
结束式语句



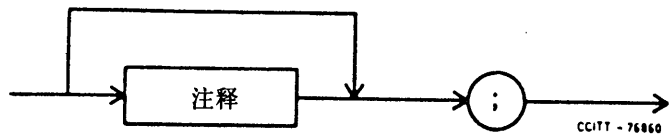
结束式



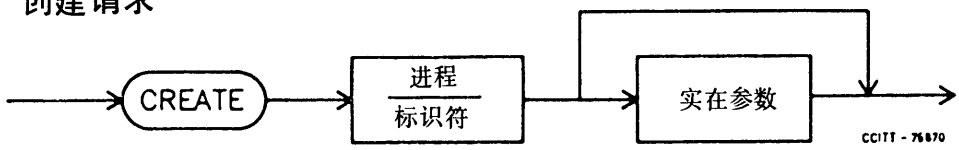
输出



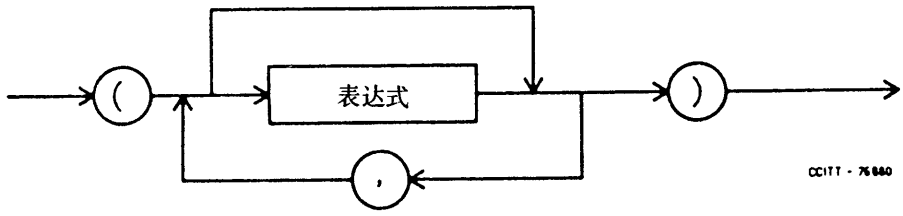
结束



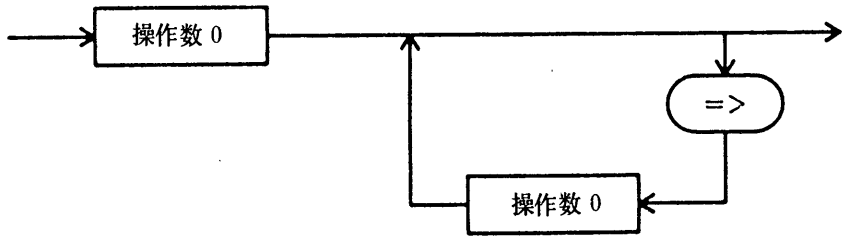
创建请求



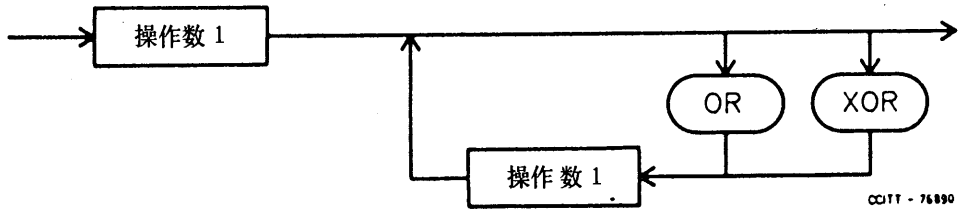
实在参数

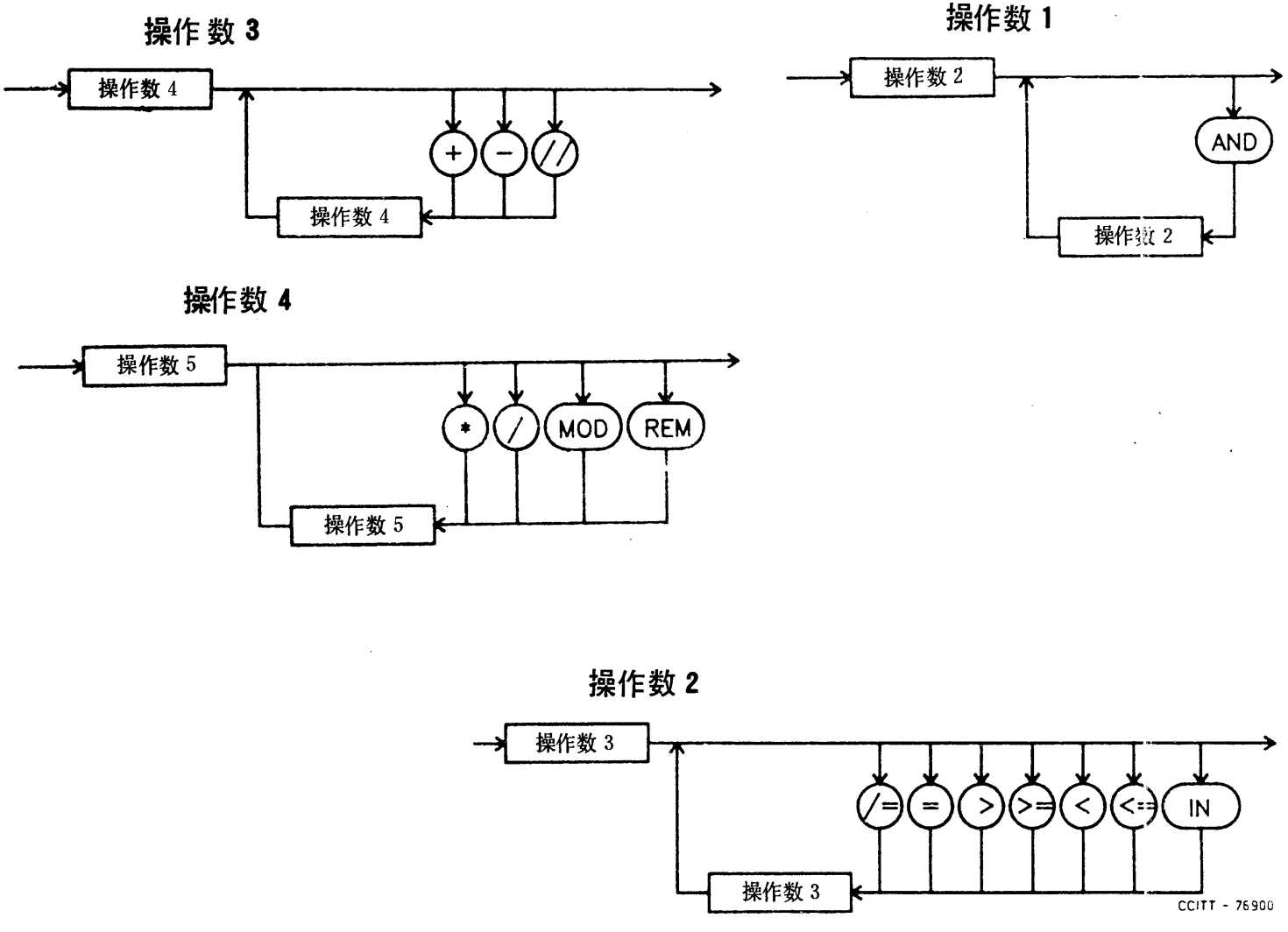


表达式

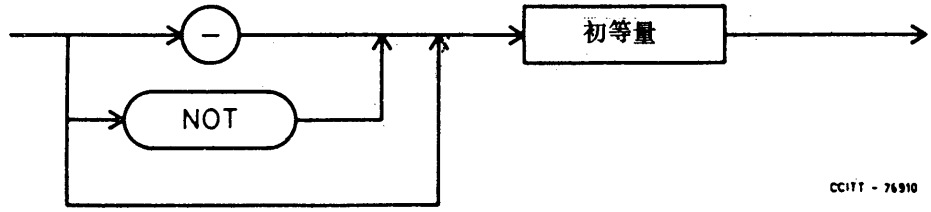


操作数 0

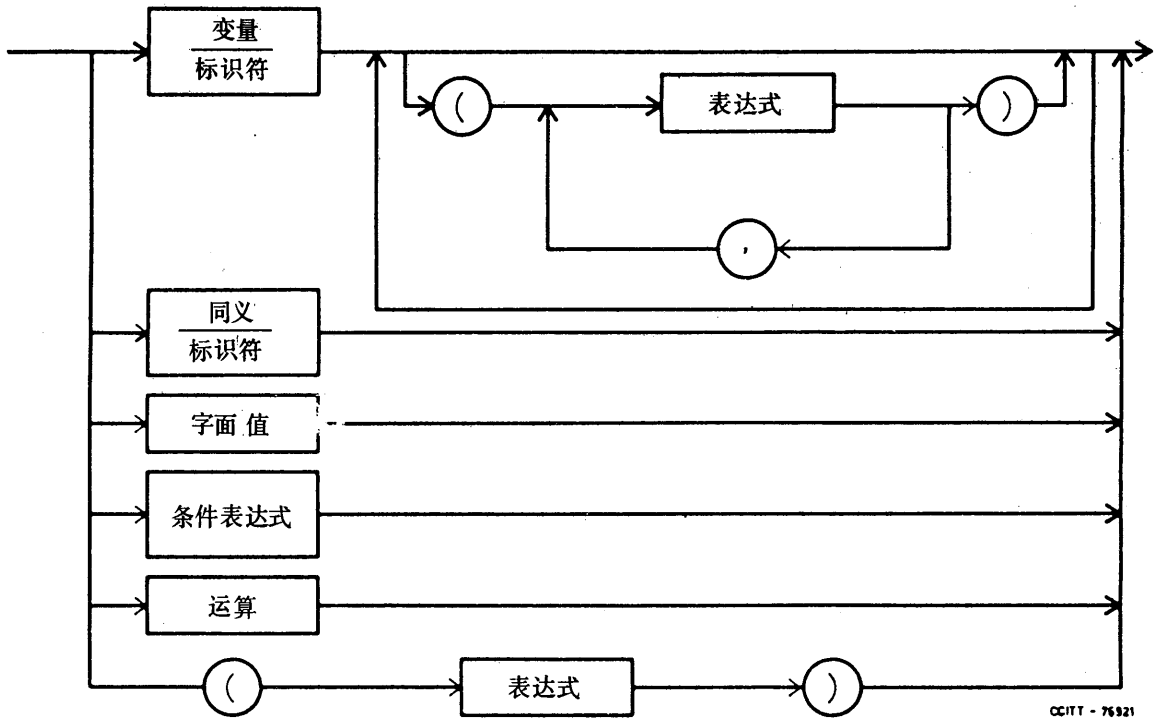



操作数 3
操作数 4
+
−
//
操作数 4
操作数 1
操作数 2
AND
操作数 2
操作数 4
操作数 5
*
/
MOD
REM
操作数 5
操作数 2
操作数 3
/=
=
>
>=
<
<=
IN
操作数 3
CCITT - 76900

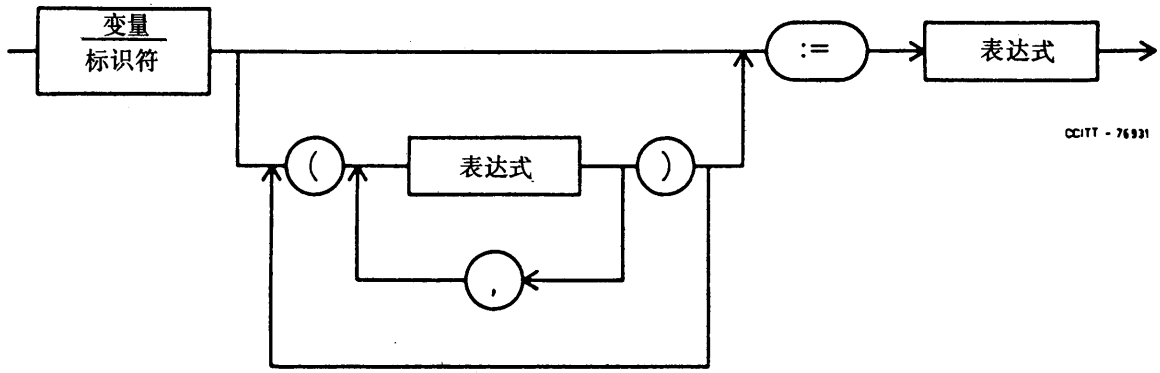
操作数 5



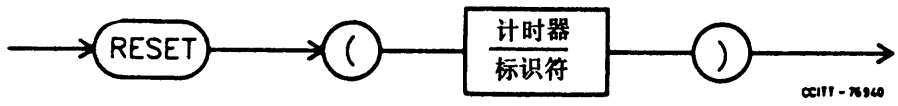
初等量



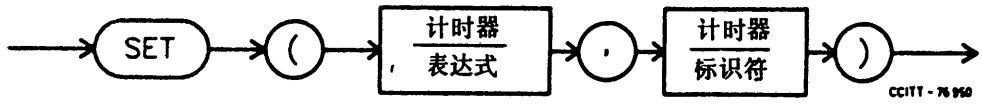
赋值语句



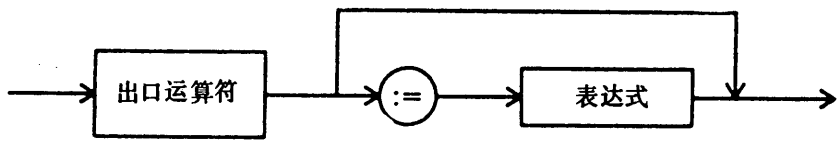
复位语句



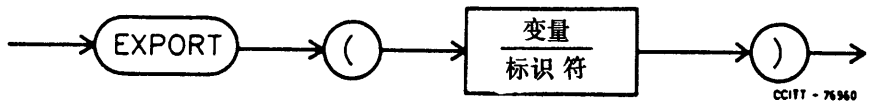
设置语句



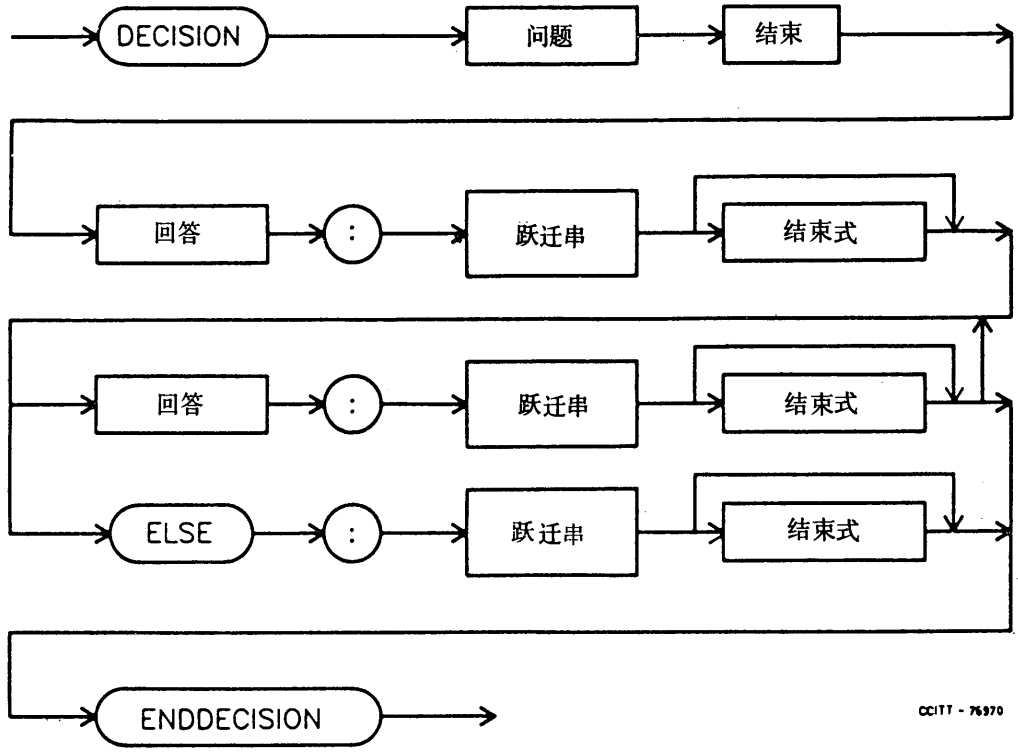
出口语句



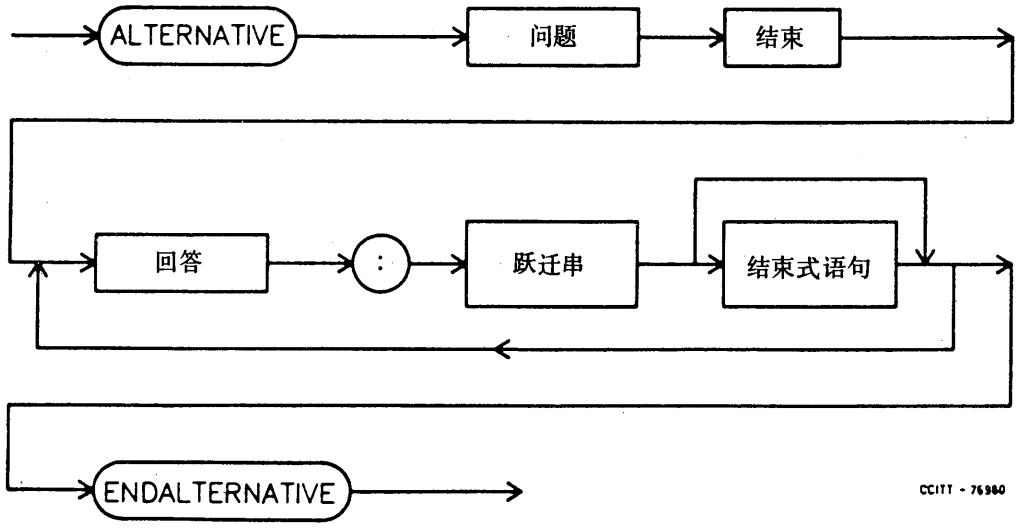
出口运算符



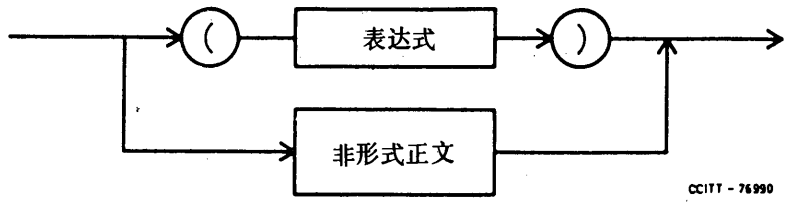
判 定


DECISION
问题
结束
回答
:
跃迁串
结束式
回答
:
跃迁串
结束式
ELSE
:
跃迁串
结束式
ENDDECISION
CCITT - 76970

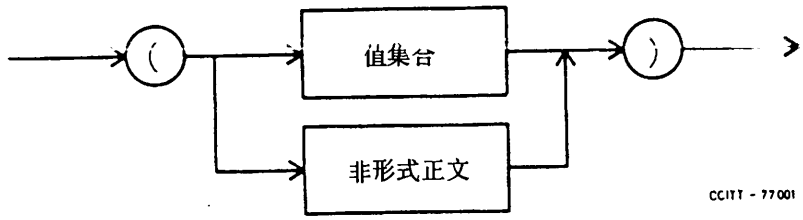
任选


ALTERNATIVE
问题
结束
回答
:
跃迁串
结束式语句
ENDALTERNATIVE
CCITT - 76980

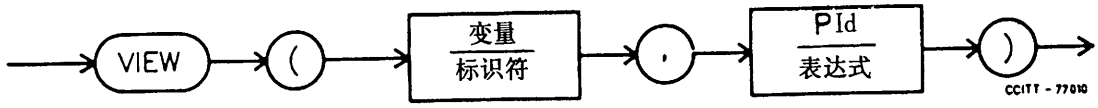
问题


(
表达式
)
非形式正文
CCITT - 76990

回答

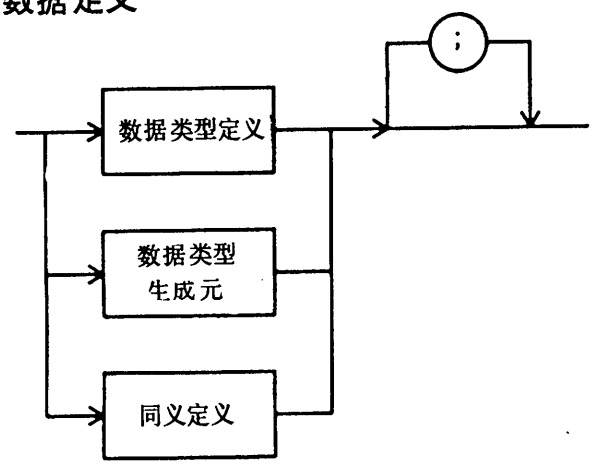


视见运算符

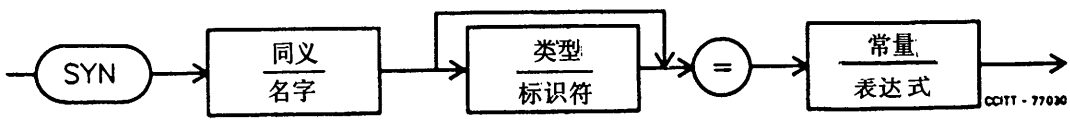


进口运算符

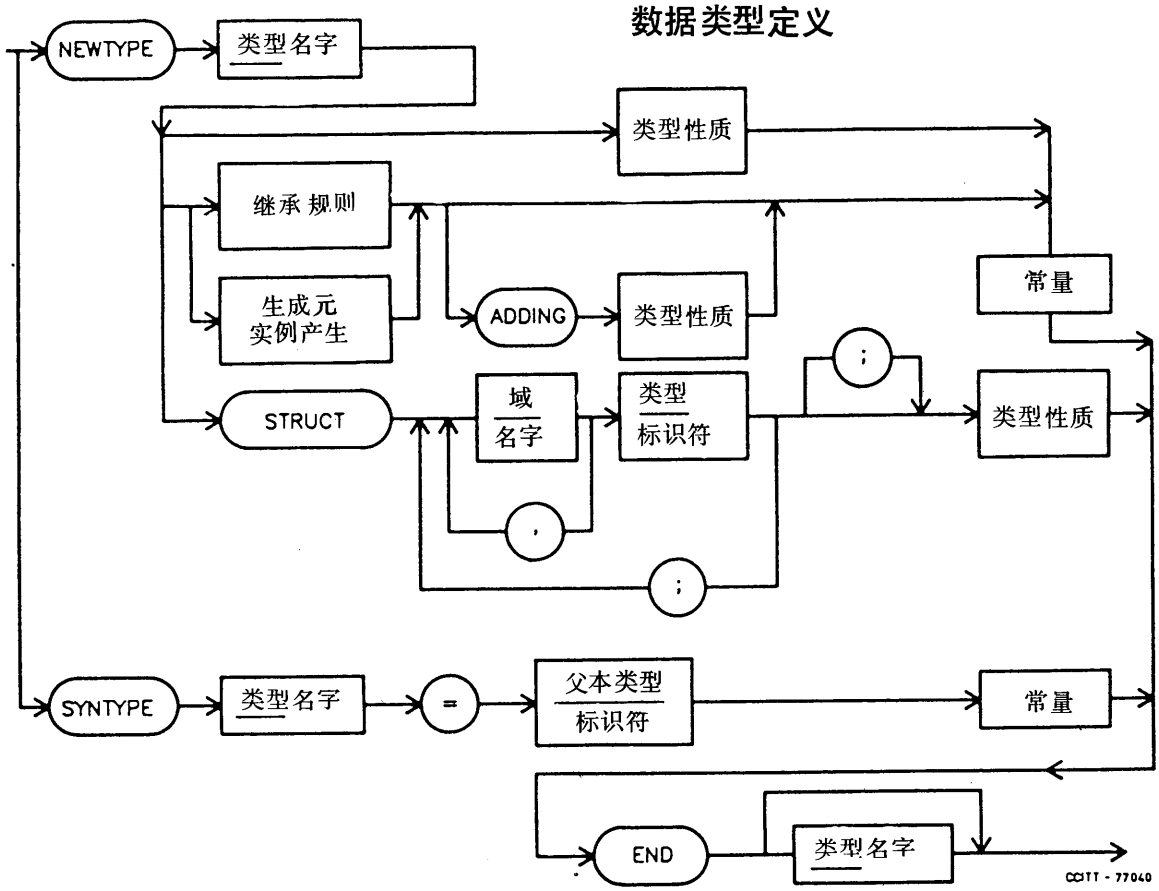
数据定义



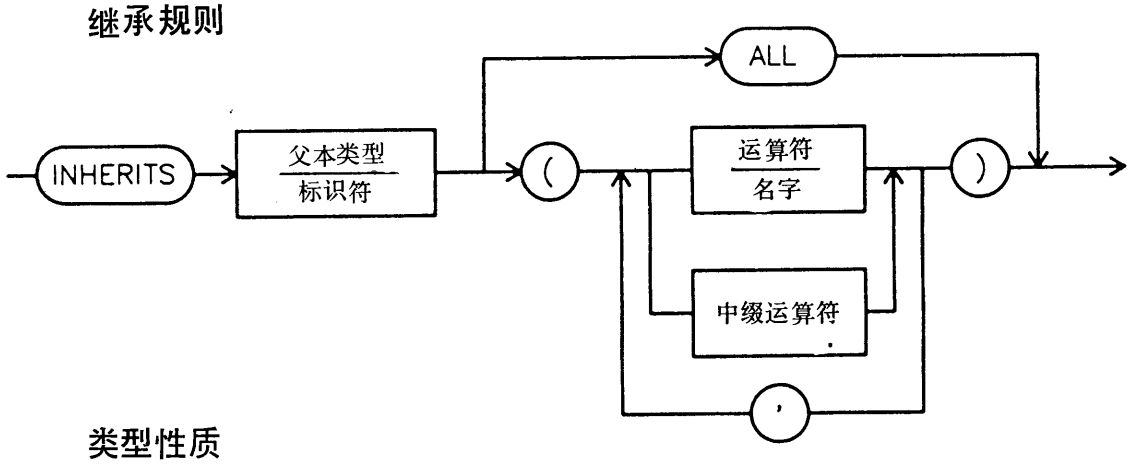
同义定义



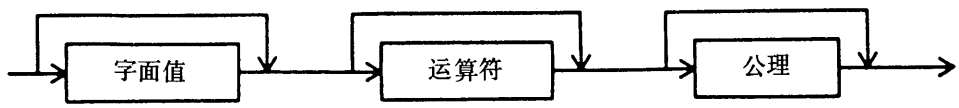
数据类型定义



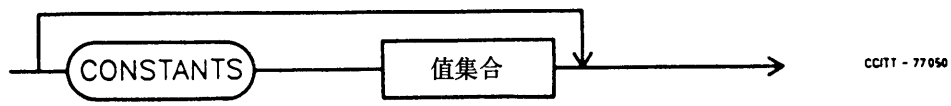
继承规则



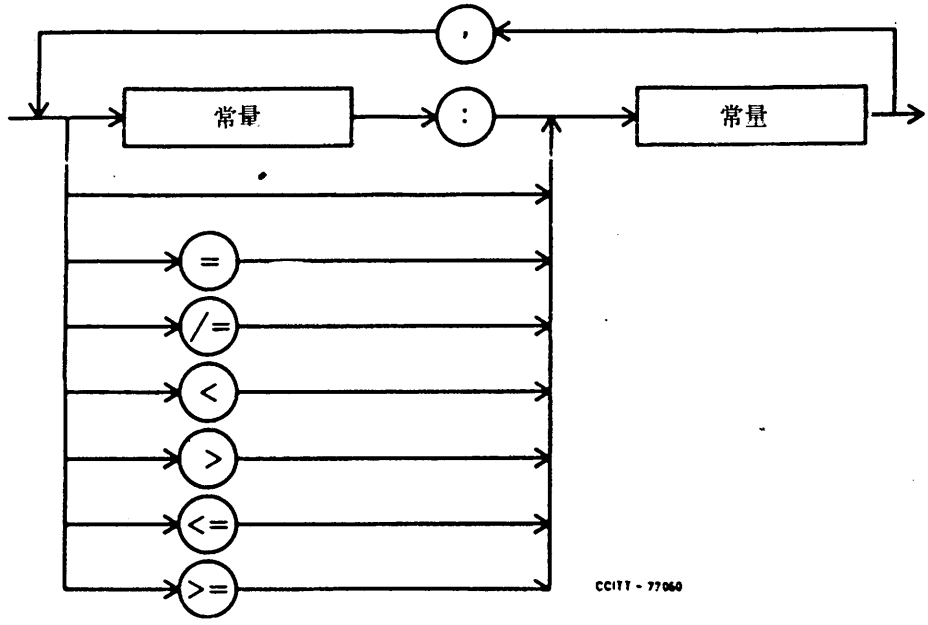
类型性质



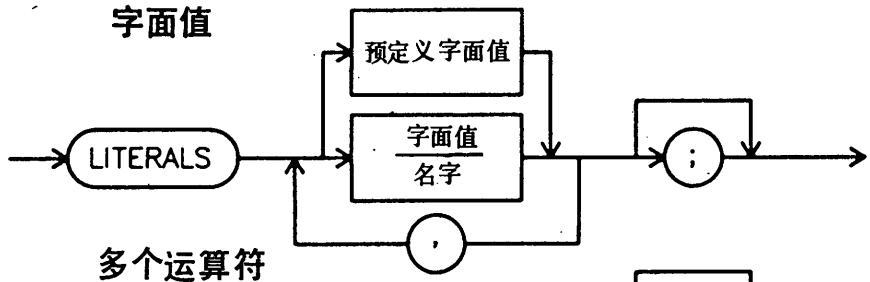
常量



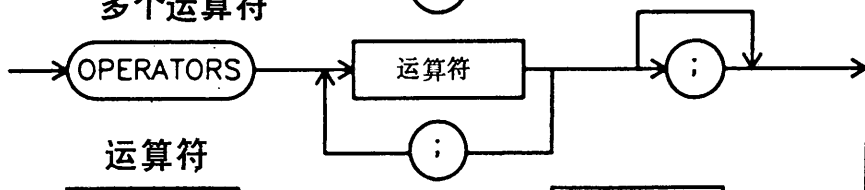
值集合



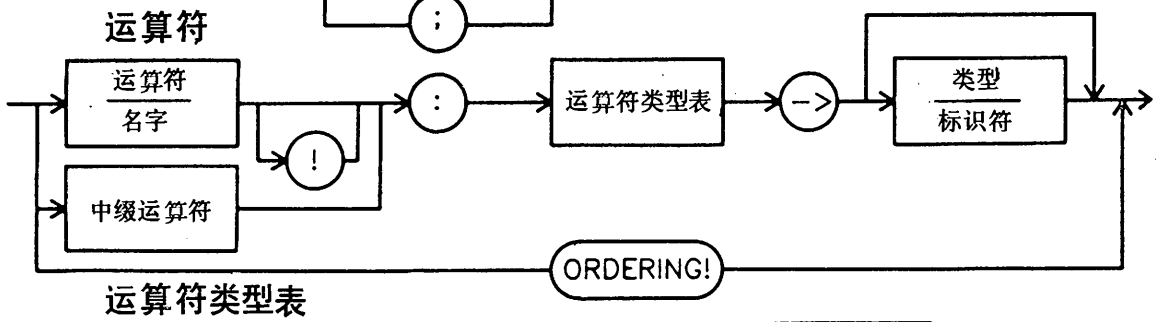
字面值



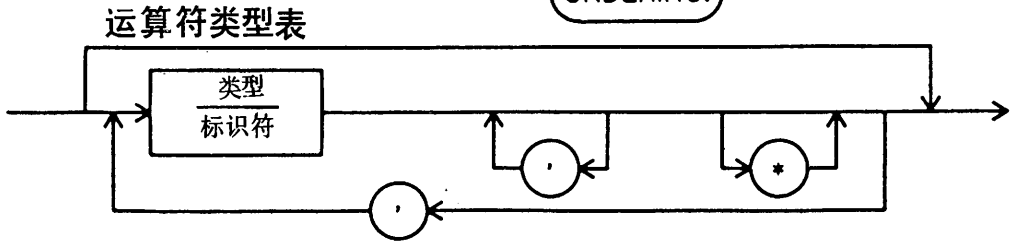
多个运算符



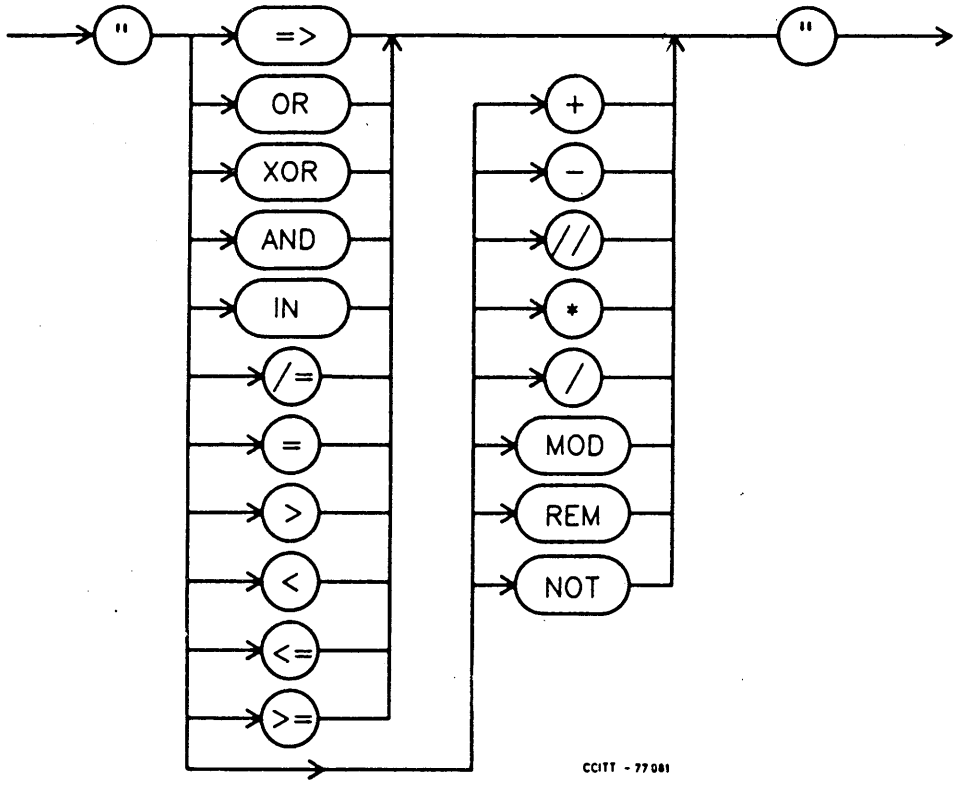
运算符



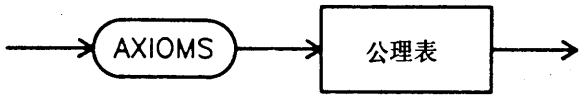
运算符类型表



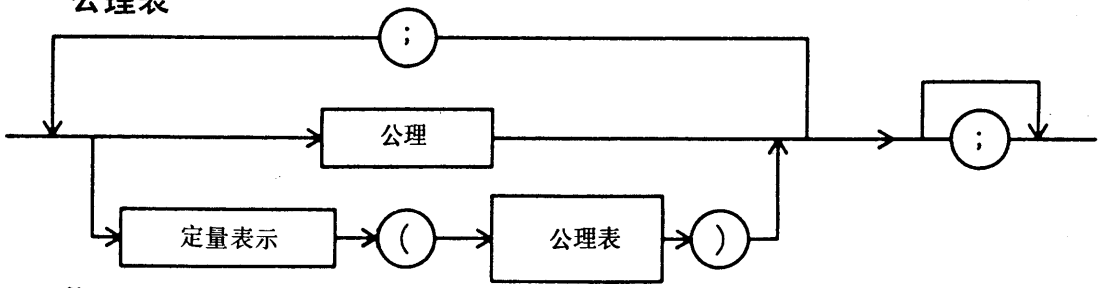
中缀运算符



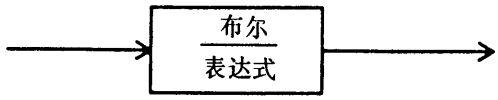
多个公理



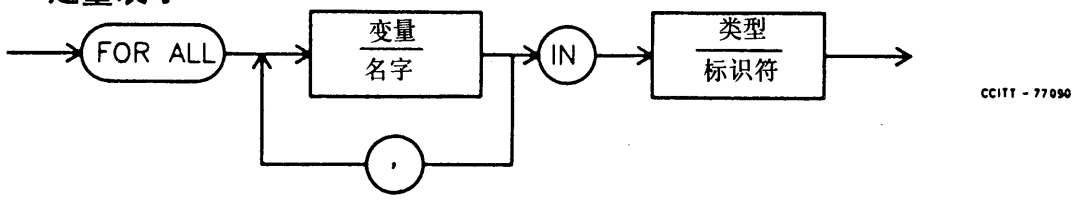
公理表



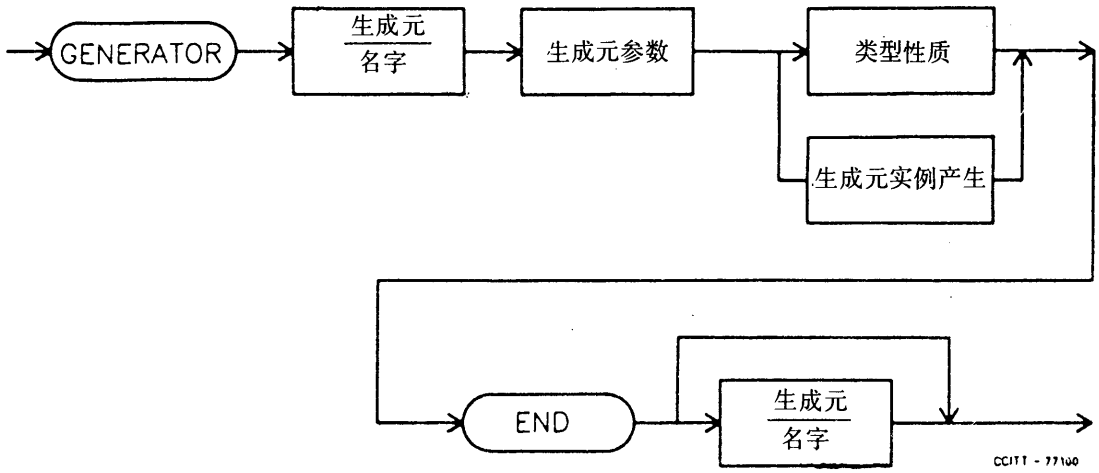
公 理



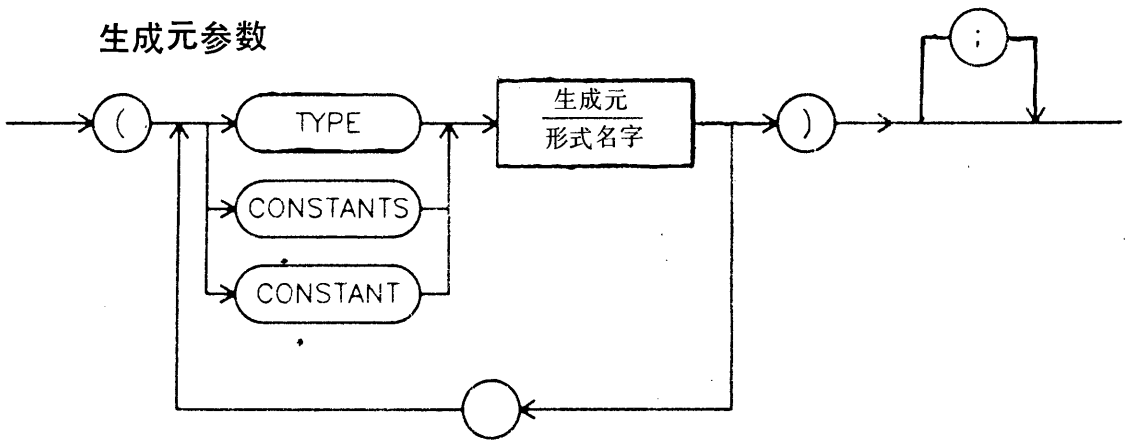
定量表示



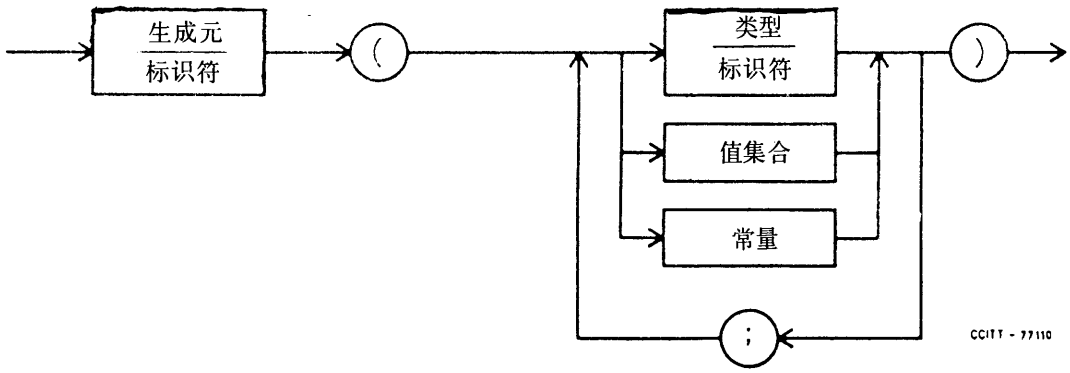
数据类型生成元



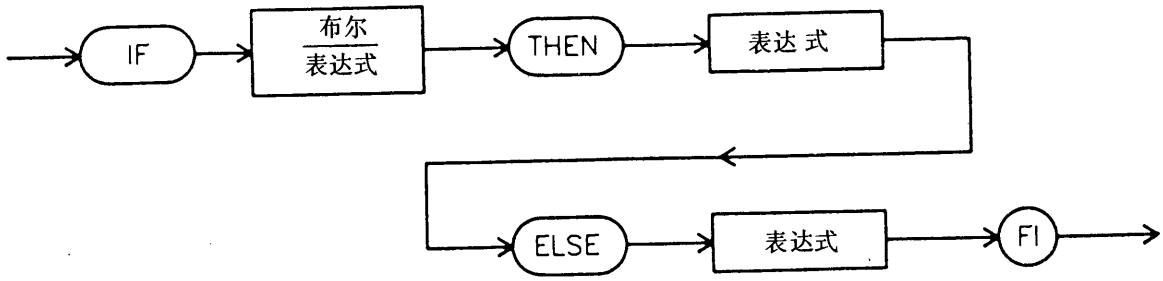
生成元参数



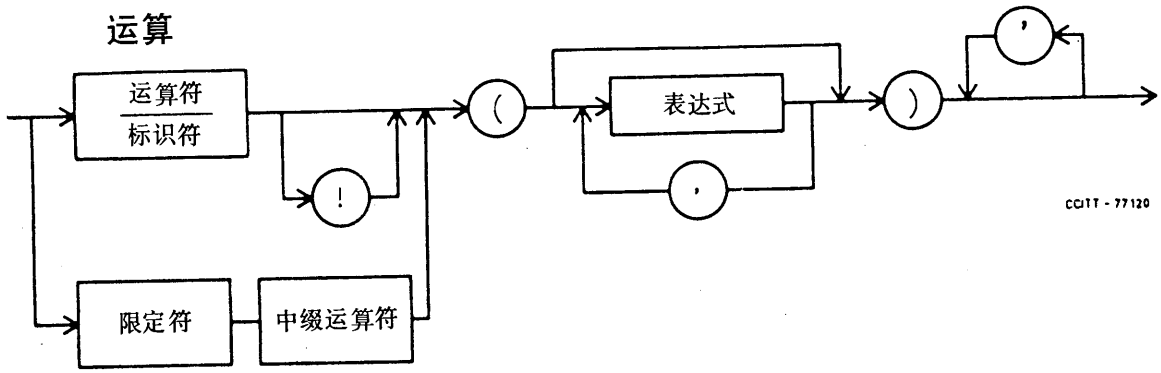
生成元实例产生



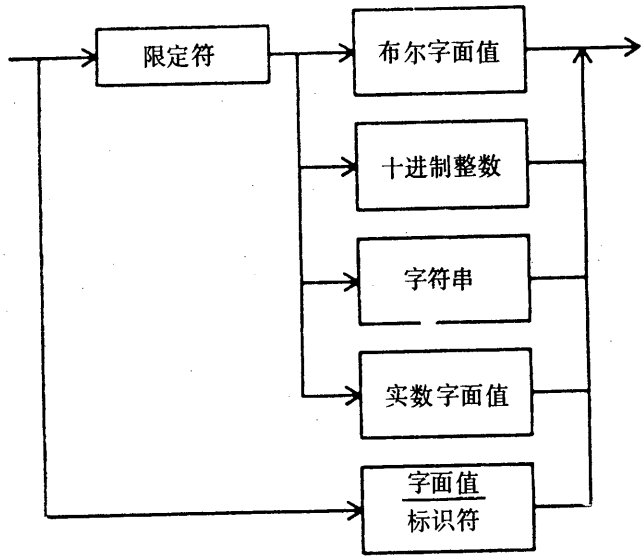
条件表达式



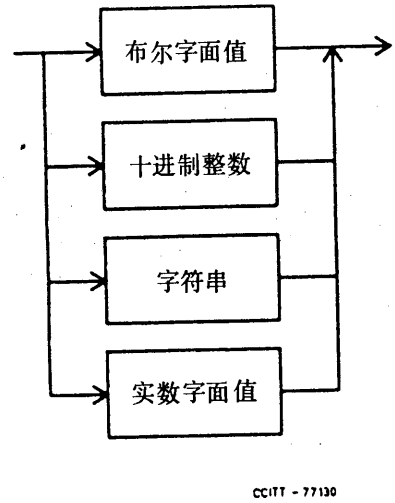
运算



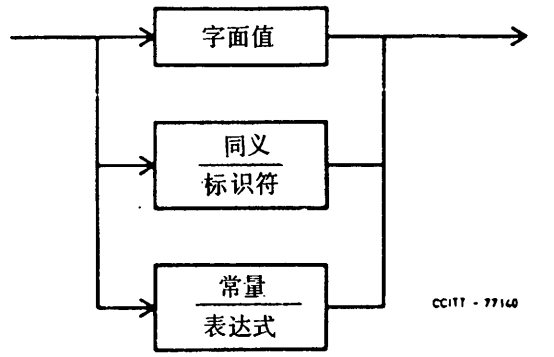
字面值



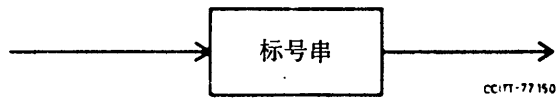
预定义字面值



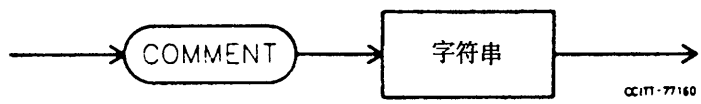
常量



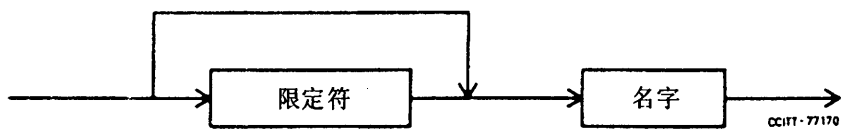
标号



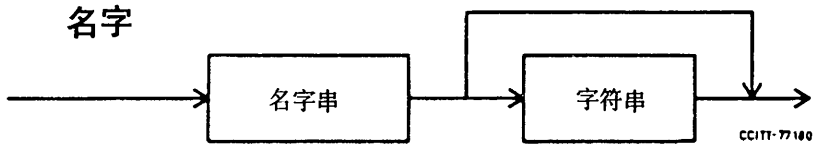
注释



标识符

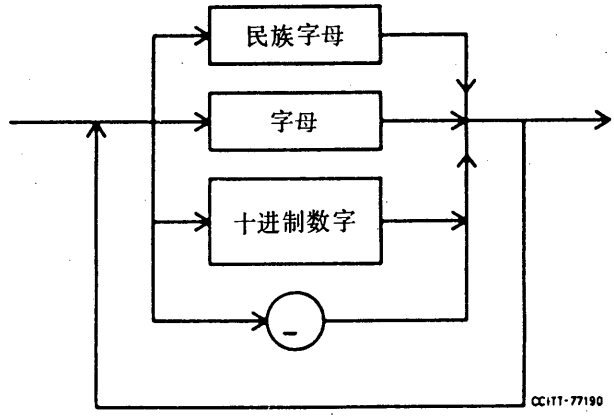


名字

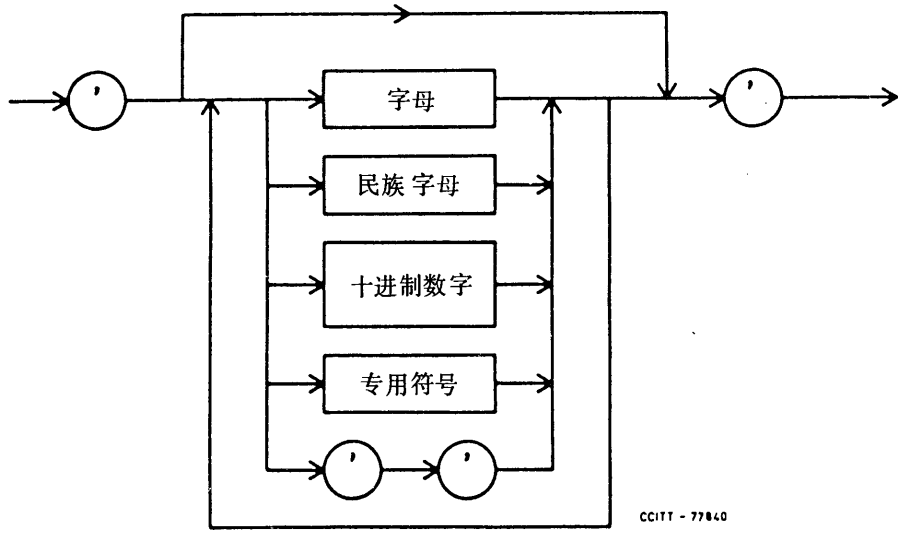


1) 所有的标点符号（如，．；’：！ =（，）和运算符（如+，－，*，<，>…）都是词法单位，它们可以代替空格。
2) 两个词法单位必须用一个或多个空格隔开。
3) 关键字和名字串属于同一个词法类型，它们都是保留的名字。
4) 在词法单位以外，几个空格和一个空格具有同样的“意义”。
5) 制表字符（VT，HT，CR，BS …）可以看成是空格。
6) 除非是在字符串之内，否则所有的字母和民族字母总是按大写字母来解释。
7) 凡是空格可以出现的地方，都可插入以符号‘/*’和‘*/’定界的注释，这些注释具有和一个空格同样的意义。注释内不得含有专用的序列‘*/’。

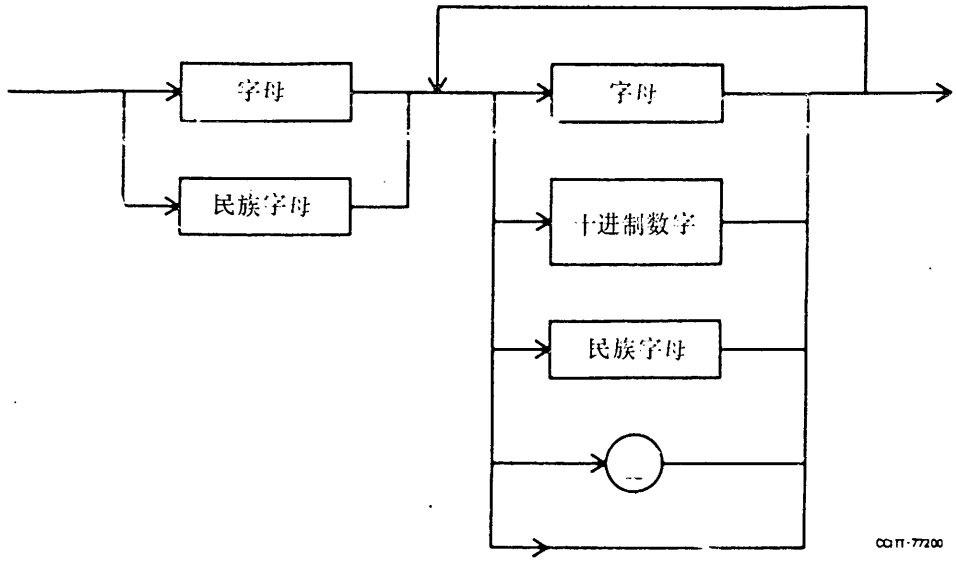
标号串



字符串



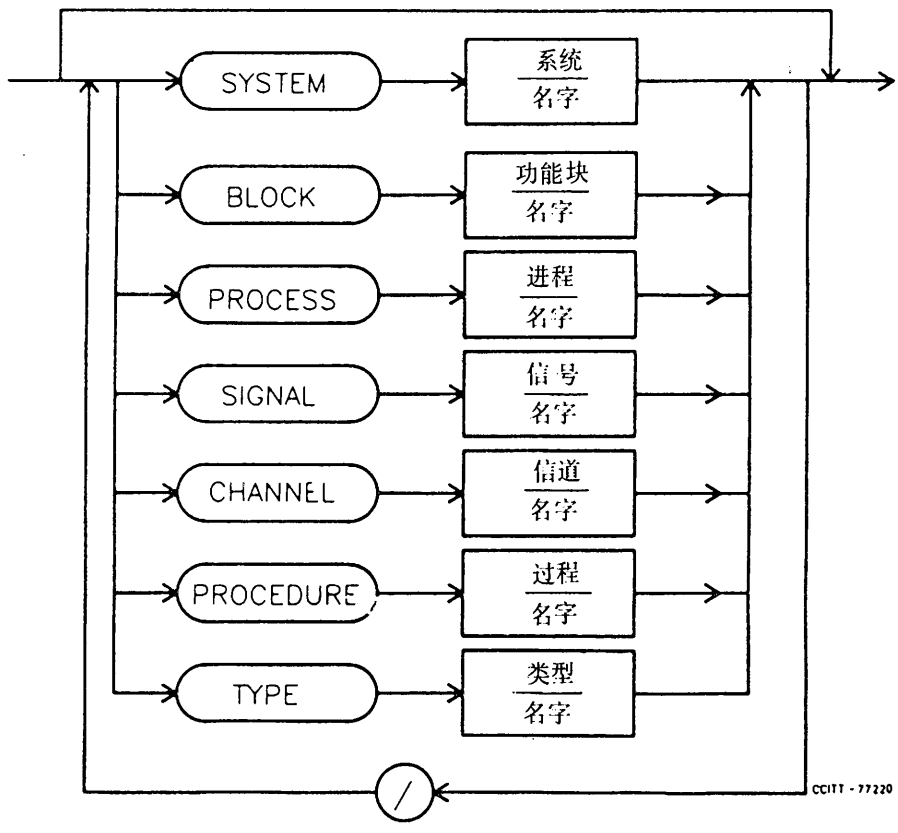
名字串


字母
字母
民族字母
十进制数字
民族字母
_
CCITT-77200

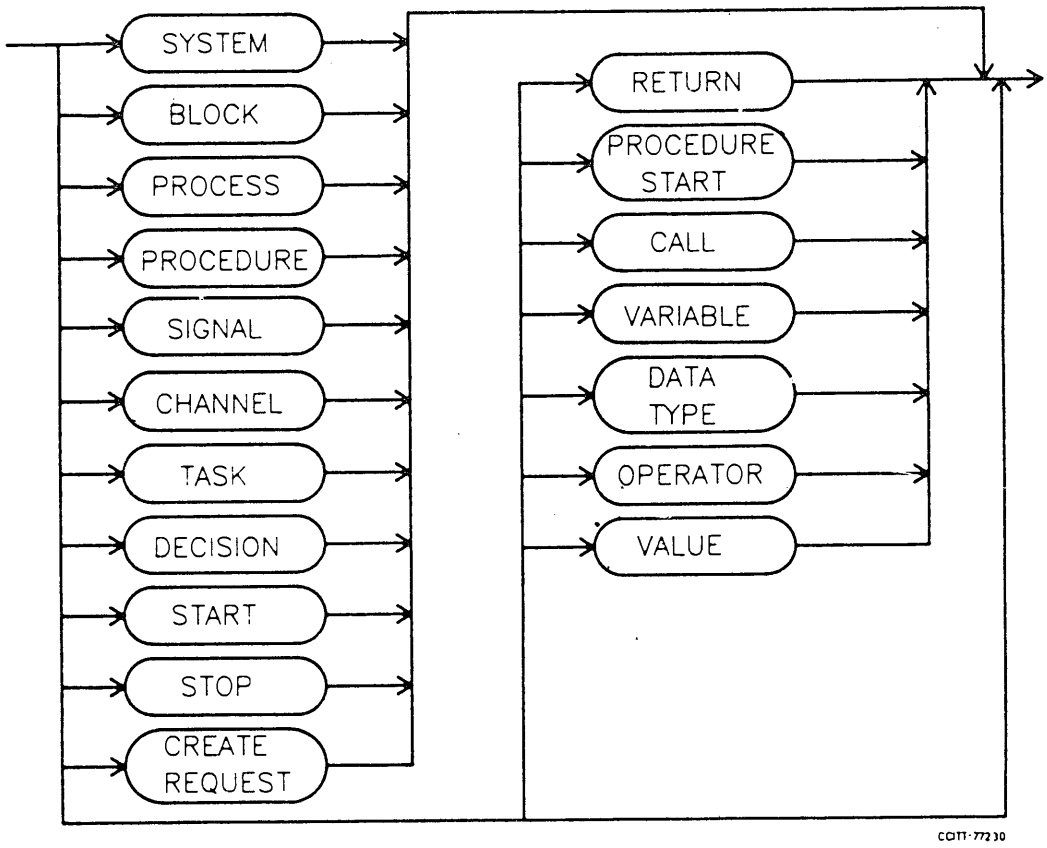
限定符


结构名字
实体类型
名字
CCITT-77210

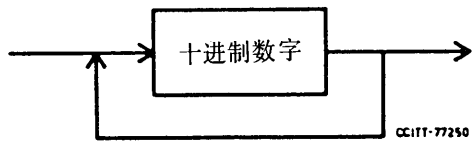
结构名字


SYSTEM
系统
名字
BLOCK
功能块
名字
PROCESS
进程
名字
SIGNAL
信号
名字
CHANNEL
信道
名字
PROCEDURE
过程
名字
TYPE
类型
名字
/
CCITT-77220

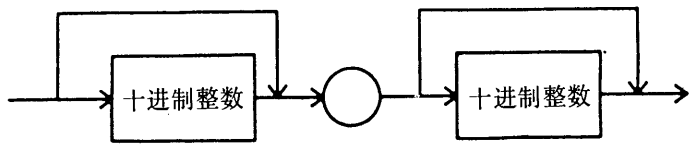
实体类型名字



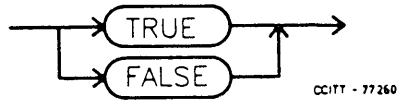
十进制整数



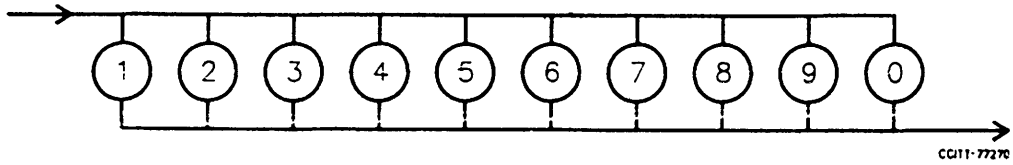
实数字面值



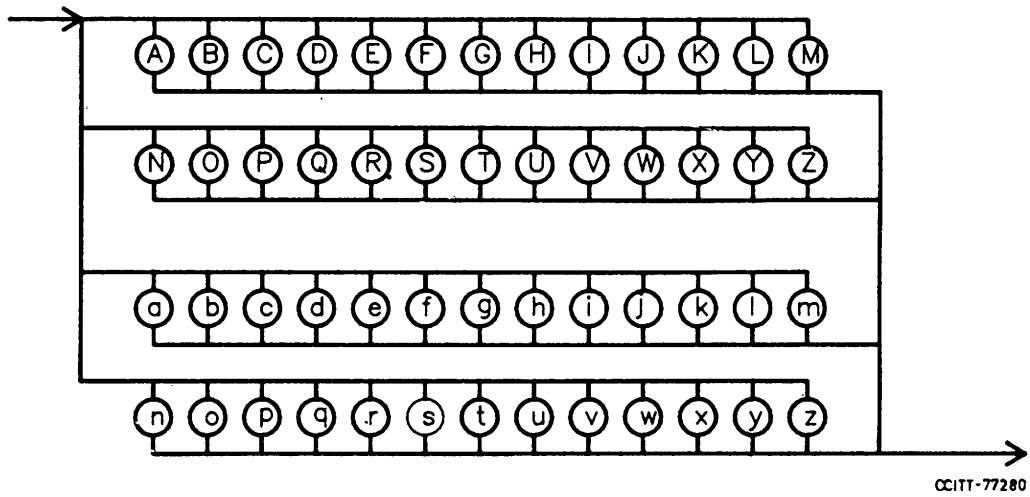
布尔字面值



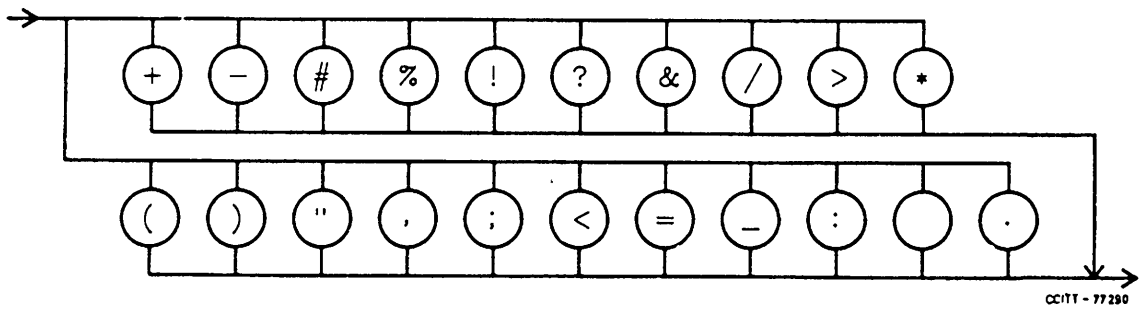
十进制数字



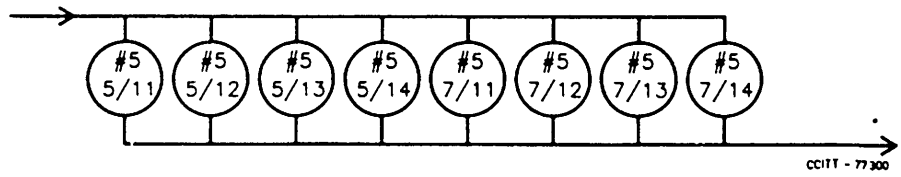
字母



专用符号

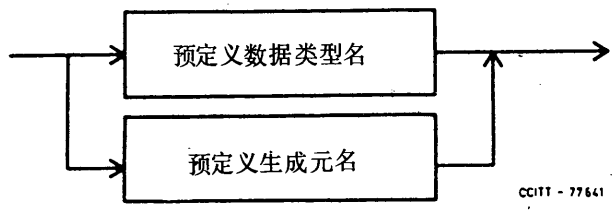


民族字母

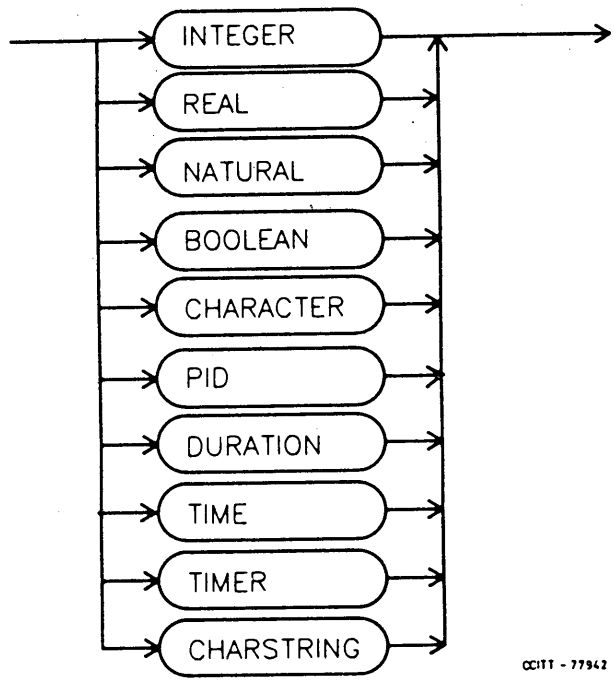


注-以上所指位置对应于CCITT 国际第5号字母表中为各国保留使用的位置。

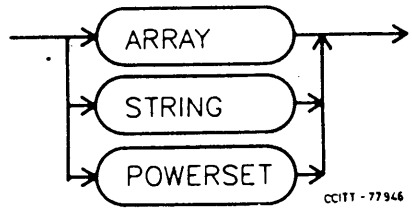
预定义类型名



预定义数据类型名



预定义生成元名



C2.3 PR 中使用的保留字

SYSTEM
ENDSYSTEM
BLOCK
ENDBLOCK
SUBSTRUCTURE
ENDSUBSTRUCTURE
SPLIT
INTO
SUBBLOCKS
CHANNELS
SIGNALS
IN
CHANNEL
FROM
ENV
TO
WITH
IMPORTEDVALUES
SIGNAL
INCOMING
OUTGOING
BLOCKS
DECISION
ELSE
ENDDECISION
ALTERNATIVE
ENDALTERNATIVE
VIEW
EXPORT
IMPORT
CREATEREQUEST
CALL
DATATYPE
VALUE
PROCESS
ENDPROCESS
PROCEDURE
ENDPROCEDURE
DCL
START
STATE
INPUT
SAVE
PROVIDED
PRIORITY
REVEALED
EXPORTED
VIEWED
IMPORTED
FPAR
EXPORTED/IMPORTED
IN/OUT
IN
TASK
NEXTSTATE
JOIN
STOP
RETURN
OUTPUT
CREATE
COMMENT
RESET
SET
EXPORT
PROCEDURESTART
VARIABLE
OPERATOR

注1－采用语法：

MACRO　宏名字

可把宏调用插入任何图中。

宏扩展是SDL/PR 程序中的一小段程序，以MACRO EXPANSION　宏名字开始，并以

ENDMACRO　宏名字

结束。

在最后这个语句中，宏名字不是必须要有的。

注2－在功能块定义图中，以下规则有效：

— 如没有功能块子结构定义，则必须至少有一个进程定义（它可显式地定义在另一模块中）。

— 若有功能块子结构定义，则包含在该功能块定义中的每个进程定义都有一进程子结构定义。

附件D

（属于建议Z .100至Z .104）

SDL用户指南

本用户指南可分为三部分。

第一部分包括目录（D.0）、序（D.1）、引言（D.2）和SDL的应用领域（D.3），这一部分介绍了SDL的内容和应用领域。

第二部分包括SDL的一般说明和基本概念（D.4）以及SDL的构成（D.5），介绍了如何用以及用何种概念才能用SDL来塑造系统。这是一个通用的部分，尚未给出SDL具体形式的细节。

最后部分包括：用SDL/GR表示系统的指南（D.6），用SDL/PR表示系统的指南（D.7），SDL/GR、SDL/PR和CHILL间的映射（D.8）以及SDL的应用例子（D.9）。这一部分给出了SDL的两种现有形式即SDL/GR和SDL/PR的用法指南。最后一节介绍SDL的工具（D.10）。

目　录

D.1 序

CCITT 规格和描述语言通称SDL，首先于1976年由建议 Z.101至 Z.103（桔皮书，卷VI.4）定义，后来于1980年在建议 Z.101至 Z.104（黄皮书）中加以扩充，并于1984年进一步得到扩充和承认，成为建议 Z.100至 Z.104（红皮书，即本文本）。

对第XI研究组来说，这一点是明显的，即为了便于把SDL 应用于电信交换系统的广阔范围，需要有用户指南。用户指南的目的是帮助用户了解SDL 各建议，了解它们在不同领域中的应用。由于SDL 的应用领域本身还在扩大，要产生一套详尽的用户指南是困难的，因此，在下一个研究周期1984—1988期间，这个文本内的用户指南本身将要求改进和扩充。CCITT 将欢迎以SDL 实践经验为基础的对本用户指南提出的改进意见。

现在（1984年）已很明显，SDL 正被CCITT 及其成员组织更广泛地使用着，且其应用范围将继续扩大。这里的用户指南希望通过对SDL 建议Z .100至Z .104补充以具体的建议和有用的例子，来帮助正在考虑使用或正开始使用SDL 的人们。本用户指南和建议有某些重叠，相信这是需要的，这样可使用户指南本身是自成体系和易于阅读的，然而主导文件仍然是建议。

D.2 引言

D.2.1 概述

SDL 可用于规格(对系统所要求的行为的规格）和描述（对系统的实际行为的描述）。SDL 能适用于电信交换系统的行为的规格说明和描述，但也能用于其它应用。实际上，对于系统行为能用扩展的有限态自动机来有效地模拟，且其重点在交互作用方面的所有系统，SDL 都是很适用的。

SDL也可作为文件编制方法的基础，以便完全地表达一个系统规格或系统描述。在这个意义上规格和描述的意义与它们在系统生命期中的应用有关。两者都抽象地定义一个系统的功能特点。描述通常包括某些与设计有关的方面（例如差错处理），对功能细节讲得比较完全。两者在涉及具体系统的设计时应协调一致，因此在系统实现以前两者都用作为规格说明，而在实现以后则都用作为系统的文件（描述）。

在规格说明或描述中，SDL可用来在详细程度不同的层次上表示一个系统的功能特性，或者表示一种功能。功能特性包括某些结构特性（功能块交互作用图）和行为。“行为”的意思是收到信号（输入）时的响应方式，即要作一些动作，例如发送信号（输出），提出问题（判定）和执行任务等。

当主管部门企图用新的特色、新的业务、新的技术等来革新一个系统而探索其可能性，同时允许设备供应者提出多种设计方案时，规格可以提得非常粗略和一般，这类规格常常不很具体。另一个极端是这样一种规格，主管部门在其中要求为某一现有的交换局更换或增加设备。这种规格显然更为详细，因为必须要有极为具体的接口规定。

一个规格和一个描述可以是完全相同的。在任何情况下，在新的开发中，从规格来导出设计是比较可取的，这样可保证一致性。

一般说来，一份描述是由设备供应者为响应一份规格而写出的（虽然一份描述也可以是描述供应者企图出售的系统）。一份描述通常比规格具有较多的细节层次，因为它需要描述系统的具体行为。

还要指出，SDL提供了以不同的形式化程度描述一个系统的手段。

第一，可以用附有自然语言的SDL构造来描述系统。这样形成的描述只将信息传递给懂得正文的读者，而不是传给机器，机器仅能自动地进行极其有限的校验。

第二，有可能给SDL 构造配备形式语句，它由已定义的各种类型的元素和作用于这些元素的运算(或操作)符组成。这些元素的性质不是规范化的：一个例子是“连接A—B”,这里A和B属于用户类型，而连接是该类型允许的一种操作。这样形成的描述将信息传递给了解所用操作符意义的读者,机器能在一定程度上懂得这种描述，并能进行校验，但它不能进行完全的校验，也不能“实现”该系统，因为不知道操作符的性质。

第三，有可能也提供全部运算符（或操作符）的全部性质。在最后的这种情况下，描述是完全形式的，机器能进行全部校验，且能从概念上实现所描述的系统。

根据使用目的，可以采用这些不同层次的形式化表示来作出描述，以适应用户的需要。当然，描述的形式化程度愈高，人们读懂它就愈困难。

D.2.2 SDL 的语法形式

由建议Z.100至Z.104所定义的SDL是一种独特的语言，它具有两种不同的语法，两者都建立在同样的语义模型基础上。一种称为SDL/GR（SDL图形表示法),它的基础是一套标准化了的图形符号和规则；另一种称为SDL/PR（SDL正文短语表示法),它的基础是类似程序的语句。两种表示法表示相同的SDL 概念。

D.2.3 SDL的基础是扩展的有限态自动机模型

在SDL的应用中，所要规定的系统用许多互相连接的抽象机器来表示。一个完整的规格要有：

1) 系统结构的定义，由机器及其相互连接来表述；
2) 每个机器的动态行为，由它对其它机器和环境的交互作用来表述；
3) 对交互作用数据的运算。

动态行为借助一些模型来描述，这些模型定义诸抽象机器的操作机制以及机器之间的通信。在SDL.中所用的抽象机器是确定的有限态自动机（FSM)的一种扩展。FSM有一有限的内部状态存储器，以离散而有限的输入、输出集合进行操作。对于每一种输入和状态的组合，存储器规定一个输出和下一个状态，通常认为从一个状态跃迁到另一个状态所需时间为零。

FSM 的一个限制在于所有需要存贮的信息必须表示成显式状态。尽管用这种方法能表示大多数系统，但它不总是实用的。会有许多要记忆的值，它们对未来的序列安排是重要的，但对从总体上了解系统则关系不大。这种信息不应占用显式状态表示的空间，因为它会使得这种表示过于拥挤。对于这类应用,FSM可用辅助存储器和对该存储器的辅助操作来扩展。地址信息和顺序号是适于存贮在辅助存储器中的信息例子。

SDL 诸建议定义了两种辅助操作，即判定和任务，它们可包括在可扩展的有限态自动机（EFSM)的跃迁之中。“判定”检查与输入有关的参数，并且检查辅助存储器中的信息（如果这些信息要影响主机器跃迁的序列安排的话)。“任务”执行诸如计数、操作辅助存储器、以及操纵输入输出参数等功能。

在SDL 中，机器间的交互作用由信号来表示，即：诸EFSM接收信号作为输入,产生信号作为输出。信号包括一个唯一的信号标识符以及一组任选的参数。SDL 允许跃迁的时间大于零。并且SDL 对信号规定了一种先进先出原则的排队机制，适用于正当机器在执行一次跃迁的期间有多个信号到达机器的情况。按照信号到达的次序，每次考虑一个信号。

D.3 SDL的应用领域

图D—1示出可以使用SDL 的范围，这是就电信交换系统的供求意义上来说的。

在图中，矩形框表示典型的功能群体，他们的具体名称可随不同的机构而不同，但是他们的活动对许多管理机构和制造商来说都是典型的。每条有向线（流线）代表一组从一个功能群通向另一个功能群的文件，SDL可用来作为每一组文件的组成部分。该图仅用来作图解说明，既不是定义性的，也不是完善的。

SDL应用的领域是能够由可通信的扩展的有限态自动机有效地模拟的项目，例如电话、用户电报、数据交换、信号系统（如七号信号系统）、信号系统和数据规约的互相配合、用户接口（MML）。

当专门考虑SPC交换系统时，能使用SDL来作说明的功能的例子是：呼叫处理（如呼叫接受处理、路径选择、发信号、计费等），维护和故障处理（如报警、自动清除故障、系统构成、例行测试等），系统控制（如过载控制）和人机接口。SDL 的应用例子见§D.9。

用SDL来说明规约的规格已在CCITT X系列建议中涉及，将SDL 应用于这个领域的辅助性用户指南附在这些建议中。

D.4 SDL的一般说明和概念

本节包含SDL的概念，而不给出其具体形式（即SDL/GR 和SDL/PR）的细节，这些细节分别在§§D.6和D.7中描述。

D.4.1 SDL概述

SDL是作为一种表现功能特性的手段而开发的，在电信系统中要规定和实现这些功能特性。

一个系统的规格或描述由两大类信息组成。第一类包括功能规格或功能描述，第二类包括较为一般的系统参数以及诸如环境条件（温度范围、湿度等）、传输界限、交换局服务等级等信息的详细说明，这些是不出现在SDL中的。

开发SDL的主要目的之一，是使用户们可以把大的系统分解成许多小块，小到足以使SDL的用户们（作者和读者）容易理解。因此在SDL中，每个系统都是由功能块组成的，各功能块由信道连接起来，信道用来表示功能块间的通信路径。这些功能块和信道可以进一步划分成多个信道和功能块。

在最低层（且常在较高的一些层次上），功能块含有一个或多个进程。进程是一些可进行通信的扩展的有限态自动机，用来塑造系统的动态行为。进程也可以是大而复杂的。过程的使用允许人们进一步将一个进程分解成较小的、便于管理的小块。

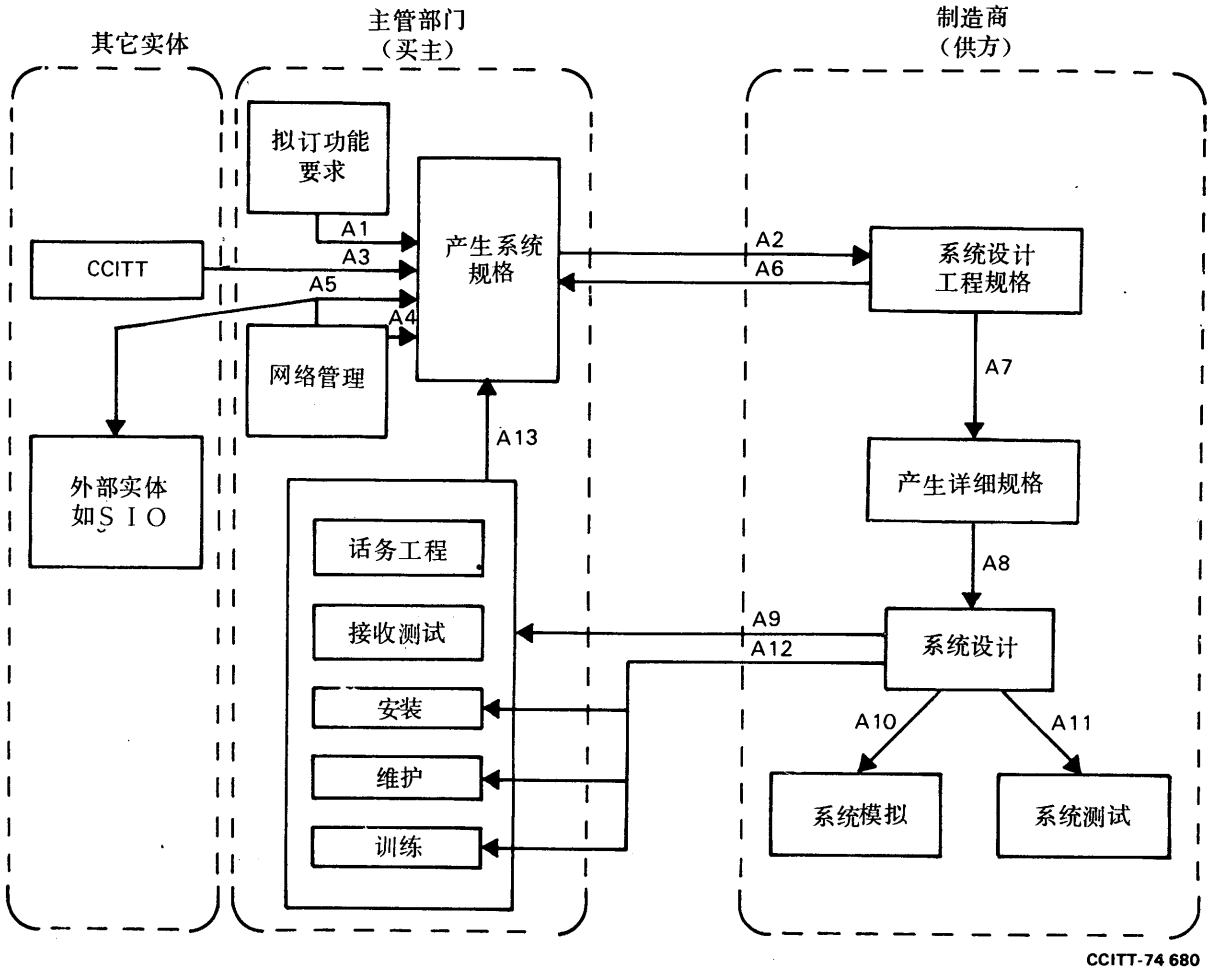
D.4.2 SDL系统的构成

D.4.2.1 概述

这一章将讨论某些SDL构件，它们可组成系统的不同结构，以便从系统结构的总体概貌出发，逐步提供有关该结构的愈来愈多的细节，以显示每个部件的内部结构。在SDL中，一个系统的最小结构在建议Z.101中描述，即系统由一组用信道连接的功能块组成—而功能块含有进程（见图D－2）。

对于不需要进一步划分的系统，Z.102的概念就不必要了。关于划分成几个详细程度不同的层次的概念包含在建议Z.102中。

全面地观察一个系统所有部件的内部结构，我们会发现同一系统的不同结构，其详细程度或者较高，或者较低，我们就说该系统结构可在不同层次上表示，而各层次的详细程度不同。



A 1 一种功能或特性的规格要求，它与实现方法无关，并与网络无关

A 2 一份与实现方法无关但与网络有关的系统规格，包括对系统环境的描述

A 3 CCITT建议和准则

A 4 对系统规格提出有关网络管理和运行方面的要求

A 5 有关的其它建议

A 6 一个实现的建议的描述

A 7 一个工程项目的规格说明

A 8 一份详细的设计规格说明

A 9 一份完整的系统描述

A10 系统和环境的适当的描述文件，供系统模拟之用

A11 系统和环境的适当的描述，供系统测试之用

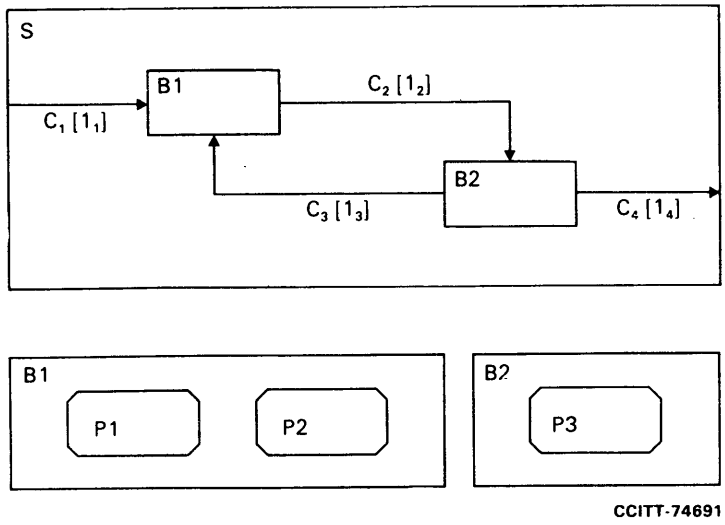
A12 安装和操作手册

A13 来自主管部门内部的专门功能小组参加拟订系统规格

注 1 - 在所有层次上均可能反复多次。

注 2 - 在某些情况下，此图中看作是一个机构内部的SDL文件如A 1、A 7、A 8，可以提供给其他机构。

图D - 1　使用SDL的一般情况



图D－2

最小系统结构举例

请注意：一个系统部件的内部结构提供了关于该结构的较多细节，却不一定提供该系统的行为的较多细节。从概念上讲，可以区别一较详细的行为表示（例如对一个新信号的处理）和一较详细的结构表示（例如兼有系统行为结构和部件结构两者）的面貌，但实际上这两个方面互相交织在一起，所以在我们提供有关系统结构的新的细节时，我们也同时提供系统行为的新细节。为了清晰起见，本章将只说明结构方面的特点。

D.4.2.2　划分准则

从一个系统表示的高层次开始，将它分解成为便于处理的小块，这种技术称为划分。这种划分的过程要为系统增添结构。

有一些对系统表示进行划分所遵循的准则，它们是：

a）定义功能块或进程，使其大小便于理解和便于处理；

b）划分实际的软件和（或）硬件时，做到协调一致；

c）与自然的功能划分相一致；

d）使功能块间的交互作用为最小；

e）重复使用已有的表示（例如信号系统）；

f）使一描述与某一规格相映射。

实际采用的准则取决于所考虑的是规格或是描述，并取决于所要求的详细程度。

每一个功能块可以用相同的或不同的准则进一步划分。

由于层次之间的关系取决于所选用的划分准则，清楚地说明已选定哪些准则是重要的，以便读者较易地理解该表示法。

划分的准则取决于用户，但存在某些限制，以保证正确的SDL表示法。这些将在后面几节中讨论。

D.4.2.3　功能块划分

可以把一个功能块划分成为一组功能块和信道，其方法几乎与将系统划分为功能块和信道的方法一样。

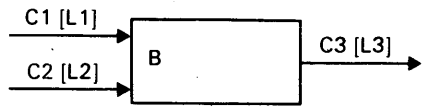
在划分过程中，有若干重要的规则要记住：

1) 连接到一个进入信道（例如图D—3中的C 1）的诸子信道（例如C 1.1和C 1.2），其信号表中必须不包含新的信号，但必须包含原信道中的所有信号。因此在图D—3所示的例子中，L 1.1和L 1.2包含了L 1中的所有信号，此外，包含在L 1.1中的信号不能出现在L 1.2中。

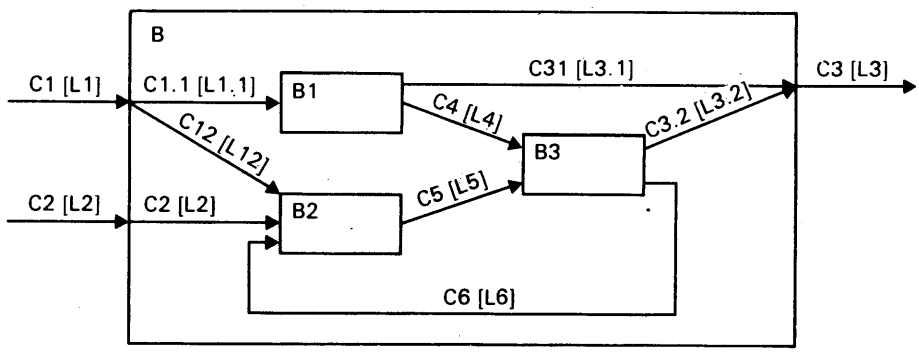
2) 连接一个走出信道（例如C 3）的诸子信道（例如C 3.1和C 3.2），其信号表中必须不包含新的信号名，但必须包含原信道中的所有信号名。因此L 3.1和L 3.2包含了L 3中的所有信号名，但与连接到进入信道的诸子信道不同，L 3.1和L 3.2可包含相同的信号名。

3) 若新信道（例如C 4、C 5和C 6）在其信号表中含有原来的功能块所不知道的信号名，这些新信号必须在原功能块（B）的内部部分的定义中作出定义。

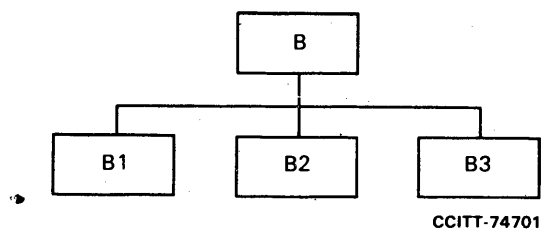
4) 若原功能块含有进程，则可有三种选择。第一，每一进程可直接分配给新的子功能块之一；第二，放弃原进程，在子功能块中代之以新的进程；第三，它们可用划分进程的规则（见§D.4.2.4）来加以划分。



a） 原功能块交互作用图



b） 划分后的功能块交互作用图



c） 功能块树图

图D—3

功能块划分

5) 父本功能块中的数据定义可用于其诸子功能块，因此这些子功能块中的每一个都能使用已在父本功能块中定义了的数据类型，毋需对它重新定义。

6) 若在父本功能块中已经定义的某一类型在一子功能块中被重新定义，则该新定义只能用于作出定义的该子功能块，而其它子功能块仍沿用老定义。不应提倡只是为了刻划某一子功能块的特征而进行一次重新定义，因为它会被读者所忽视，读者会认为老定义是有效的。在某些情况下，特别当涉及到行为有某种改进时，进行重新定义是应该的，这时应注意用适当的注解来强调这种重新定义。

D.4.2.4 进程划分

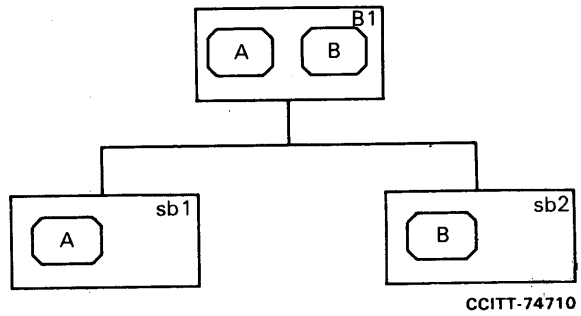
进程是表示一个功能块的（部分）功能行为的手段。

审视这个行为，有时会感到有必要把它划分成一些子行为，这相当于说：我们把一个进程划分成一些子进程。

各子进程合起来应表示与该父本进程行为（全部）相同的行为。由父本进程接收的每个信号应由一个且只由一个子进程所接收，而由父本进程产生的每个信号应由诸子进程中的至少一个来产生。

一个进程的划分可以是划分含有该进程的父本进程的功能块所引起的结果，或者也可以与任何结构的划分无关，即可以自主地划分进程。

D.4.2.4.1 一个进程可作为划分一功能块的结果来进行划分。首先应明确，划分一功能块并不一定意味着划分代表其行为的进程。会有这样的情况：一个进程代表诸子功能块之一的行为（或行为的一部分）。在此情况下进程不需要划分（见图D—4）。



图D—4

划分一功能块并不引起该功能块中诸进程的划分

.在进程所表示的行为对应于两个或更多个子功能块的行为时，则按照每个进程只能表示一个功能块（或其一部分）的行为的规则，该进程必须按该规则所要求的数量划分成多个进程。

如果功能块已划分成“n”个子功能块，则与该功能块相关联的每一进程可划分成“m”个子进程，这里“m”可为大于零的任何数目，与“n”无关。此外，还应注意，该功能块的每一进程是独立地进行划分的，与别的进程无关，因此，某个进程可划分成两个子进程，另一个进程可划分成四个子进程，而第三个进程则划分成一个子进程。

如一个进程只划分成一个子进程，则从结构的观点来看，该子进程等于该父本进程。

D.4.2.4.2 进程的划分能独立于与它有关的功能块。在这种情况下，划分父本进程所产生的诸子进程在同一功能块中代替了父本进程。

D.4.2.4.3 平行地划分进程和划分功能块，其理由可能是为了要表示出不同子功能块的行为，而与功能块无关地独立划分进程的理由可以是为了简化表示而将行为的某些具体方面孤立起来。

请注意：若把某些方面作为子行为（诸如频率确认或数字识别）来考虑，则用过程或宏来表示它们会好得多。

相反，若被孤立起来的诸行为是“主要”行为，都具有同等价值（重要性），则用子进程来表示它们较好。在前一种情况下，每个子行为可处理与主进程输入信号相同的信号，而在后一种情况下，每个子进程有一组输入信号，与其它子进程的各组输入信号无关。

对于这后一种情况可以举一个例子：把“呼叫处理进程”划分成“用户处理子进程”、“路径选择子进程”和“连接处理子进程”（见图D－5）。

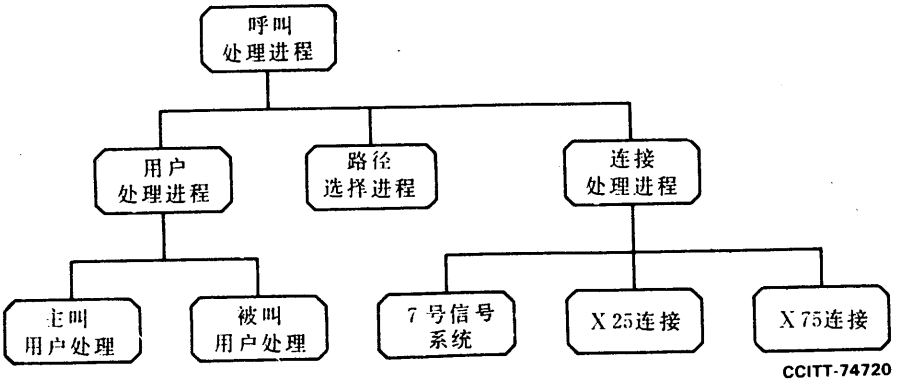
在划分进程时，要记住一些重要规则：

1) 进程的划分会引起进入信号组的划分，致使某一子进程会收到某一信号子集，另一子进程收到另一子集，等等。这些子集互不相交，但总起来包含了父本进程所收到的信号集。它们也可能含有由划分产生的其它信号，这些信号系用来传递子进程间的信息的。

2) 父本进程给出的数据定义仍可由子进程使用，毋需再重新定义。如果在父本进程中定义的数据类型在一子进程中被重新定义，则新定义只适用于作出重新定义的子进程，而其它子进程仍沿袭老定义。

3) 在诸子进程中，所有的变量都必须重新定义。在父本进程中或在该父本进程所在的功能块中，被定义的过程对诸子进程来说是可见的。

4) 对每一进程定义，可能有数目不等的进程实例。当一个进程被划分成几个子进程时，对该进程的创建请求要使得（列表在该进程子结构定义中的）每一个子进程各创建一个实例。这些子进程中的每一个都是一个进程，具有任何SDL进程的全部特性。

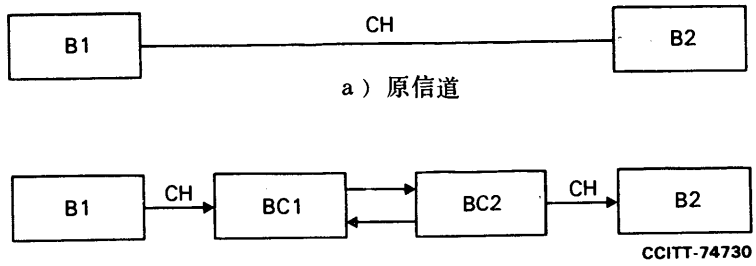


图D－5

进程划分举例

D.4.2.5 信道划分

信道可以划分，与它所连接的诸功能块无关，这样就可表示信道在传送信号时的行为。为了准确地表示信号被传送的情况，在某些情况下有必要表示信道的行为。做到这一点的办法是把信道看成是一个独立的项目，把它所连接的两个功能块看作是信道的环境，如图D－6所示。



图D－6
信道划分举例

用这种方法考察一个信道，我们就能显示它的结构（构成它的诸功能块），每个功能块和其它各功能块的互相连接，以及代表诸功能块行为的诸进程。

在进行划分的信道中，进入信道和走出信道都不分割成多个子信道，但必须只接到一个功能块。除了这个限制以外，已划分的信道具有的属性和已划分的功能块的属性相同。包含在信道交互作用图中的诸功能块和诸信道也可作进一步的划分。

D.4.2.6 功能块划分、进程划分及信道划分的相互影响

至此我们已彼此独立地考虑了功能块、进程和信道的划分，实际上这些活动是有紧密联系的，例如已可看出，一个功能块的划分通常引起连接的信道的划分以及有关进程的划分。

相反地，一个进程的划分孤立出通常由系统的特定部分处理的具体的行为（每一个行为都可在功能块的概念上得到映射）。

对一个系统表示进行划分的动机可以不同，如§D.4.2.2中所述，但结果都是表示系统的所有实量（功能块、信道或进程）的划分。

这将在后面几节较详细地讨论。当面对诸如复杂性、管理或反映实际系统结构等情况时，功能块、进程和信道的划分应该是不言而喻的。

SDL建议为划分这些实量提供了结构、规则和注解。但用户必须面对这样的事实，即在划分中会涉及其它SDL实量，以及随之而来的各种实量互相关联起来的情况。

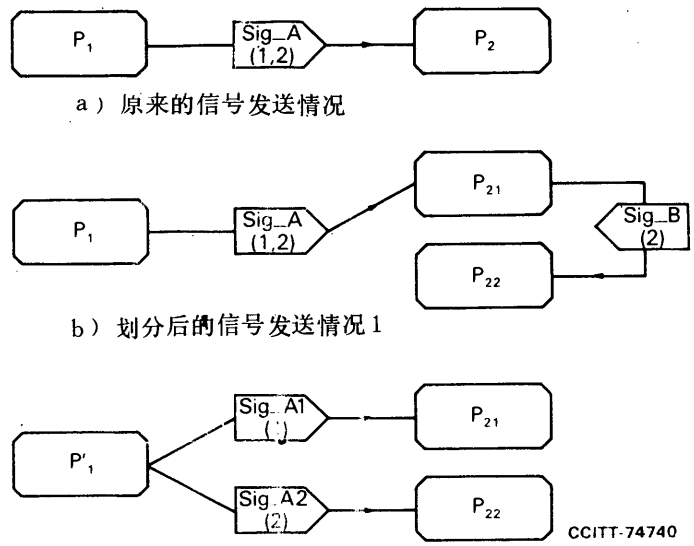
D.4.2.6.1 信号

信号从一个进程传递信息到另一个进程（环境的行为可由一个或多个SDL进程来模拟）。这种信息可以非常复杂，并且在系统生存期的某个时刻被传递，即当发送进程解释某一跃迁时，以及在这之后当接收进程到达某一状态的时候。

在把进程划分成诸子进程时，会要求“组装”在一个信号内的信息必须分配给一个以上的子进程。

这可通过两种方法来解决。在第一种方法中，信号不受影响，它将被一个子进程所接收，该子进程将产生必需的信号，以传送别的子进程可能需要的那部分信息。第二种方法则影响信号，在这种情况下它被划分成许多信号，每一个信号传给一个需要该信息的子进程。

这两种机制示于图D—7。



a） 原来的信号发送情况

b） 划分后的信号发送情况1

c） 划分后的信号发送情况2

图D—7

出现进程划分时的信号发送机制

所述的这两种机制似乎是等价的，且似乎第二种机制总能映射到第一种内。但是有一点不同，即前者进行的是串行的信息传递，而后者进行的是并行的信息传递；还有，在第二种情况下，发送进程是和原来情况下的进程不完全相同的，因为两个进程发送不同的信号。

除上述考虑外，还存在一种情况：设想在描述的高层次上我们使用一个功能很强的信号，它把多种信息类型都装配在一起了。这时第二种机制是唯一可用的方法。

当对系统表示进行划分时（这样就给出较多的结构细节），我们会处于这种境地，即装配在信号中的信息不再能包装在一起了，因为它们不是在同一地点或（和）同一时刻被利用。划分的结果是信息分配到不同的功能块中（不同的信号始发端），或在不同的时刻来使用。后一种情况的一个例子是：在高层次上的信号“被叫用户选择”在较低层次上变成一个“数字”信号的序列。请注意：我们能设想一个对所有“数字”进行收集的装配过程，然后发送“被叫用户选择”信号，但这会是一种不符合实际的表示，或者甚至更坏，会是一种不可能的表示：在我们想表明在每位“数字”后可能会出现从出线处的往回释放时，就会产生后一种（不可能）情况。

前一种情况的一个例子是信号“线路建立完毕”，该信号在一较低层次上被划分成一组由不同的子功能块发出的信号，每个子功能块控制总的线路建立中的一部分。

这些例子说明了，作为系统划分的结果，有好几种情况要把一个信号进行划分。

SDL不推荐任何特别的符号，即使其中有一些是由定义SDL的研究组建议和讨论过的。

这里建议在具有不同层次的SDL表示中为一些信号加注解，这些信号是由某一较高层次的信号划分所产生的。这会大大增加SDL表示的易读性，这些SDL表示把不同的层次衔接起来了。

应在定义某一信号时用注解说明其父本信号。对于每一个信号也应有一前向参考，用来指明由于它的划分所产生的诸信号。

应注意信号的划分既影响发送者，也影响接收者，还影响含有一些新的信号名字的信道。在几个信号被划分的情况下，由维持老信道，在其两端加子信道所带来的好处就丧失了。在某种意义上说，这种表示法已变得不利，因而现在有必要对老信道的一部分重新定义，这种重新定义的程度取决于被划分的信号的数目。

图D－7中示出的第二种方法，是以多个新信道来代替老信道。在这种情况下这种方法较有吸引力，因为尽管发送者和接收者都受到影响，但毕竟这种方法比较简单。

D.4.2.6.2 状态

将一个进程划分成多个子进程也要考虑对其组成部分即状态、判定、任务和数据的影响（输入和输出要受信号划分的影响）。

状态应该是进程的某一逻辑状况（例如通话阶段或测试阶段）的表示。将一进程划分成多个子进程导致将逻辑状态划分成为某几个子进程的一组状态（不一定全部子进程都有这些状态）。

这对层次间有相互关系的人为解释和机器解释均有影响。

增加细节的数量，会给人们造成一种抓不住总的情况的倾向，而这会引起不好的效果（有多少次在更新程序时认为作得十分完善了，但仍然对系统全局产生不好的效果!）。

逻辑上可组合在一起的细节如果分散开来，会对理解全局情况产生更为有害的影响。用SDL把一个表示在n个层次上组织起来，就是要努力避免这个问题，但重要的是要把不同的层次互相联系起来。

互相联系是可以做到的，办法是通过适当注解来引用给出全貌的较抽象（较上面的层次）的表示，并且参照必须合在一起考虑的其它部分。

机器解释的情况和人为解释是完全不同的。机器没有全局观念，即机器要考虑所有的项目，考虑每个项目是同样的认真，且每次只考虑一个项目，每次只考虑一个关系。由于这个原因，将相互连系的信息分散开并不会影响机器解释（忽略解释时的时间消耗）。机器解释的问题是把进程划分加以形式化的问题，使得机器能够懂得以前称之为“空闲”的状态现在是分散在几个子进程上的一组“空闲”状态，等等。

唯一可能使机器意识到这种对应关系的办法就是在程序中明确地作出声明(忘记作出某些声明的概率很高，故这种办法效率最低，也不大可靠)，或者是使划分进程成子进程的规则形式化。

这个问题由方法论来认真处理（并予解决），并且也是SDL建议无意中涉及到的一个方面。

D.4.2.6.3 判定

如果作出判定所根据的数据被划分了，则进程的划分将要影响判定。如果一子进程必须根据一个数据作出判定，而现在这个数据不再能得到了，则必须发送一个信号来要求该数据的值，而该进程必须等待回答。信号发送（请求）和接收（回答）的机制可以隐蔽起来，办法是声明该数据是进口的，并在一旦得到该进口数据时就驱动判决。

应注意子进程通常是分配在不同功能块中的，因此不能使用共享数据的视见机制。

D.4.2.6.4 任务

任务代表一组动作，这些动作对进程以外没有直接的影响。当进程被划分时，由任务实施的这组动作也可被划分，使某些动作在一个子进程中实施，而其它动作则在不同的子进程中实施，等等。

请注意这种分割仅在该进程的子进程内进行。如某一组动作在一给定的进程内被实施，则同一组动作就被该进程的诸子进程全体所实施。在这种情况下，这和状态划分有微小的不同，状态的划分会取决于其它进程的划分，因为一个进程的某个状态是某种逻辑状况的一部分，该逻辑状况可包括多个进程。

再回到任务上来。将诸动作划分给几个子进程实施往往要求引进一些新信号。一旦一个子进程所执行的动作结束了，就用新的信号来触发另一子进程来执行原先动作中的一个子集。

任务划分通常是由进程的数据划分所引起的一种副作用。设计（和表示）方法论可再次提供形式化的符号来清楚地表达一组动作是怎样划分给几个子进程的。

为信号划分作注解的建议方便读者，同样的建议也适用于任务划分。

D.4.2.6.5 数据

数据关联于进程。当我们划分进程时，我们也划分其数据。SDL中没有指出此种划分的形式的方法。每个子进程需要定义其变量，如有新的同义词和类型，则也要定义。

要注意：划分数据不仅影响到该进程的诸动作（如果某子进程不再能得到某一信息，则应携带该信息的输出就不能发送，根据该信息操作的某个任务就不能实施，等等），而且会影响到其它进程，如果涉及到了出口数据或透露数据的话。

出口进程中的划分情况应通知诸进口进程，使它们能访问新的子进程以取得信息。如果拥有它的子进程被置入的子功能块与视见者所在的子功能块不同的话，一个以前可视见的数据可能不再能被视见到。因此它的值应被进口（或出口）；或者实施显式的信号交换。

D.4.2.7 划分情况下的系统表示

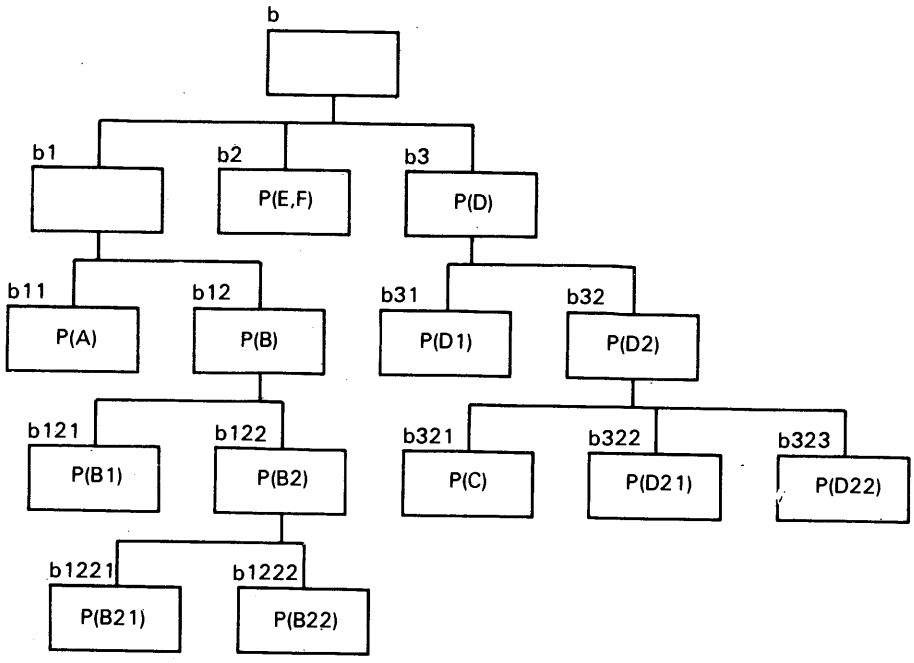
在系统表示为一组由信道互连的功能块的情况下，以及在每个功能块的行为由一个或多个进程来表达的场合，我们可用单一层次来表示。这意味着我们所看到的所有用于表示的元素是处在同一个层次上。当我们进行划分时，我们就在各种文件之间引入了一种层次关系。我们要有一个文件包含系统结构的表示，这里所说的系统是由“n”个功能块组成的。

不同的文件可以把一个系统表示成为是由不同的一组功能块所组成，其中有些功能块是从包含在原先文件中的功能块所导出的（原先文件中的有些功能块已被因功能块划分而得到的子功能块所代替）。后一个文件必须和原先的文件相联系。

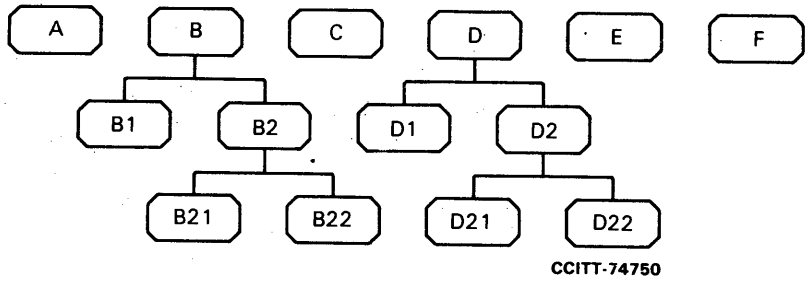
不仅仅是把诸文件彼此联系起来以得到一个完整的系统表示，它们还应按这样一种方法组织起来：即有可能在各个不同层次上来获得系统表示，从总的概览开始，一个层次比一个层次表示得越来越详细，这意味着把各种文件分组，使它们形成不同层次的系统表示。

并非所有的层次都包含相同的元素。在第一层，系统表示可由一些功能块和信道表示所组成，不包括描述每个功能块行为的诸进程。在一较低的层次上，我们可能希望包括某些功能块的行为表示，但却不包括另外一些功能块的行为，甚至还可能希望表示一个功能块的部分行为，但并非全部行为，全部行为只能在一更低的层次上提供。表示的最低层次（更为具体的表示）应包含所有功能块的行为的完整表示，亦即表达这种行为的进程的全集。

这种层次表示法的第一个副作用是我们必须指出进程与功能块的联系。对系统进行划分（按一个功能块来考虑）产生出一种树形结构。对“n”个表示该系统的行为的进程进行划分产生“n”棵树，这些树的每一个进程应属于功能块树中的某一个功能块。此外，我们还可能在一定层次上引入新的进程，并可能从该层次起进一步加以划分。如果对这些新进程进行划分，又要产生树状结构，该结构中的每个进程也应属于功能块树中的一个功能块，如图D—8所示。



a）功能块树



b）进程树

注-功能块b和b1没有相关联的进程。
功能块b2含有2个进程（E，F）。
功能块b321含有进程C，它只出现在该层次上。
（b＝功能块　　P＝进程）

图D－8

将进程分配给功能块

考察功能块树，我们能作出以下几点分析：

首先，每棵树总有一个且只有一个根，这个根就是系统功能块。即使在系统从开头就被表示为由几个功能块所组成的场合，也是如此，在功能块树中，这些功能块将表示在比如层次1。根块可只含有系统名字。对层次1上的诸功能块可给出信道定义，但除非诸功能块有相关联的进程，这一点不是必要的。

第二点考虑来自对树的叶子的考察：它们并非全在同一个层次上。这是由于在树上的功能块划分次数不同。划分的次数取决于几个方面，大多根据制订规格的人（或设计师）的主观判断决定。

SDL只要求仅当诸叶子功能块的行为已完全确定（即所关联的进程定义足以表示其行为时），它们才能停留下来而不被进一步划分，因此一个叶子功能块必须至少含有一个相关联的进程。

当一个系统表示是在几个抽象层次上给出时，我们可选择这些层次中的任何一个来表示该系统。选择某一个层次意味着要考虑该层次上的诸功能块，考虑与它们有关的进程，以及在较高层次上的所有叶功能块和它们所联系的进程（见图D－9）。

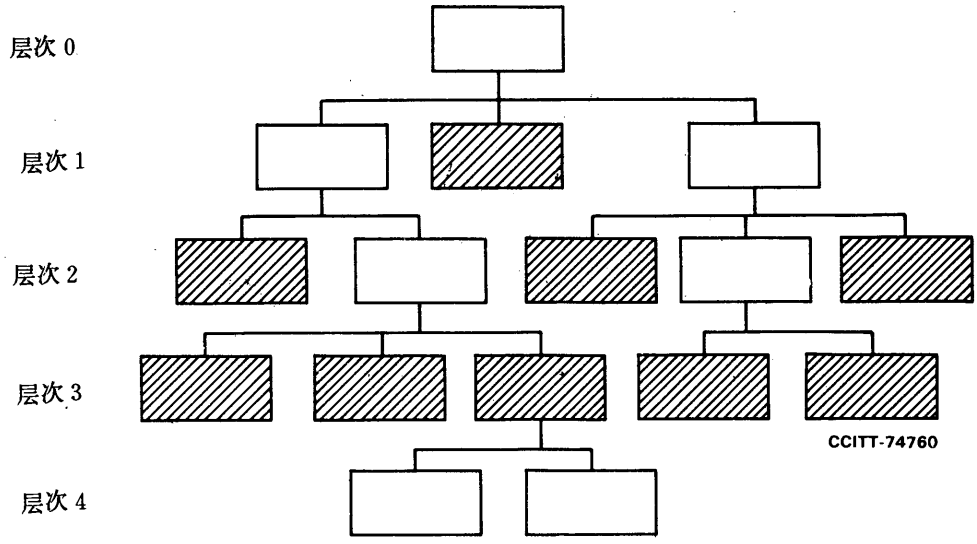


图 D-9

表示层次3

在某一个层次上表示一个系统可能不完全，这是由于在该层次上有的功能块没有所关联的进程，而所关联的那些进程又不能完全地表示其行为。

讨论表示的层次、以及选择某一个表示层次的理由是：

— 我们在设计中可能会达到某种详细程度，它由某个层次来表示（在这种情况下在该层次上的诸功能块都是叶功能块，而它们是不完全的，因为工作还没有做完！）。

— 我们希望以某种详细程度来考察系统表示，因此我们选择一个系统表示的合适层次；它能很好地符合我们所需要的抽象程度。应注意在有些情况下一定的表示层次可由具有不同抽象层次的文件组成。系统的一部分可在层次 2 表示得很详细，而另一部分在层次 4 还很抽象，这意味着当我们选择层次 3 的表示时，我们既有很详细的部分，同时也有只能看成是概览的部分。

— 表示（和设计）的方法如下：每一个层次都具有准确的含义，例如第一层对应于规格，第二层提供整个系统的结构，第三层提供模块结构（机架、软件功能），第四层提供详细的结构（印刷电路板、过程、软件模块）。在这种情况下就可根据某一读者的需要来选择相应的某个层次。同时应注意，该方法将可避免组成某一表示层次的部件在详细的层次上的不一致。

除了总的系统表示和用不同层次给出的系统表示以外，SDL 还有“一致的系统表示”概念（见图D—10）。

我们定义：可把系统的任何表示看成是一个单一层次的表示，这里的所有功能块可以取自系统结构的任何层次，条件是：

— 若一功能块可看成是一个叶功能块(它或者是一个叶功能块,或者连有表示其行为所需要的全部进程)，则可选来作为该一致系统表示的一部分。

— 若一功能块被选中，则从划分其父本功能块所得到的所有功能块都应直接包括在内，或者通过包括它们的子功能块而包括在内。

— 应提供定义信号流动的全部文件，这些信号在连接（该表示中的）各功能块的信道上流动。这可能意味着：根据所选择的划分策略，一旦某个功能块被选取，则其父本亦应被选取，至少是其中定义数据和信号的那些部分。

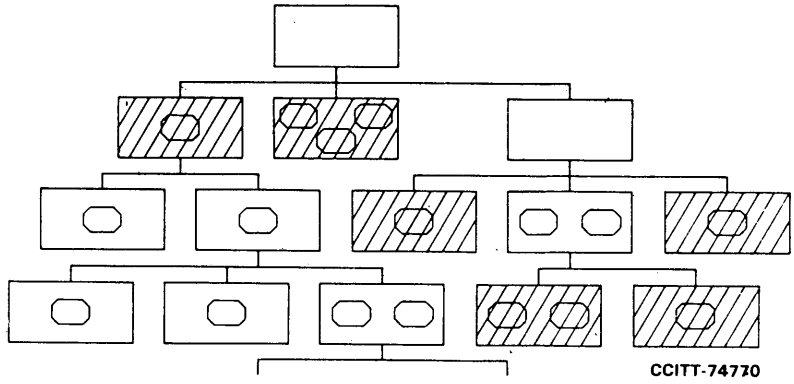
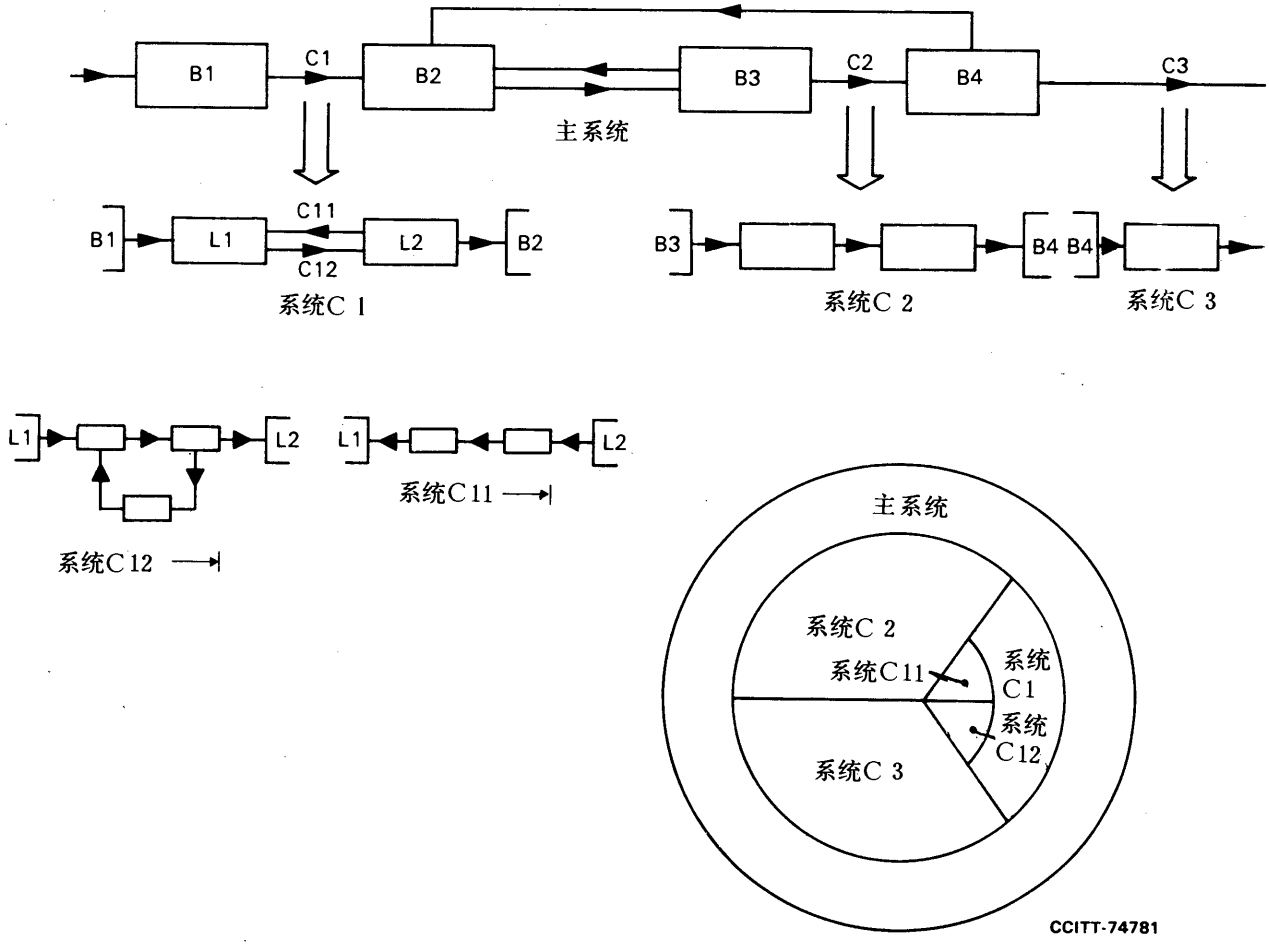


图 D-10

一致性 系统表示示例

在某些信道已被看作是“系统”而加以划分的场合 必须提供这些“系统”的表示。

这种表示具有与一般系统表示相同的文件类型。应加上注解，以使这些系统与主系统相联系。在某种意义上我们可把这些系统看成是对应于“主系统”的内部系统，每个这样的系统可有几个表示层次，而且如果它们所含的某个信道本身也被看成是一个系统，则还可再有内部系统（见图D—11）。



图D—11

把信道划分看作是虚拟的系统表示

D.4.3 SDL概念

这一章将说明SDL的每一个概念，并给出如何使用这些概念的一般准则。大部分例子将采用SDL/GR，同样的准则也适用于SDL/PR。

D.4.3.1 系统

已如前述，SDL是用来塑造系统的，因此系统就是一个SDL规格或描述所定义的东西。这样说来，一个SDL系统可以模拟一个电话系统（或交换局）的一部分，或者模拟电话系统的一个完整网络，或者多个电话交换局的某些部分（例如一条中继线两端的中继线控制器）。关键在于从SDL的观点看来，SDL系统包含有规格或描述所试图定义的任何东西。环境在规格以外，不在SDL中定义。

系统通过信道与环境联接。从理论上说，只需要一条进入信道和一条走出信道来与环境联接，实际上，通常为通到环境的每个逻辑接口规定一条信道。

每个系统由许多用信道连接起来的功能块组成。每个功能块相对于其它功能块而言是独立的。在两个不同功能块中的进程之间，通信的唯一手段是通过信道发送信号。

D.4.3.2 功能块

如在SDL的构成一章中所述，一功能块可包含一个或多个进程结构。功能块的完整定义要求在功能块树图底部的诸功能块含有一个或多个进程定义。

在功能块内部，进程间可通过信号或共享值来互相通信，这样，功能块不仅提供了一种把进程组合起来的合适机构，也提供了数据可见性的边界。由于这个缘故，在定义功能块时应加小心，以保证在一功能块内将进程分群时，是合理的功能上的分群。在大多数情况下，应先把系统（或功能块）分成功能单位，然后再定义功能块里的诸进程。

D.4.3.3 信道

信道是功能块之间进行通信的工具。通常信道是一种功能性实体，可用来表示特定的信息通路。事实上可以通过划分信道，来形式地规定每个信道的行为。

信道定义含有一个信号表，列出能由该信道传递的全部信号。此信号表作为一种手段，用以保证由信道一端某一进程发送的每个信号，都能被位于信道另一端的功能块中的进程所接收。这样，信道定义就成为每个功能块接口定义的一部分。在有多人参加的大工程项目中，就信道中有哪些信号以及这些信号的定义方面及早达成一致，可减少两个进程不能互相通信的概率。

D.4.3.4 进程

进程是一种扩展的有限态自动机，它规定一个系统的动态行为。进程基本上是处于等待信号的状态。当收到一个信号时，进程作出响应，执行特定的动作。应根据该进程所能接收的每个信号类型来规定相应的动作。进程含有许多不同的状态，使它在收到一个信号时能执行不同的动作。这些状态提供了早先已出现过的动作的记忆。根据收到的具体信号作完相应的全部动作以后，就进入一个新的状态，这时进程又等待另一信号。

进程可以在系统创建时就存在，或者作为另一进程发出创建请求的结果而被创建。另外，进程能够一直生存下去，或者能通过执行一个内部停止动作而停止。创建进程的某些重要属性是：

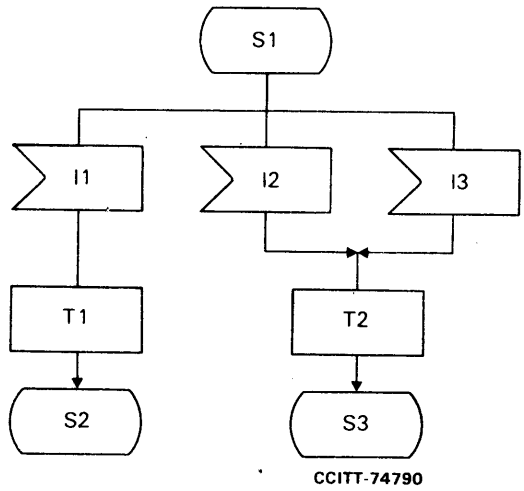
1) 创建系统以后，进程只能由同一功能块内的另一进程来创建。允许在别的功能块里创建进程的一个方法，是让每个功能块里都有一个特殊的进程，当此进程收到来自另一功能块的某个进程的一个信号时，就会创建一个进程。在许多实例中，这个特殊进程是某种“操作系统”进程。

2) 进程一旦被创建，就有它们自己的生存期。进程只有在一次跃迁期间当执行一条停止动作时才消亡。若系统允许外面进程的杀掉（kill ）操作，塑造这类系统的一个方法就是安排一个专门的杀掉信号，当收到此杀掉信号时，进程就执行停止动作。

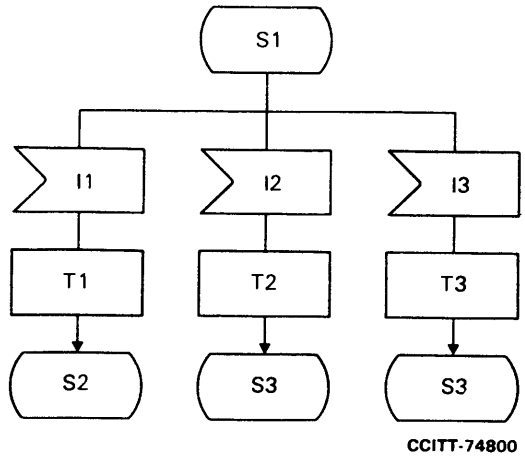
3) 进程被创建后，就知道它自己的标识名及其父亲的标识名，如不给予别的信息，它不能和其它进程通信。有两个方法来解决这一问题：第一，该进程的一个或多个参数含有该进程将与之通信的诸进程的标识名；第二，它能接收一个信号，其中含有一个进程实例标识符作为其实在参数之一。

D.4.3.4.1 状态

一个状态是进程中的一个点，在该处没有动作正在执行，但要监视输入队列，看有没有进入信号到达。根据输入信号中给出的信号标识符，该信号的到达会使进程离开该状态而执行一特定的动作序列。一个已经到达且已引起一次跃迁的信号于是就被“消耗掉”而不再存在。在跃迁期间，进程并不明确地知道跃迁是哪个输入信号引起的，这只能从因果关系来推断（即：这个跃迁只能在接收到某一特定信号时才出现）。在图D—12中，只有收到I 1后才执行任务T 1，但若收到I 2或I 3，都将执行任务T 2，如对T 2来说知道收到的是哪个输入很重要，则按图D—13所示来设计进程较好。



图D—12
执行任务取决于三个信号中的两个，但不取决于收到的是两个中的哪一个



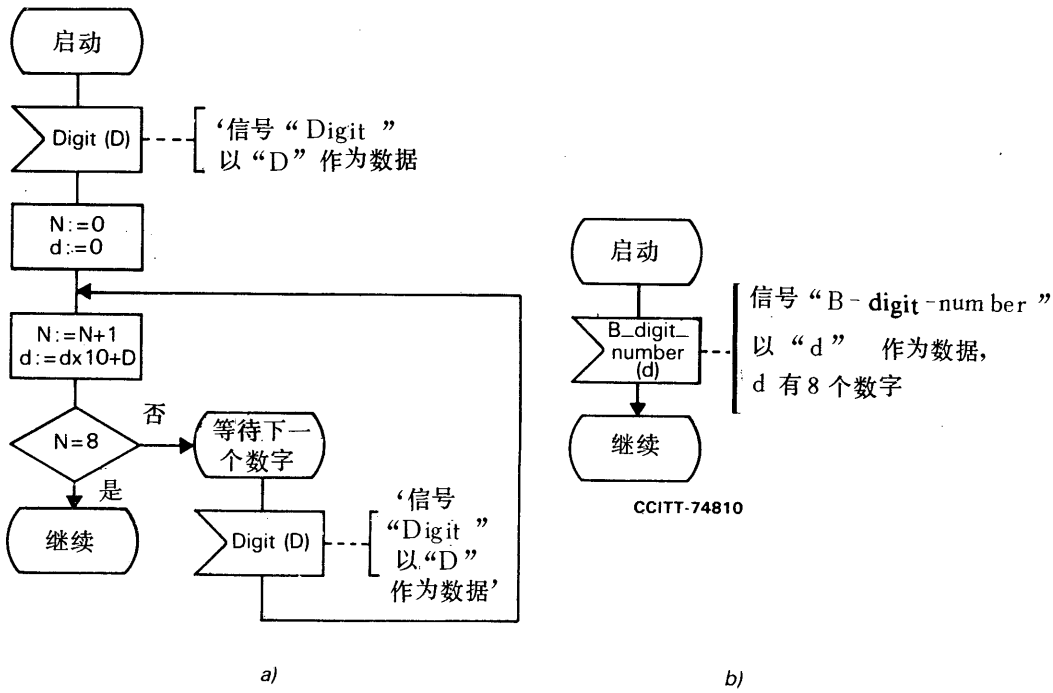
图D—13
执行任务取决于所收到的信号

D.4.3.4.1.1 *决定需要哪些状态*

当定义一个进程的诸状态时，SDL图的作者通常可有一些方法上的灵活性，他可能需要一个能使他确定进程诸状态的方案，这个方案可以是非形式的，也可以是形式化的。为了使所产生的SDL图不至于不必要地太复杂（要确定过多的不同状态），但也还要发挥SDL所固有的优点（不去人为地过分减少状态的数目）；这就需要作出好的判断（通过实践取得）。在作者开始制图之前，必要的准备工作（在§D.4.2中已讨论过）必须已经完成，例如：

— 用功能块来构成系统；
— 每一个功能块用一个或多个SDL进程来表示；
— 选择输入信号和输出信号；
— 进程中对数据的使用。

所有上述因素对决定每个进程的状态都有重大影响。在图D—14中用两个例子来说明“选择”输入信号对SDL图中状态数目的影响。



注-例a）和例b）用不同的详细程度来表示相同的功能，结果是状态数目不同。

图D—14
接收一个8位数字的电话号码

D.4.3.4.1.2 *减少状态数目*

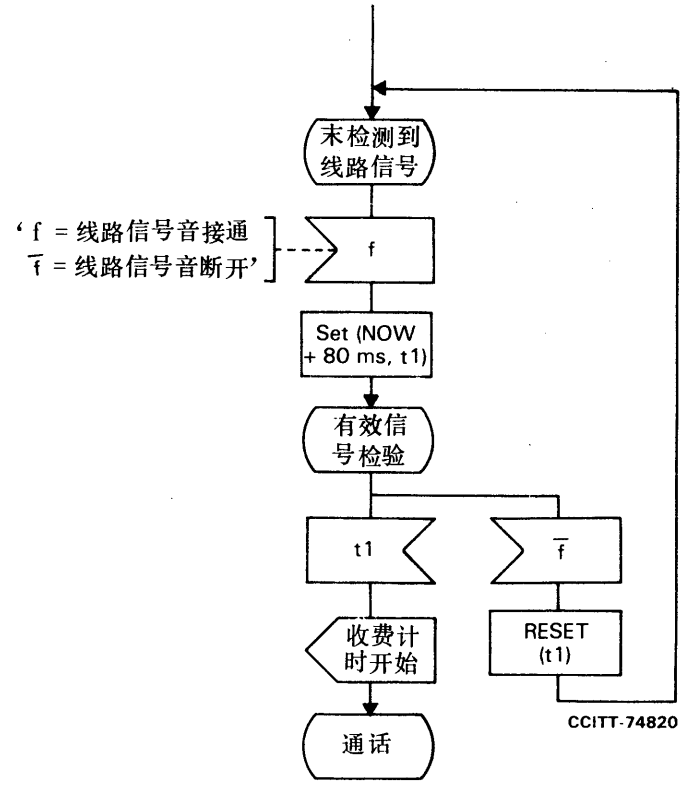
在已采用了一个确定进程诸状态的方案以后，SDL图的作者可能会觉得所用的“状态太多”了。状态的数目很重要，因为一个SDL图的大小和复杂性往往与状态数目密切相关。有若干办法可用来减少状态的数目。但是一个SDL图是复杂的这一事实本身，并不是改变它的理由，因为该SDL图的复杂性可以仅仅是它所确定的进程所固有的复杂性的反映。通常状态集合的选择应能最为清晰地表达进程及其环境间的交互作用序列，这种清晰度通常不是将状态数目减至最少所能取得的。需由一个进程来处理的独立序列的数目对状态数目具有倍增的影响，因此希望用分开的进程来处理独立的序列，因为这样会减少状态的数目，并增加清晰度。

状态的数目可通过把公共功能分离开、把状态归并或采用过程概念等办法来减少；特定的数据结构也可减少状态数目，使用宏有助于实现可替换的不同表示。

D.4.3.4.1.3　分离公共功能

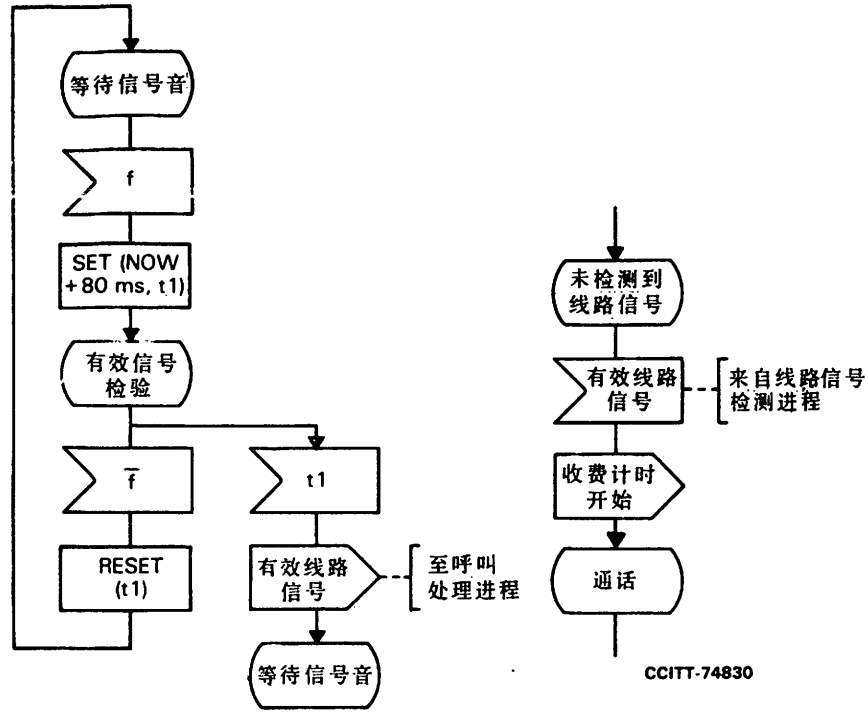
在规划SDL图时，如果要定义一进程的某一特定而又重复出现的特性，会要求表示重复状态，这一点是清楚的。图D—15示出了线路信号进程的SDL图的一部分，它说明了这样的要求，即线路信号音必须在一特定的时间长度内持续出现，才能认为已检测到线路信号。

为了对此作出规定，需要在状态“未检测到线路信号”和状态“通话”之间有一个中间状态。假设在一个完整的图中，这一公共功能在信号受检测的任一点都必须重复，则替代办法就是定义一个独立的进程，它负责在规定的识别时间内监视线路信号音并检测线路信号。有了这个新的进程，就能够把图D—15中的图改画成图D—16所示。(在一给定的因果关系中，可使两个图等效，只要引进一个新的信号“有效线路信号”即可。)



图D—15

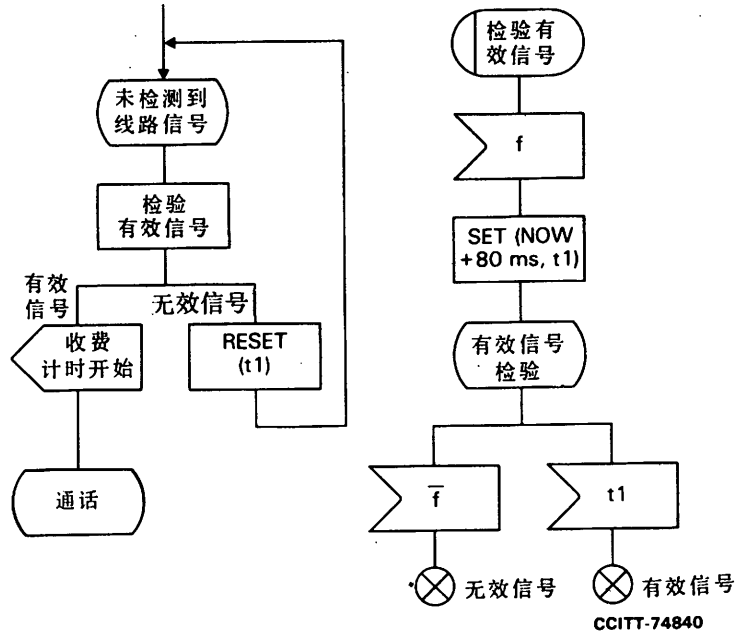
一个SDL图的例子：呼叫处理和线路信号检测组合图



a）线路信号检测进程　　b）呼叫处理进程

图D—16

分离一公共功能（线路信号检测）以避免诸如“有效信号检验”这类重复状态（和图D—15比较）



图D—17

使用宏的图D—16的例子

应注意图D—15中所示的进程和图D—16中的两个进程之间有细小的差别。

图D—15中的进程一收到 t 1就开始收费计时，而呼叫处理进程（图D—16中的进程 b ））要在收到有效线路信号后才开始计时，而有效线路信号是由图D—16中的进程a)在收到 t 1后产生和发送的。这意味着在第二种情况下，计时开始要经过 t 1加信号产生、发送和接收的时间。此外，在信号产生、发送和接收的这段时间里，还可能有其它信号已经排上了队。

应注意的第二种情形是：如果被一子功能接收的信号必须也同样被主功能所接收，那么就不可能把此子功能分离开来，因为一个信号总是只通向一个进程。

若在上例中，同一个信号 f 必须在另一个状态中被接收，这时它不需要 t 1的证实时间，那么也不可能把此证实子功能分离到另一进程中去。

一般说来，如果对诸信号的处理与主进程中的状态无关，那么用分离开的进程的解决办法是有用的。在这种情况下，诸预处理和后处理进程可以处理具体的细节序列，使主进程得以从琐事中解脱出来。这样还往往获得良好的模块性，因为象信号系统这类专门的功能可以从比较面向业务的主进程中分离出来。

这个问题的一种不同解决办法是使用MACRO 符号，如图D—17所示。在这种情况下，我们可以得到比较简洁的图，而丝毫不变动原来的图的语义。此外，如果该进程的逻辑有需要，该MACRO 可从几个状态上被调用。

D.4.3.4.1.4 状态归并

如果在一个SDL图中，两个状态的未来发展是相同的，则不论它们的历史状况如何，均可将其归并成一个，而不影响该图的逻辑。

图D—18示出一SDL图的一部分，它具有两个状态，它们的历史不同，但未来却“完全一样”。在图D—19中，这两个状态已合并成一个状态。这是一个相当平凡的例子，其复杂程度降低不多，但此技术可以用来获得显著的简化效果。SDL语义不允许在某一状态之后跟有判定，来确定该状态以前该进程的历史(在此例中即：所送出的是A 6呢还是B 4？)，除非这个信息在进入该状态以前就已显式地存贮起来。应注意：在一输入中命名数据就会使该值被存贮起来。

一个状态和该进程可能有的诸逻辑状况应有清楚的关系，因此把不同的逻辑状况归并入一个状态是不好的。

当以后要更改一个归并得来的图时要多加小心，用户应研究拟议中的更改是否会以相同的方式影响原来的两个（或多个）流程分支。

D.4.3.4.2 输入

D.4.3.4.2小节说明输入的概念，以及在SDL图中不带保存概念的输入的用法。保存的概念以及SDL图中输入和保存合在一起的用法在§D.4.3.4.3中说明。

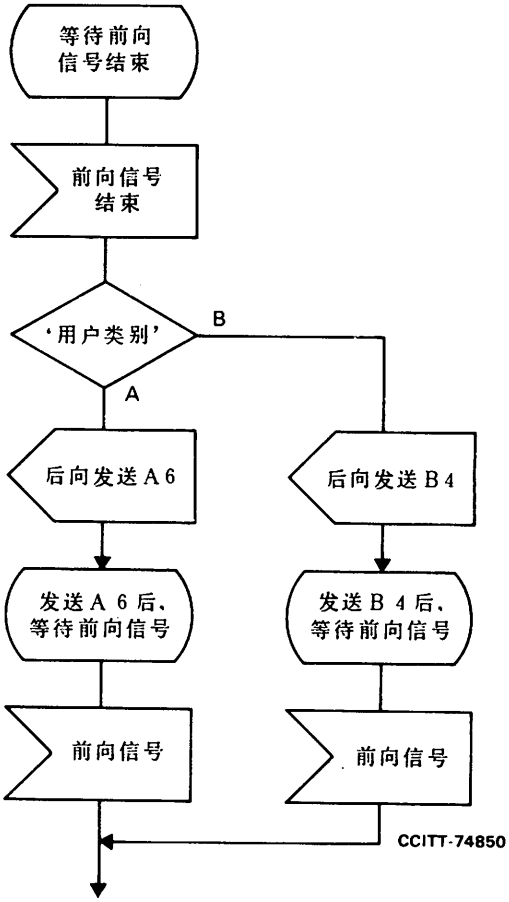
D.4.3.4.2.1 概述

附属于某一状态的一个输入符号表明：若在输入符号中给出信号名的信号到达时此进程正好处于该状态，那么机器就应解释跟在该输入信号后面的跃迁。当一个信号已经触发了一个跃迁的解释，则该信号不再存在，我们说它已被消耗掉了。

信号可有相关联的数据。例如，名字叫做“数字”的一个信号不仅可用来触发接收进程执行一次跃迁，并且也为它带来了数字（0—9）的值，这个数据可由该接收进程使用。

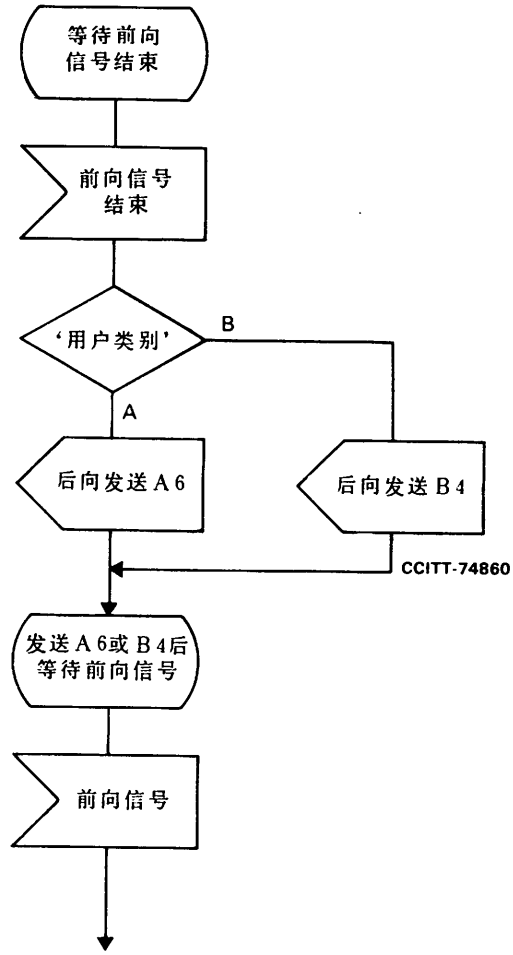
为使这些数据项能被进程所利用，它们必须在输入符号内命名，并括在括号内。这样做就能在跃迁动作期间使用这些数据。如果数据不是显式地存贮起来，它就不能在以后另一次跃迁中被访问。

要用逗号把一个数据项和其它数据项分隔开。从输入中接收数据的例子示于图D—20、D—21和D—22。



图D—18

例：状态归并前某SDL图的一部分



图D—19

例：状态归并后某SDL图的一部分

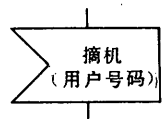
当该输入被机器解释时，接收进程就能利用已命名的数据。

图D—20示出摘机信号的接收。摘机信号具有相关联的数据（用户号码），其值为9269。这个信号可通过三种方法接收，如图中（a）－（c）所示。

图D—22示出如何用一个信号来发送和接收几个数据项，每一项必须用逗号和它后面的项隔开。在图D—22 c)中，我们示出如何通过在数据项表中留空白的办法略去不需要的数据项。

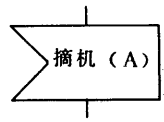
应注意：在输出信号中我们可以写E 1、E 2或E 3的表达式，但在输入信号中我们必须用变量来接收发来的值。

在SDL中，不需要画出输入符号来表示（因其到达而）产生零跃迁的信号（零跃迁即不含动作且导致返回到原来状态的跃迁）。习惯上，在某一状态上不用一显式输入符号表示的任何信号，必存在一个在该状态下的隐式输入符号和零跃迁。根据这个惯例，示于图D—23的两个图在逻辑上是等效的。可任用其一。



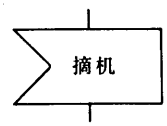
注-值9269存贮在名为“用户号码”的变量中。

a)



注-值9269存贮在名为“A”的变量中。

b)



CCITT-74870

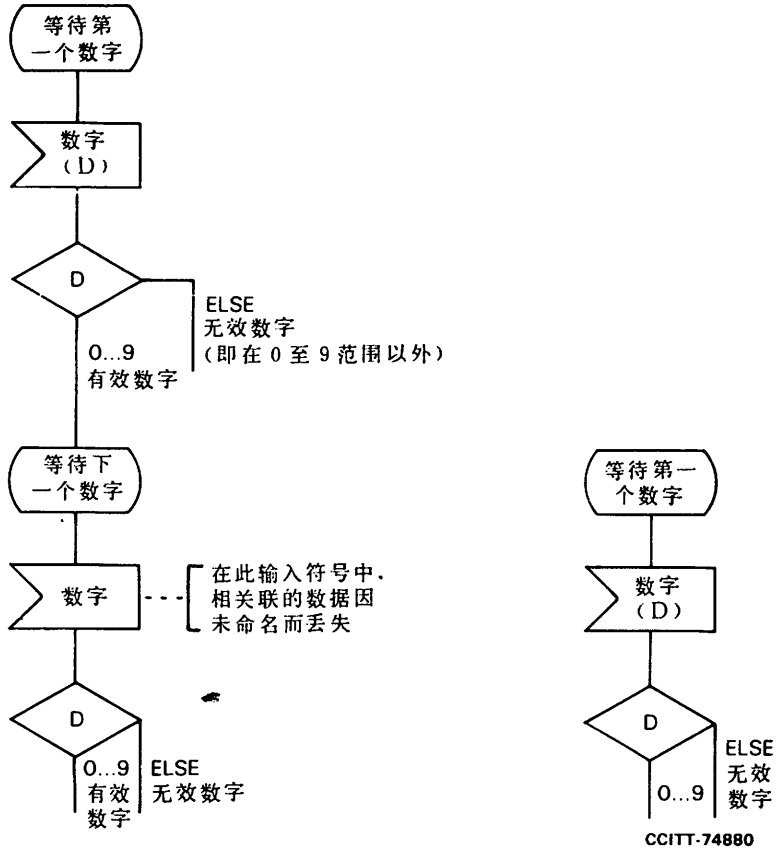
注 1-没有数据名，故值9269丢失，不能为接收进程所利用。

注 2-信号名（摘机）必须与输出的信号名一致，但数据名可以选择，或相一致，或不一致。

c)

图D—20

一个进程中的数据接收举例



a）不正确　　b）正确

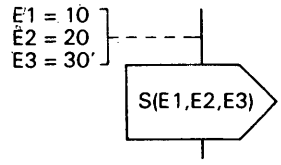
注 1 - 在a）中，第二个判定将使用得到的第一个数字作为D值。

注 2 - 在b）中，D值将是当前数字的值，以前诸数字的值均被冲掉。

注 3 - 由于已声明D是数字的一个形式参数，D的值是自动地被存贮的。

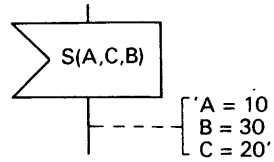
图D—21

一个数字接收进程的一部分



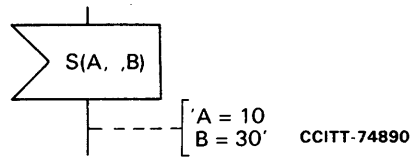
注-输出信号S 有三个变量，称为E 1 、E 2 和E 3 ，这些项各有其值，例如当前为10、20和30。

a)



注-在接收进程中，对应的输入信号S 分别把这些项命名为A 、C 和B 。

b)

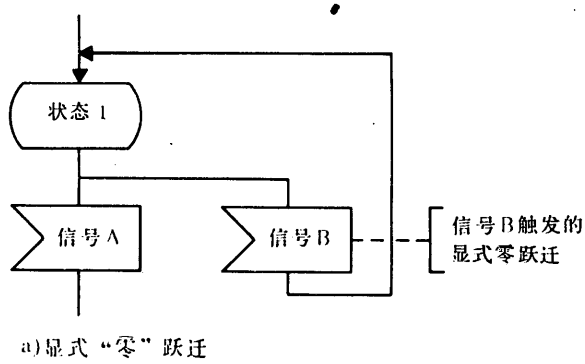


注-这个输入信号只命名两个变量，中间的数据值被丢失。

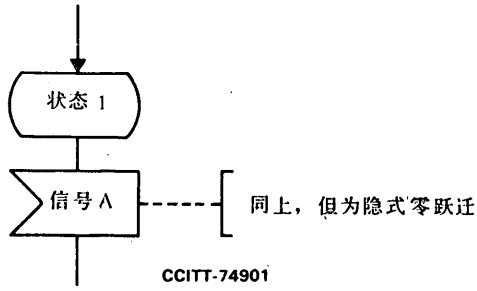
c)

图D—22

一个信号带有几个相关联的数据项



a)显式“零”跃迁



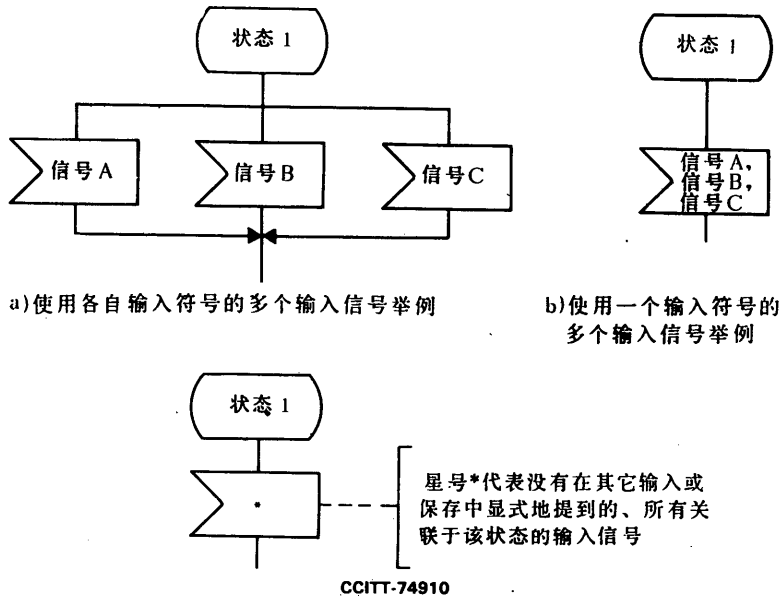
b)隐式“零”跃迁

注-如信号B 有数据关联，则在两种情况下数据均丢失。然而如在情况a ）中示出数据名（如：信号B(x)），则数据值将保留。在这种情况下这两种情况不再完全等同。

图D—23

“零”跃迁的显式表示和隐式表示

在多个输入导致同一跃迁的场合，所有有关的信号名字均可置于一个输入符号内。图D—24示明了这一点，其中的几个图在逻辑上都是等效的。如在一个输入符号中提到所有的信号名，它们必须用逗号隔开。



a)使用各自输入符号的多个输入信号举例

b)使用一个输入符号的多个输入信号举例

c)使用星号符号的多个输入信号举例

图D—24

多个输入的各种表示法

D.4.3.4.2.2　隐式排队机制

当一个进程到达一新状态时，可以有一个或多个信号正等待着被消耗。这意味着诸信号必须以某种方式排队，否则它们将被丢失。当一个信号抵达其目的功能块时，它就被交给接收进程的输入队列。SDL语义为每个进程定义了一种先进先出原则的排队机制，在这里，信号是按其到达进程的先后次序被该进程所吸收(消耗)的。当进程进到某一状态时，从队列中交给它一个且只交给它一个信号，这意味着若该队列不是空的，该进程就消耗队列中的第一个信号；若队列是空的，进程就在该状态上等待，直到有信号到达队列，之后信号就被该进程所消耗。

图D—25用这种队列概念来说明所画的SDL进程的操作，其中跃迁时间不为零。应注意：

— 没有用保存概念，因此诸信号按照它们到达的次序被消耗掉。

— 信号到达的顺序很重要。如果信号“C”在状态1和状态2间的跃迁期间先于“B”到达，则状态序列将成为1.2.3.，而不是1.2.4。

— 因为当进程进到状态2和状态4时队列都不空，该进程在这两个状态上都不等待。

— 不可能为一信号指定优先权。

若跃迁时间为零，则每一信号在其到达进程的当时就被消耗，除非采用保存操作（§D.4.3.4.3)。

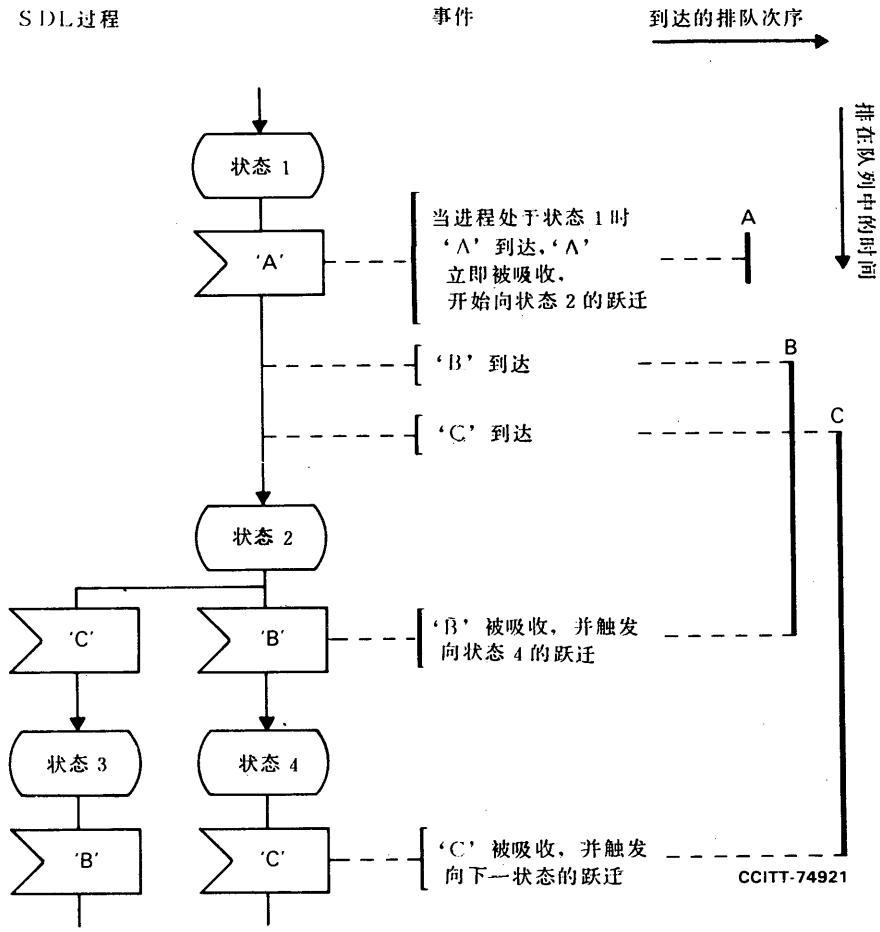


图 D—25
隐式排队机制的操作举例

D.4.3.4.2.3　非正常出现的信号的接收

在每一状态中，所有可能的信号都必须隐式地或显式地示出。但在几乎所有的状态中都会出现例外（并不预期在该处出现的但理论上可能出现的信号、未定义的信号或逻辑上错误的信号等）。通常作者不指出这类可能性，其后果是如有这种信号到达就被丢弃；但如作者要在其图中包括这些例外，则所有的状态都要增加一个额外的输入。

另一种可能性是采用一个状态的多次出现（见§D.6.3.6.5.1）和“全部”符号（*）（§D.6.3.6.2.1）。例如，如果要使信号A—ON-HOOK（A挂机）能在所有状态上被接收并且产生完全相同的动作，则可采用图D—26中所示的方法。

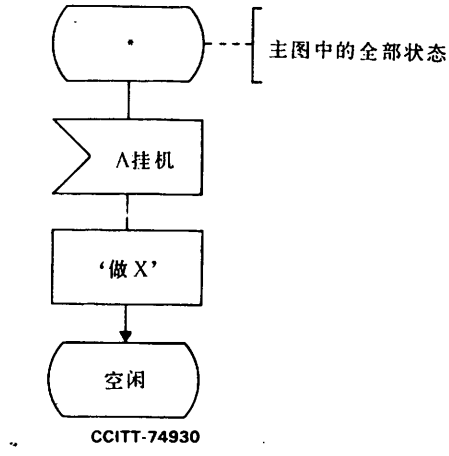
D.4.3.4.2.4　信号的同时到达

建议Z.101中包含有信号能同时到达一进程的可能性，并且建议中说可以任意安排它们的次序。

如果用户设计一个能得到同时到达的信号的进程，应小心不使到达的次序安排打乱该进程的预期操作。

SDL不建议信号的优先权，即：信号的同时到达意味着随便选用其中之一。

当进程进入某个状态时如可得到几个信号，则只向该进程提供一个信号，并因而把它看成是一个输入。SDL语义含有保持其它信号的意思。



图D－26
处理在几个状态上出现的信号的例子

D.4.3.4.2.5　外部输入与内部输入的比较

外部信号是不同功能块的进程之间的信号；内部信号是同一功能块中诸进程之间的信号。从语义上讲外部信号和内部信号之间没有什么不同。但在以前的SDL建议中，语法上曾规定过区别，这些语法上的不同已被去掉，现在外部信号和内部信号具有相同的语法。早先的关于内部信号的GR语法（符号中的两条垂直线）已不再使用。采用内部信号的老图还允许使用，但机器对内部信号的解释方法和外部信号完全相同。

D.6.3.4.2.6　发送者标识

每个信号带有其发送进程的进程实例标识符。当信号被接收时，一个称为SENDER （发送者）的进程变量获得该信号中所含的发送进程的进程实例标识符的值。图D—27示出使用这种方法的一个例子。

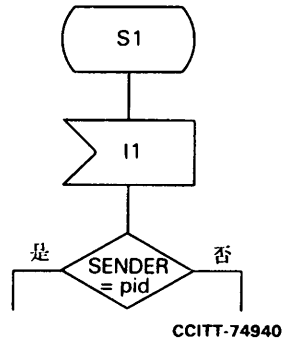


图　D—27
SENDER变量的用法

D.4.3.4.3 保存

保存的概念使得信号的吸收可以延迟，直到随后到达的一个或多个其它信号已被消耗为止。在§D.4.3.4.2中已讨论过：除非使用保存的概念，信号是按它们到达的先后次序消耗的。

在一些信号到达的先后次序不甚重要、而且实际到达的次序是不确定的场合，可用保存的概念来简化进程。

在每个状态上，每个信号可按下列方式之一来处理：

— 把它表示为一输入符号或；

— 把它表示为一保存符号或；

— 包含在一隐式输入信号中，可以导致一次隐式的零跃迁。

在§D.4.3.4.2中介绍的、隐式排队机制的操作，也可应用于保存的概念。信号一到达就进入队列，且当进程到达一个状态时，队列中的信号按它们到达的先后次序每次一个地受到检查。包含在显式或隐式输入符号中的信号被吸收（消耗）了，并且相应的跃迁被执行。在保存符号中显示的信号则不被消耗，仍留在队列中原来的序列位置上，而考虑队列中的下一个信号。

图D—28示出一个带有保存符号的SDL进程例子。应注意信号S和R被消耗的次序是先R后S——和它们被接收的次序相反。一个保存符号能对信号进行保存，只限于进程处于（保存符号所在的）这个状态期间，并且保存到跃迁到下一状态为止。在下一状态，该信号将通过一显式的或隐式的输入被消耗掉（如图D—28所示），除非标有该信号名的保存符号再次出现，或者在该隐式队列中，在它之前碰巧有另一个保存的信号要被消耗（如图D—29所示）。

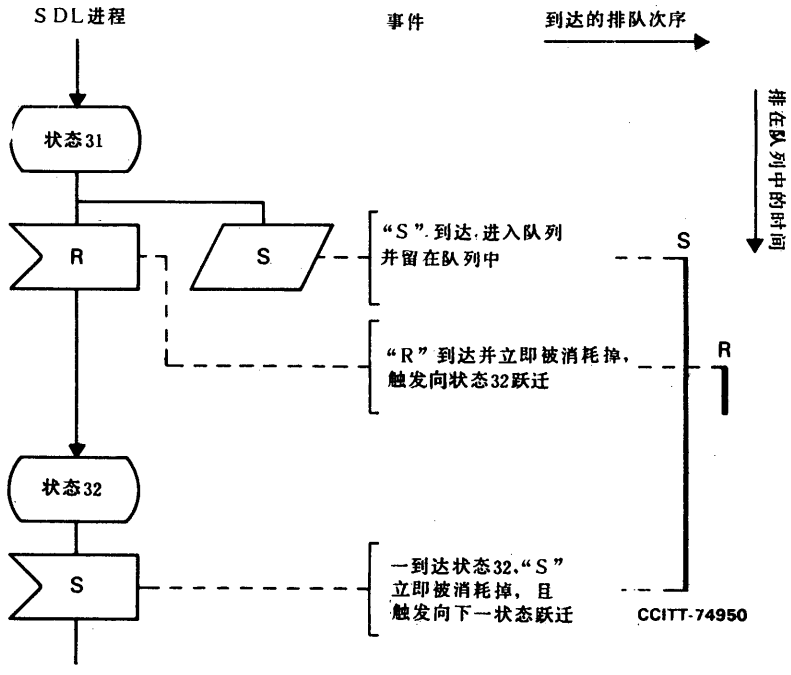


图 D—28

一个带有保存符号的SDL图的例子，示出隐式排队机制的操作

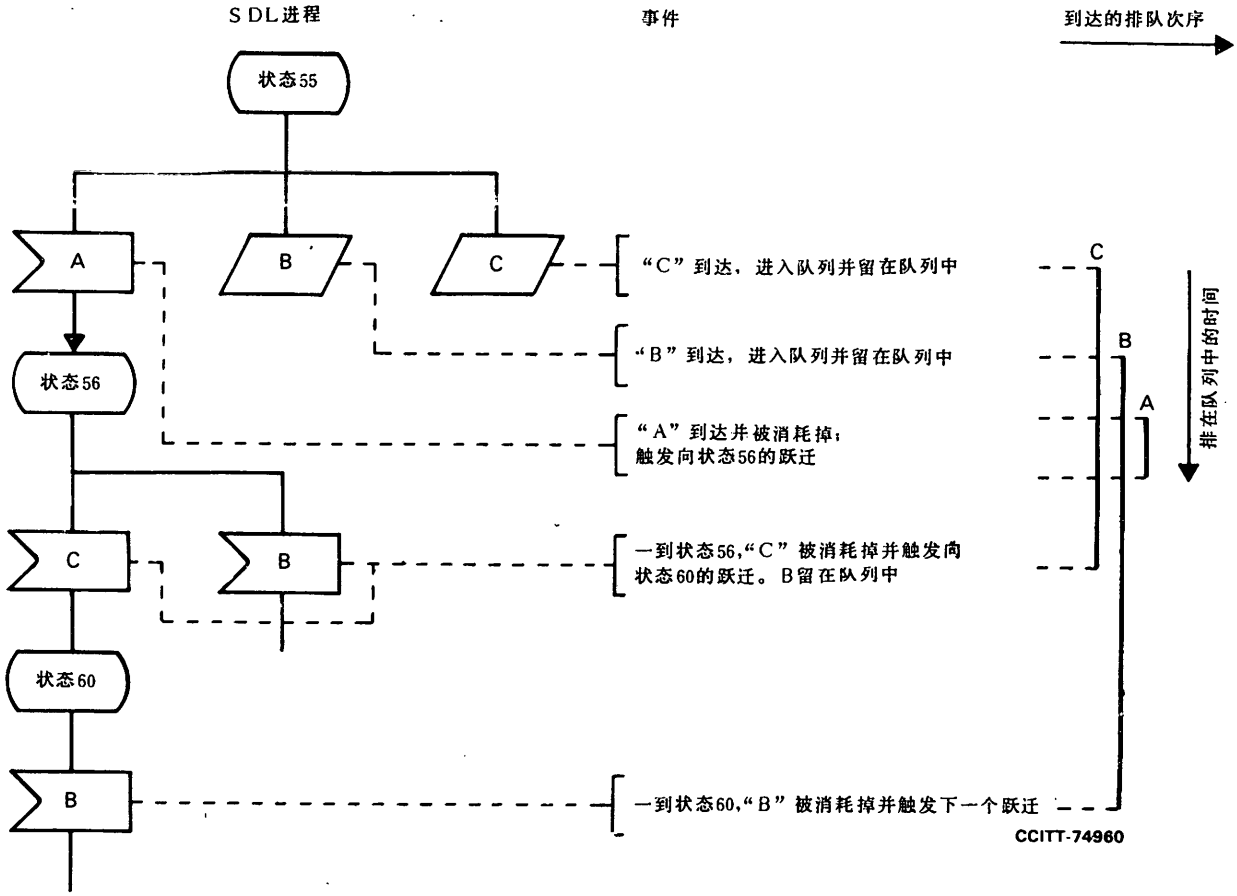


图 D—29

使用保存符号的第二个例子

保存信号只有通过相应的输入符号（显式的或隐式的）才能对进程生效。特别是，在保存信号没有被识别为一输入之前，在判定中不能提出关于它的问题，也不能使用它所关联的数据。

在某个状态上如果要保存多于一个的信号，可以给每个信号一个保存符号，或者把所有信号都写在一个保存符号中。

如要保存多个信号，保存符号的语义隐含了要保留它们到达的先后次序。

在图D—30中给出了使用保存的第三个例子，在图D—31中描述了其概念性的排队机制的操作过程。

保存能用来使图简化。例如，通过对一个信号进行保存，就可以避免接收它以及不得不存贮其数据直到下一个状态。

虽然保存能用于描述的每一个层次，但在较低的层次上，可能有必要描述实现保存概念的实在机制。

如果保存用得不小心，保存的信号队列会变得很长，或者会使记忆信号的时间拖得太长，以致于在需要吸收一个新信号的地方去吸收一个旧的信号。

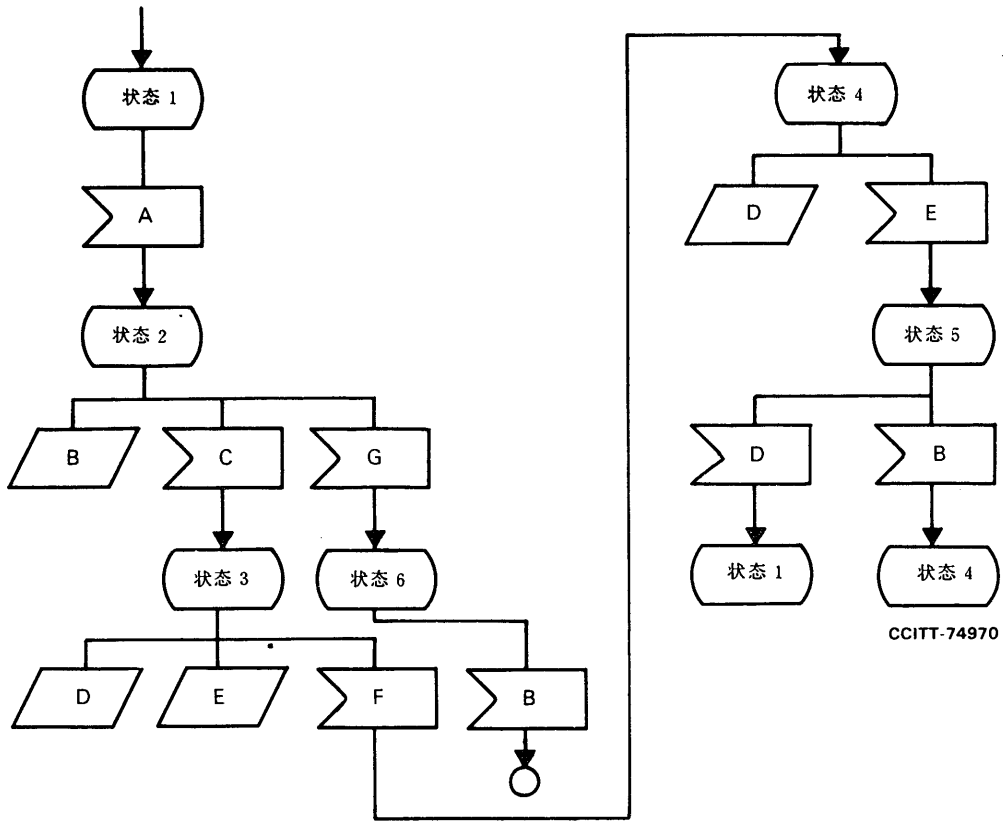


图 D—30

一个较复杂的SDL 图的例子，它具有许多输入和保存

当前状态	事件
状态 1	（进程到达状态1，队列中有信号“A” 、“B”、“C”、“D”、“E”） 队列中的第一个信号“A” 被消耗掉（显式输入），触发向状态 2 跃迁。
状态 2	队列中的第一个信号“B”出现在保存符号中，故留在队列里。
状态 2	第二个信号“C”被消耗掉（显式输入），触发向状态3 跃迁。
状态 3	队列中的第一个信号“B”被消耗掉（隐式输入），并触发零跃迁。
	“F”到达并进入队列。
状态 3	（在再次到达状态3 时） 队列中的第一个信号“D”出现在保存符号中，故留在队列里。
状态 3	第二个信号“E”出现在保存符号中，故留在队列里。
状态 3	第三个信号“F”被消耗掉（显式输入），触发向状态 4 跃迁。
状态 4	队列中的第一个信号“D”出现在保存符号中，故留在队列里。
状态 4	第二个信号“E”被消耗掉（显式输入）触发向状态 5 跃迁。
状态 5	队列中的第一个（也是唯一的一个信号“D”被消耗掉（显式输入），触发向状态 1 跃迁。

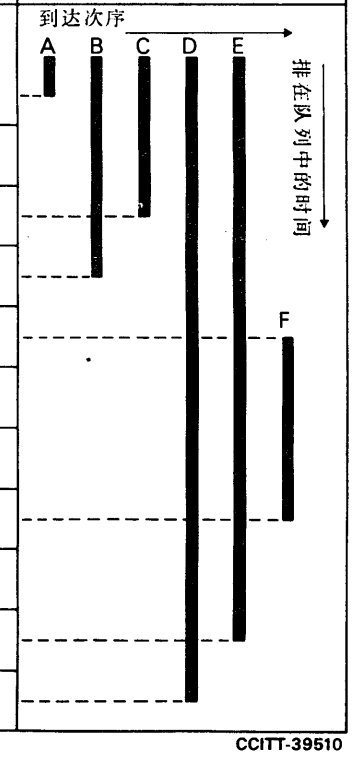


图 D—31

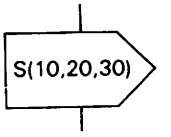
概念性排队机制的操作过程

D.4.3.4.4 输出

输出符号表示将一个信号从一个进程发送到另一进程。由于对信号接收和消耗的控制配属于接收进程（见§D.4.3.4.2），与输出符号直接有关的语义比较简单。从输出进程的观点看来，一个输出常可被看作一次瞬时动作，动作一旦完成，就对该发送进程不再进一步产生直接影响，该进程也不直接地知道该信号的命运。

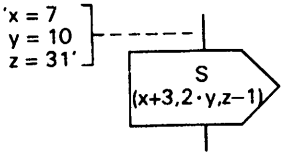
若作者难以决定是否应将某一动作表示为一个输出或表示为一个任务，则应参考§D.4.3.4.6。输出动作表示对一信号及其关联数据值（如有的话）进行发送。将数据值关联于一输出信号，可通过将值放在括号内，或将具有值的表达式放在括号内（见图D—32和图D—22）。

每一输出必须指向一具体的进程实例。由于在产生一规格或一描述之初通常不可能知道该进程实例的标识符，信号寻址的正常方法是在符号或关键字TO中采用一个变量或表达式。图D—33、D—34和D—35给出了几个例子。在图D—33中，在进程创建时把一进程实例标识符的值赋给进程参数 output - to，然后在该进程内部把 output-to 作为这个进程和所连进程之间的链接。在进行系统设计时，应小心地保证由output-to 指示的进程类型能接收发送的信号。在图D—34中，用内部进程变量来把一个信号送回给发送该信号的那个进程。在图D—35中，把信号发送给本进程最近创建的子孙进程。



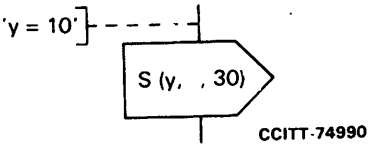
注-信号S 带有三个关联值，10、20和30。

a)



注-在解释S 时，x、y和z（在本例中）分别具有值7、10、31。S 发送值10、20、30。

b)



注-在解释S 时，y（在本例中）具有 值10，S 发送值10、一个未定义值和30。

c)

图 D—32

带有关联数据的输出信号

PROCESS X; (output_to PID);

OUTPUT SIGNAL TO output_to;

图 D—33
用形式参数进行信号发送寻址

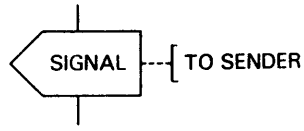
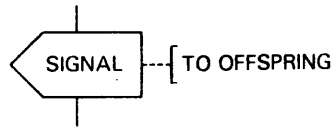


图 D—34
把信号发送回SENDER



CCITT-75000

图 D—35
把信号发送到一个子孙进程

D.4.3.4.5 允许条件和连续信号

允许条件根据规定的允许条件有条件地接收信号。若条件为真则接收信号，且跃迁得到解释。若条件为假，则信号被保存，进程仍停留在原来状态，直到另一信号到达或条件由假变成真为止。这可由图D—36的例子来说明：进程在状态S1启动，而在另一进程P2中，X的值是9。当信号B到达时，X的值和10相比较。应注意：由于这是一个在别的进程里的值，故要求作进口操作以确定其当前值。由于X不为10，进程停留在状态S1，直到收到另一信号或P2送出一个新的X值。在本例中，A到达并引起向S 2的跃迁。在该跃迁期间X变为11，因此现在，在状态S2，附于信号B的条件为真。由于信号B是队列中的第一个信号，故执行在它后面的跃迁，而进程在状态S3结束。

允许条件的一些重要属性是：

1） 只有当进程到达一个状态时才检查允许条件。因此，如果在收到A 后执行跃迁时，X 的值已从9 变到11再变到12，则进程将仍停留在状态 S 2 。

2） 允许条件只能基于含有IMPORTED变量或含有局部变量的表达式，而不是含有VIEWED变量的表达式。其原因是由实现该允许条件的基础机制所决定的。允许条件利用信号来确定在允许条件中的一个值何时已经改变。对于局部变量，是在进入状态之前检查该条件。由于值对进程来说是局部的，当进程处于该状态时它们不能改变，因此不需要连续地监视该条件。对于IMPORTED变量，SDL 利用信号改变IMPORTED变量来作为触发器，以便知道何时检查允许条件。VIEWED变量直接访问该变量的值，由于没有信号发送，一旦进程进入一个状态，它将没有办法来检查该变量的值。

3） 虽然对每个状态可能使用一个以上的允许条件，但对同一个信号却不允许用多于一个的允许条件。因此，图D—37中所示的条件是不许可的。若对一给定信号要求有多个条件，它们可组合成一个布尔表达式，如图D—38所示，这样当两个或多个条件同时为真时就不会出现多义性。在图D—38中，用户已决定，X 的条件比Y 的条件优先级高，因此若两者皆为真，将执行路径 C1 。这里的理由还是因为SDL 的基础系统不对信号附加优先权。

连续信号的基本性质和允许条件相同，只是它不附带信号。因此当进入状态而在队列中没有能引起跃迁的信号时，这些连续信号将受到检查，如果有一个信号为真，就执行跟随它的跃迁。这可由图D—39的例子来说明。起初进程处于状态 S1，X 的值为9，当信号A 到达时，它引起向 S 2 的跃迁；在跃迁期间，X 变到11，由于队列中没有其它信号，所以执行向S 3 的跃迁。

连续信号的一些重要属性是：

1） 和允许条件一样，条件的值只在到达一个状态后才受检查；

2） 同样，在连续信号中只能用IMPORTED变量和局部变量；

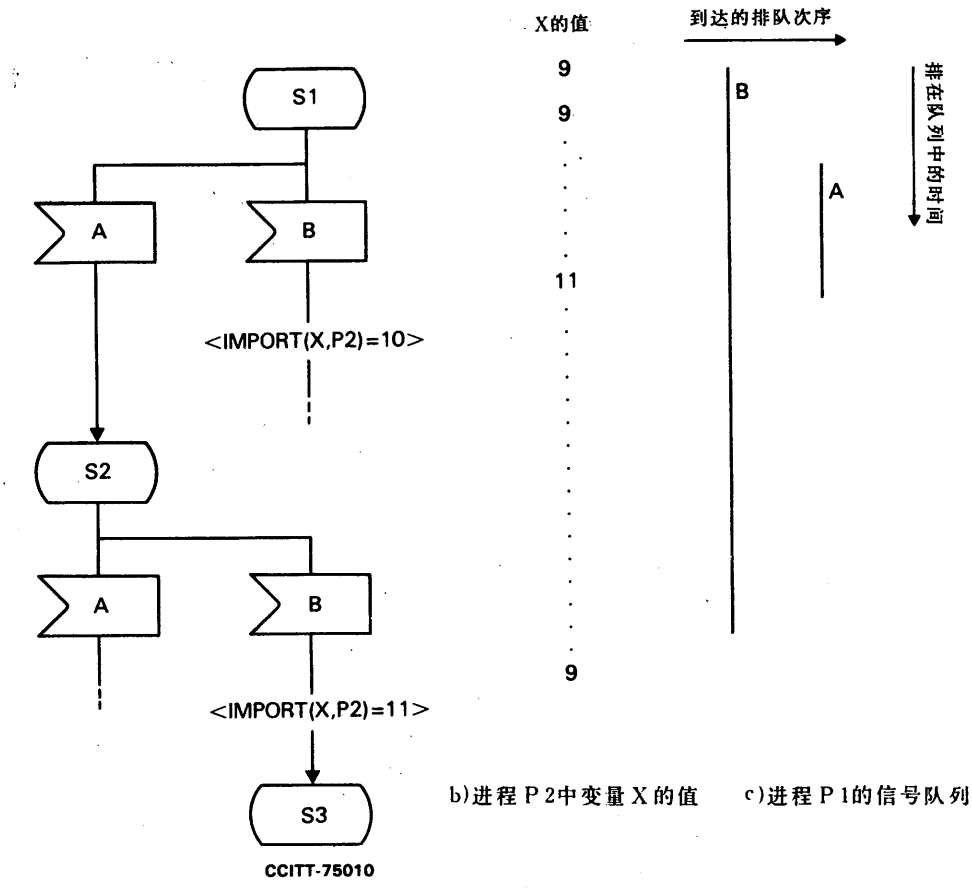
3） 每个状态允许有多个连续信号。当一状态附有多个连续信号时，优先数最小的连续信号最先受检查。不能有两个连续信号具有相同的优先数。在所有情况下，连续信号的优先权低于任何其它信号。在SDL 中，唯一使用优先权的地方就是在连续信号上使用优先权，这又是由SDL 的基础系统所引起的；不过在SDL 中，是用进口变量时所发送的信号来作为连续信号，这一方法允许连续信号采用优先权，而且事实上当出现多个连续信号时，为了避免混乱也需要有优先权。这在图D—40中作了说明。起初进程处于状态 S1，其局部变量X 的值为10，Y 为11，由于两个连续信号皆为真，具有较高优先级（较小的优先数）者被选中，从而执行向 S2 的跃迁；在 S2 上，基于Y的条件不再为真，所以即使X的连续信号优先权低于Y的，但仍执行它后面的跃迁，而进程到达状态S 3 。

D.4.3.4.6 任务

任务用来表示在跃迁中所执行的对数据的操作，但判定和输出信号的产生除外。数据只能被拥有该数据的进程来改变。在任一个图中出现的任务，其性质取决于所描述的进程的性质以及所要求的详细程度。任务所包括的动作例子有：

a） 硬件动作，诸如发送忙音等；

b） 数据运算。



a)进程 P 1的一部分　　b)进程 P 2中变量 X 的值　　c)进程 P 1的信号队列

图 D—36
允 许 条 件

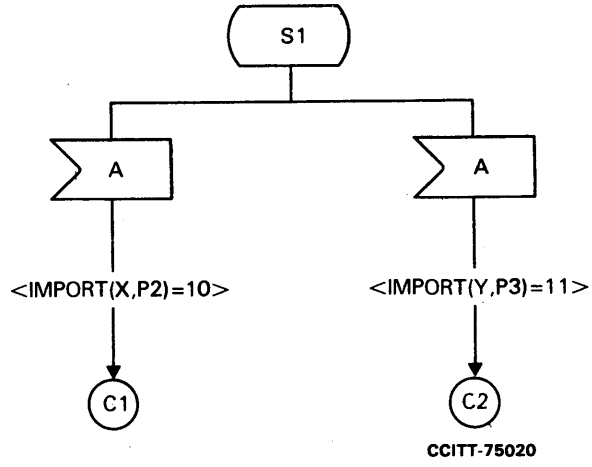


图 D—37
错误的允许条件

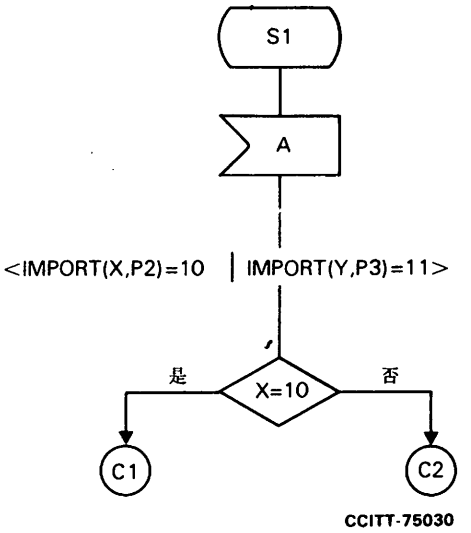
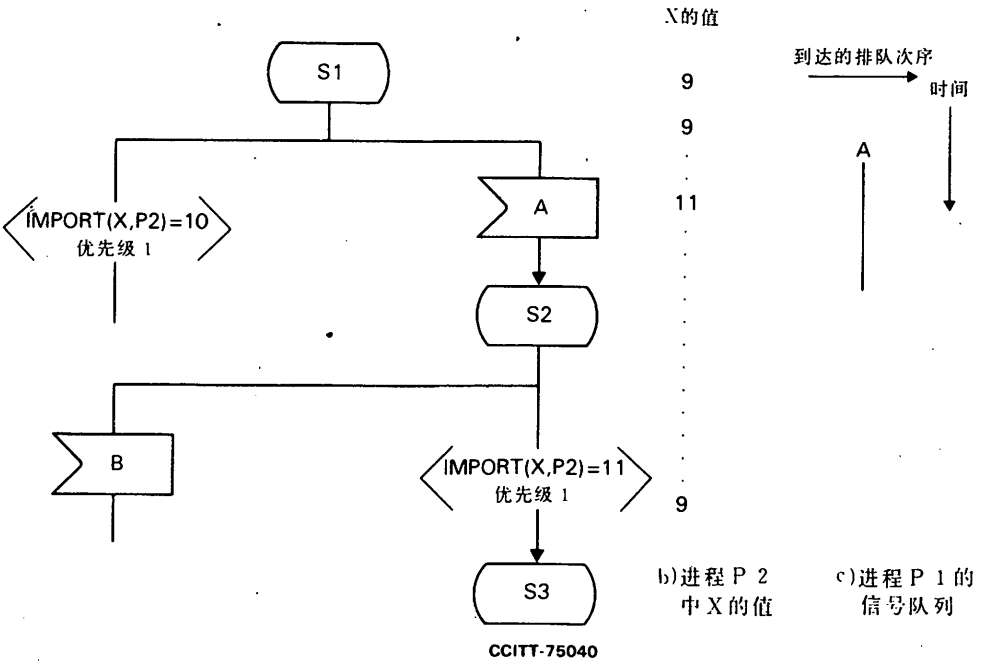


图 D—38

图D—37的正确解答



a)进程 P 1 的一部分

b)进程 P 2 中 X 的值

c)进程 P 1 的信号队列

图 D—39

连续信号

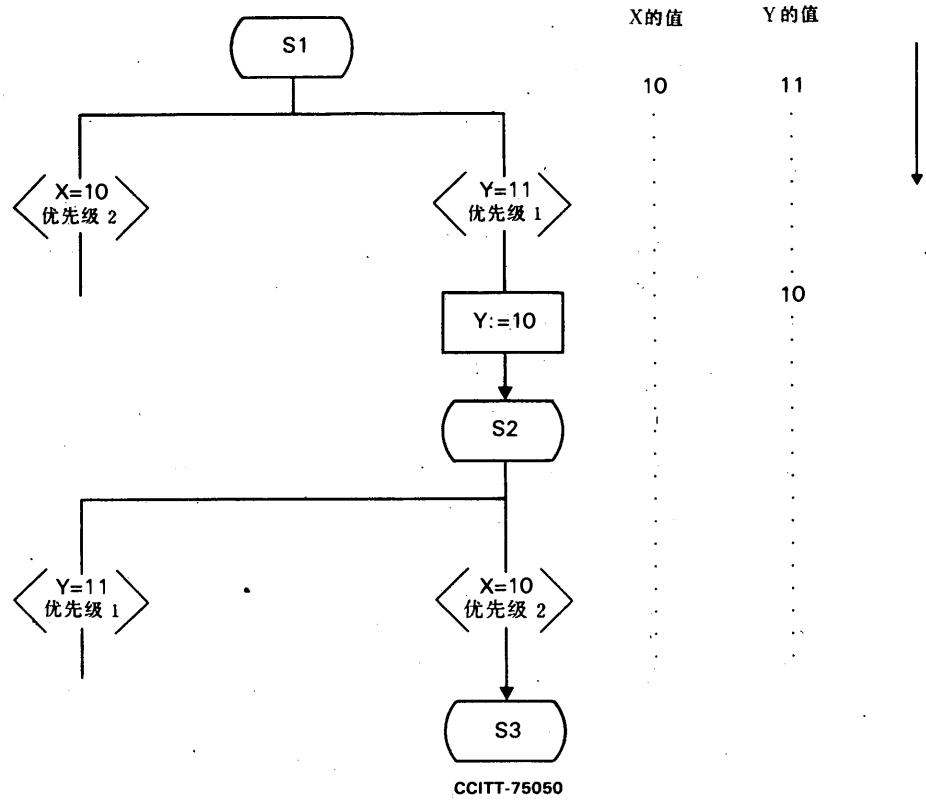


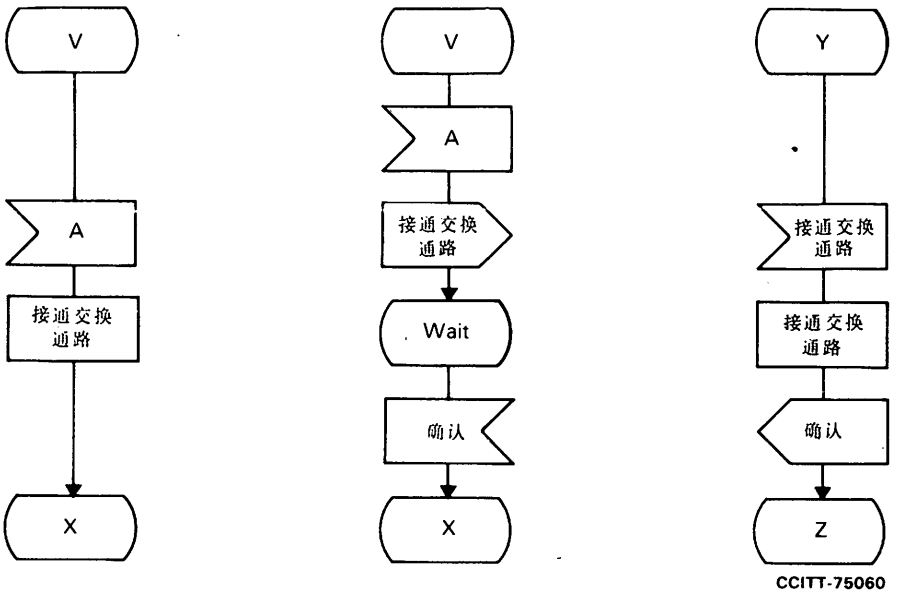
图 D—40

带优先级的连续信号

SDL 用户们在决定用一个任务还是用一个独立的进程来表示所定义的系统的某些方面时，有时会感到困难。考虑图D—41中所示的进程，“接通交换通路”应表示为一个任务或是一个独立的进程呢？如果尚未确定一个单独的交换通路控制进程，则任务符号将是适宜的（见图D—41*a*））；若已确定一个单独的交换通路控制进程，则必须用与该控制进程通信的信号（见图D—41*b*））。

D.4.3.4.7 判定

判定是在跃迁内部的动作，它提出一个问题，涉及进程可用的数据项（在执行该动作的瞬间）的值。根据对问题的回答，进程沿着判定后面所跟的两条（或多条）路径中的一条前进。SDL 图的作者们应保证诸进程是这样确定的：即它们不至于执行得不到回答（或数据）的判定。这种判定将使该图无效，并会引起相当大的混乱。

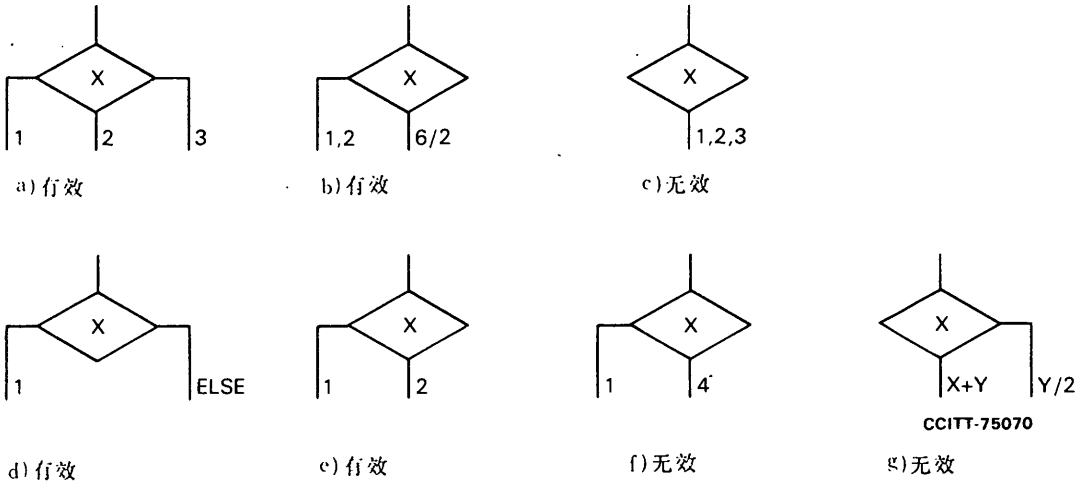


a)用一个进程的解　　b)用两个进程的解

图　D—41

关于“接通交换通路”的两个解

使用判定的若干例子示于图D—42。例中变量X 的值可为1 、 2 或3 。



a)有效　　b)有效　　c)无效

d)有效　　e)有效　　f)无效　　g)无效

图　D—42

使用判定的例子

方案c）是无效的，因为该判定后面未跟有至少两条路径。方案e）有效，因为在SDL中有条约定:若路径的选择不可判定，它将被解释为一隐含的路径直接引向一停止符号。方案f）无效，因为路径之一超出了范围。方案g）无效，因为选择路径的条件含有变量，因此不可能静态地算出结果。

如一个回答在同一跃迁中返回到该判定，则中途必须执行若干会影响判定中的问题的动作。然而,即使用了这条规则，还可能形成死锁，如图D—43所示。因此当有的回答在同一跃迁中返回到一判定时，总要十分谨慎。

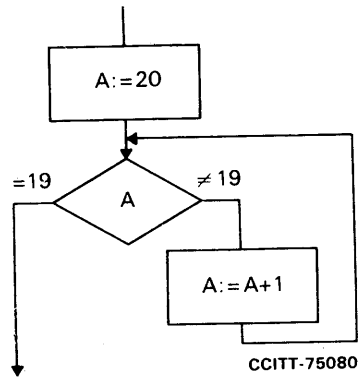


图 D—43

合法使用判定的例子，该判定形成了死锁

可使用进程当前可用的任何值来作出判定，包括:

— 由一个任务存入的数据;

— 通过一次输入接收到的数据;

— 进程创建时的数据集合（作为一实在参数传递给进程的数据）;

— 共享数据。

在判定时提的问题可为询问一表达式的值，该表达式可包括常量和任何上述各类数据值。

D.4.3.5 过程

过程是一种技术，用以向进程增加结构SDL中的过程在许多方面和编程语言中的过程相类似。但应告知用户，它们之间有一些重要的不同，应予注意。记住了这些，就可开始讨论过程。

从语法上讲，过程和进程非常相似。过程流图有一个过程启动节点，它和进程启动节点稍有不同，过程还有一个返回节点，来代替停止节点。从语义上讲，它和进程很不相同，尽管完全相同的进程节点和过程节点可以有相同的含义。

在进程或过程流图中，凡是允许有任务节点的地方都可出现过程调用。在某种意义上讲，过程可作为任务来解释，但下述几点例外：

1） 过程可含有状态，因此可接收信号。所以过程定义的一部分要提供一“附加保存组”，它列出过程被调用时所要保存的一些信号。由于过程可以在系统层上定义，所以重要的是在调用该过程的每个进程中列出全部必须加以保存的信号。

2） 过程可发送输出信号。始发的进程实例标识符是已调用该过程的进程的进程实例标识符。

3） 过程有它自己的局部变量，而不去访问任何进程变量，除非该变量作为IN/OUT参数传递给过程。

4） 为了使过程能在多处被调用，允许使用SIGNAL参数来为主调进程的诸实在信号名和该过程中所用的诸信号名建立同义词。

当一个过程被调用时，就要创建过程环境，并且机器开始对该过程进行解释。对过程的解释继续进行，直到到达返回节点为止。当过程正在被解释时，所有传送给该进程的信号或者被保存，或者被该过程接收。过程没有自己的输入队列，而是使用调用它的进程的输入队列。这就是为什么对过程来说具有适当的“附加保存组”是很重要的原因。附加的保存组是当过程被调用时动态地构成的，它包括了主调过程中不作为调用参数给出的全部有效输入信号。如果信号不在该组中，也不在过程流图中的状态节点的常规保存信号组中，则该信号将被丢失。

使用过程的一个例子可在§D.9中找到，该例子介绍了把过程用于规约的规定。

D.4.3.6 SDL中的时间表达

在SDL中，有几种可能需要表示时间的场合：

a） 在一定时刻（绝对时间或相对时间）激活一个过程；

b） 在执行跃迁动作时消耗的时间；

c） 将信号从一个进程传送到另一个进程所需的时间，这会涉及到信道。

D.4.3.6.1 计时器和超时

在一个系统中，测量时间和请求时间到期的需要是由计时器和一组作用于计时器的操作来满足的。

“Timer”（计时器）是一个预定义类型，在进程中，所有计时器的使用都必须加以声明。对计时器可执行“SET”和“RESET”操作。SET操作请求在达到一特定时间的时候发生超时，而RESET操作则撤销任何已请求的特定时间。（应注意：在计时器未发生超时的时候再进行SET操作，就等效于先RESET再SET。）

```
DCL T1 Timer;
SET (NOW + 20 s, T1);
RESET (T1)
```

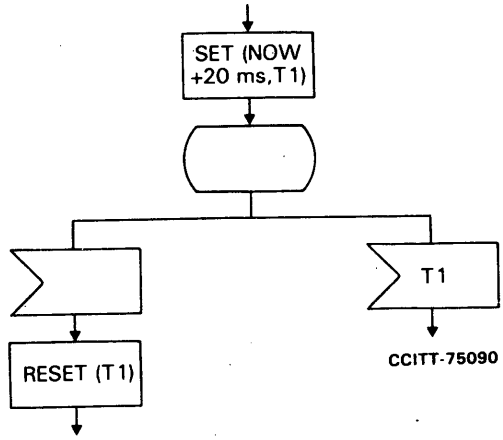
图 D—44

计时器的声明和操作举例

在SET 操作中必须规定一绝对时间。通过加上“NOW ”操作可以把相对时间转换成绝对时间。“NOW”操作产生当前时间。例如，从现在起20秒的表达方式是：NOW + 20 s

只在进程处于一个状态时才监视计时器。当信号到来时要检查计时器，如果计时器的时间等于或小于当前时间，才把该信号送入进程的输入队列，计时器于是隐含地复位。

超时信号的名字和计时器的名字相同，且不携载数据值。



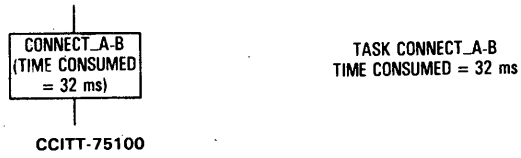
图D—45

计时器使用举例

D．4.3.6.2　规定被动作消耗的时间

当使用SDL 来模拟系统性能时，必须规定在跃迁中消耗的时间。现用的SDL 不适应这一点，但提供了下列指南：

在这种情况下，要将一时间值（在执行一给定动作时消耗的时间长度）作为一个参数附加于该动作名字，这个任选的参数可用一关键字来标识（例如TIME CONSUMED = 时间值）。



a)一个需时32 ms的动作由一含有时间消耗的任务符号来表示　b)用SDL/PR来表示同一个动作

图　D—46

跃迁时间的表示

作为附加于动作名字的一个参数来指示时间而不用注释，是因为注释应为人提供解说，而不一定必须为机器所理解。

D．4.3.6.3　信号传递时间

从信号的产生（输出）到消耗（输入）所经历的时间，表达起来是极为复杂的，因为它取决于若干因素。

将信号从一个功能块传送到另一功能块，这个动作会消耗掉一段绝对时间，这部分时间可表达为（若信道已显式地表示出）由信道执行的诸动作所消耗的时间之和。

根据系统负载和信号的处理策略，存在一相对时间消耗。同样地，在激活接收进程中所耗费的时间取决于操作系统的特点和进程的优先级、系统负载、并发事件等，这个时间只能估计，并且只有在研究系统能力时才涉及。

另一种不能静态地确定的时间，是在队列中等待前面的信号被消耗掉所耗费的时间。

通常，时间是不容易静态地表示的，且它只是在模拟和研究系统能力时才被涉及。

不过，若在比较详细的系统描述层次上全部动作均有时间消耗指示，且该描述含有信道表示（操作系统），则有可能模拟该系统，並有可能从系统能力角度出发用各种技术来评估它的性能。

D．4.4　与SDL结构有关的正文

下面介绍与SDL结构有关的正文的约定，这适用于GR和PR两者。

SDL中的正文可为：

—形式正文；

—非形式正文；

—注释。

形式正文是可以形式地予以解释的正文；非形式正文只能非形式地予以解释；而注释只是注解，不能被解释。一个图的意义有无注释都一个样。

D．4.4.1　形式正文

形式正文用SDL／PR语法来表示。在SDL／PR中，正文的允许组合在语法中定义。与一图形结构相关联的正文或者包含在该符号中，或者包含在与结构相连的正文扩展符号中（见§D．6.3.6.19）。

在形式正文中，大小写字母可以自由地混用。一个名字用大写或小写字母来表示是没有区别的，不过我们建议最好要前后一致地使用大写和小写字母。

D．4.4.1.1　名字

一个名字代表它在正文中所关联的实量，并用于标识该实量。下列各项均要求有名字：

BLOCK，PROCESS，SIGNAL，CHANNEL，STATE，
SAVE，INPUT，OUTPUT，MACRO，CREATE，START，
PROCEDURE和CONNECTOR。

名字必须以一字母开始，它可以含有字母、数字、民族字母和下横线，但注意不允许有空格。

合法名字的例子是：

CALLHANDLER RINGING-TONE

不合法名字的例子是：

START CHARGING　（它含有一个空格）

1982　（它以一数字开始）

INPUT的名字用来标识接收到的SIGNAL，反过来也成立：OUTPUT的名字用来标识发送的SIGNAL，因此OUTPUT-SIGNAL-INPUT采用了同一个名字。连接符（在PR中是JOIN）有一个相关联的名字，此名字可以用一数字开头。连接符还有一点特别，就是它没有相关联的正文串。

在SDL／PR中的所有关键字都是保留字，因此不能用作名字。全部保留的关键字在SDL／PR概要附件（2），中给出。

D．4.4.1.2　形式参数

形式参数用在语句中或用在进程和过程的启动符号中。

形式参数用关键字“FPAR ”来指明。

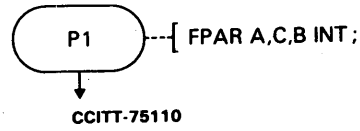


图　D—47

形式参数举例

形式参数的使用见§D．4.3.5 。

D．4.4.1.3　实在参数

实在参数用在诸如创建请求、过程调用、输出等结构中。实在参数用一对括号定界。

OUTPUT 数字　(a,b,c,d);

图　D—48

在一个输出中的实在参数举例

D．4.4.1.4　语句和表达式

任务和判定中的正文服从语句和表达式的SDL／ PR语法。

D．4.4.1.5　定义和声明

信号定义、数据类型定义和变量声明总是用有关这些结构的SDL／ PR语法来表达。

D．4.4.2　非形式正文

与GR符号或PR语句相关联的非形式正文只能非形式地进行解释。SDL／ PR关于正文串的语法适用于非形式正文，即，该正文用一对单引号括起来。

正文串的例子是：

‘Is　subscriber　free’

‘$a^2 + 2\ a\ b + b^2$ ’

请注意：在正文串中允许用’（单引号）以外的任何字符。如果需要用到’，则可用双引号”来代替。

例：

John’s　house. — ‘John’’s　house ’（约翰的房子）

若在整个SDL表示中只使用非形式正文，则引号可以省略，如果是这样，则在该SDL表示的开始就应指明这一点。

非形式正文可以有任何适于表示信息的形式。在这种情况下，作者应采用一些在他自己和读者之间便于理解的共同基础。应注意读者可能是一部机器。在这个意义上说，在读者理解方面非形式正文必须足够地形式化，以便得到明确的理解（图D—49）。

非形式正文的例子是：

- TASK 'do while x <5.....'
- INPUT x 'contains subs_identity'
- DECISION 'subscriber busy?'
- (busy);

图 D—49

非形式正文举例

与SDL语句相关联的CHILL正文是一个（从SDL观点看来）“非形式”正文的例子，尽管它是一种表示动作的很形式化的方法。

在非形式正文和注释之间有很明确的区别。

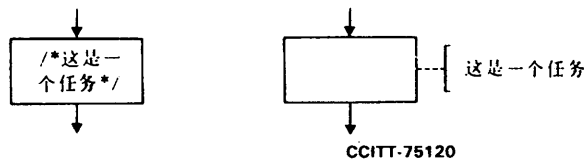
采用注释是为了帮助读者理解该表示法，有或没有注释并不改变该表示。

非形式正文则是该表示本身的一部分，因此若我们不考虑所附正文（形式的和非形式的），就不能理解该表示。

D·4.4.3 注释

SDL中的注释没有任何形式的或非形式的意义。写上它们只是为了帮助读者更好地理解GR-图或PR-正文。

在SDL/GR 中，注释可加在图的任何地方。注释的识别或者通过附加注释符号，或者用'/*'和'*/'定界（见图D— 50）。



图D－50

GR中的注释举例

在PR中，注释或者用关键字“COMMENT”来指明，或者用 '/ * ' 和 '* / ' 括起来。

```
TASK A:=2 COMMENT 这是一个任务 ;
or
TASK A:=2 /* 这是一个任务 */;
```

图 D—51

PR中的注释举例

D．4.5 未定义情况

D．4.5.1 概述

在系统从启用到退出服务的生存期中，通常会出现故障。有些故障（对软件系统来说则是大部分的故障）是由于规格说明中未作定义的情况引起的，因此检查SDL规格很重要，以便在实现以前尽可能多地发现这类情况。往后当规格已经实现且已提供了描述时，还应检查该描述是否100%地描述了系统—不多也不少！

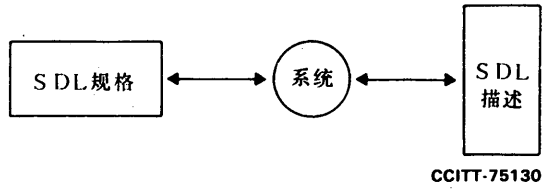


图 D—52

规格、系统和描述之间的一对一对应关系

D．4.5.2 SDL未定义情况举例

D．4.5.2.1 试图对一数据对象赋予某一在它范围以外的值，或赋予不属于其类型的值。例如，变量SUBSCRIBER- NO 具有类型INTEGER- RANGE (100000：999999)，而试图赋予它FALSE，1000000，87或BUSY等值。

D．4.5.2.2 一信号被发送到某一不存在的进程。

D．4.5.2.3 一个信号被发送到某一（新近）消亡的进程实例。不过，它可以不属于系统出错一但在SDL规格（或描述）中它是一个未定义的情况，必须用基础的系统（或模拟）程序来处理。

D．4.5.2.4 一个信号向某一不存在的信道传送。

D．4.5.2.5 物理系统中的行为出错也会出现在SDL描述中，如果已取得一一对应的关系的话。

D．4.5.2.6 将信号引向某一信道或某一进程而未对该信号、信道或进程作适当的声明。

D．4.5.2.7 当进程试图执行某一判定时，该判定的回答或数据尚未准备好。

D．4.5.2.8 试图为某一进程再创建一个实例，但对该进程来说其实例数已达到了最大允许的数目。

D．4.6 SDL中的初等数据指南

高级数据的概念在§D．4.7 中说明。

D．4.6.1 概述

在SDL中，进程之间的通信是通过使用信号、通过共享值、以及通过值的出口（或进口）来进行的。

进程利用输出来发出一个信号。信号是一种数据流，它携载信息到另一个特定的进程去。若接收进程通过输入识别了这个信号，则与此信号相关联的数据就可为此进程所利用。

一进程可读另一进程的数据值，如果它们属于同一功能块，且该数据已被声明为一共享值。

此外，进程可通过声明某一数据项可以出口而透露此数据项，所有其它有相应的进口声明的进程在提出申请时都可得到该数据的副本。

对某一特定的进程来说，在某一特定时刻，可有两类主要而互补的数据：

1） 该进程可以利用的数据；

2） 该进程不能利用的数据。（这第二类包括保留的但还未能以到达信号的形式被该进程利用的数据。）

只有可利用的数据能被进程用来执行其各种动作：判定、任务或输出。

当数据用于判定或输出时，在这个抽象的层次上它自身是不会改变的；但是，一个特定的任务能够生成、存贮、修改或破坏数据。

处理数据的例子示于图D—53（用于判定），图D—54（用于输出）和图D—55（用于任务）。

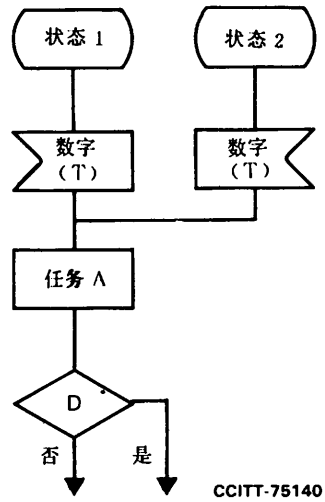
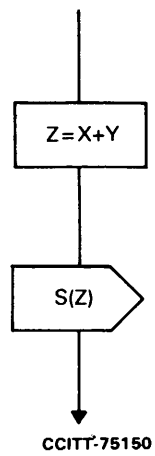


图 D—53

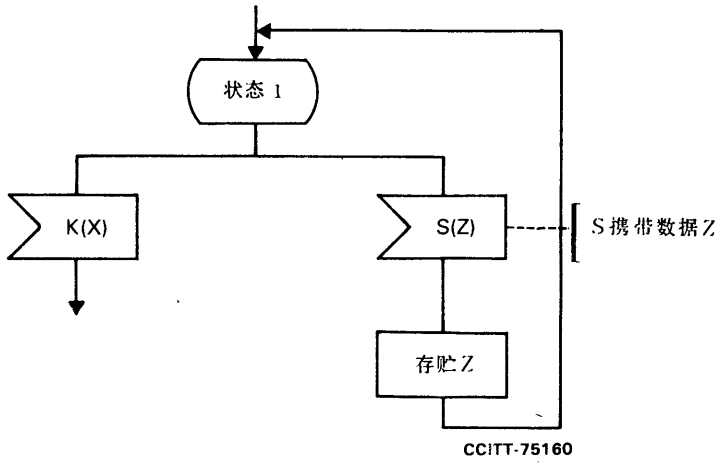
通过判定对输入的数据提问



注—S 是信号名，Z 是所关联的数据。

图 D—54

通过输出发送数据



注- S 和K 是信号名，Z 和X是所关联的数据。

图 D—55

通过任务将进入的数据存贮起来以备后用

共享值和进口/ 出口值的概念可用来作为一种交换信息的手段，代替信号。

D．4.6.2 在一个进程内部处理数据

存在三种进程间相互通信的方法，即通过SIGNALS ，SHARED VALUES 或IMPORTED / EXPORTED VALUES 。这三种技术本质上是不同的，例如，某些数据项的IMPORTER 不能像SHARED VALUES 那样透露这些数据项。

数据局限于某一进程实例。这就是说，所有数据有一个且只有一个拥有它的进程实例。

数据拥有者进程实例能够改变数据的实在值。

举例：设数据项 a 和 d 为整数且局部于进程 P，那么图D—56中动作序列的例子在进程 P 内部是合法的。

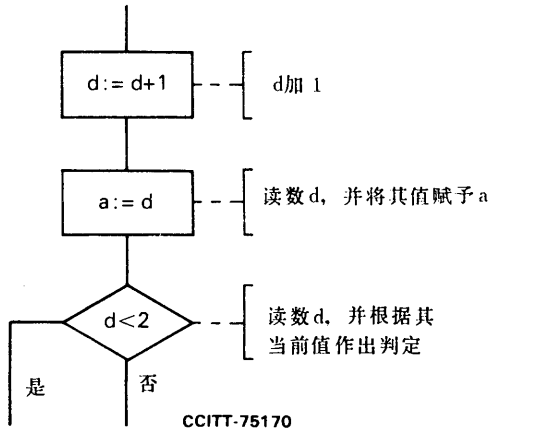


图 D—56

进程内部施加在局部数据上的动作举例

D．4.6.3　在多个进程间处理数据

两个进程可采用信号以外的办法互相交换数据。一个进程可以读另一进程的某一数据值，如果它们属于同一功能块而且该数据被声明为“共享值”（§D．4.6.3.1）。另外，一个进程可以读另一进程的某一数据值，如果已声明该数据为“可出口的值”（§D．4.6.3.2）。不拥有某数据的进程不能修改该数据的值，但是可以做一个副本用作局部数据，这样就能像对所有其它局部数据项那样对该副本进行操作。

一个数据项不能既被进口，又被共享。

D．4.6.3.1　共享值的读取

一进程可通过声明其某些数据值或全部数据值为“共享值”而透露它们，这样做的效果是使所有其它进程（与进行透露的进程属于同一功能块）有可能读取该共享值，就象这个共享值就是局部值那样。这意味着一个视见进程所看到的值总是等于透露进程所看到的值。

可以透露的对象有若干限制。

D．4.6.3.2　可出口值的读取

一进程可声明其某些或全部数据为“可出口数据”，其效果是使所有其它有一相应的“可进口数据定义”的进程在申请时可得到该数据值的副本。这意味着在进口的进程申请时，给它的数据值副本是个常量，即使后来该数据的实在值已被出口者进程所更改，此副本的数据值维持不变。

D．4.6.3.3　使用共享值和使用可出口值的区别举例

在图D—57的例子中，进程1只有一个实例。若可能出现多于一个的实例，则必须提供某种方法，来唯一地标识有关的数据项d。

D．4.6.3.4　共享值

D．4.6.3.4.1　概述

SDL用户会发现，共享数据的定义提供了一种简便的方法，用来规定两个进程之间的通信联系。但是在实现这样规定的系统时出现了许多问题，这一节打算指导用户们来避免和克服这类问题。用SDL来描述已实现了共享数据的系统困难较少，因为这些问题已在实现中予以克服，且有可能将选择的解决办法映射到SDL上去。

在本节的余下部分，假设进程A拥有数据。进程A将这些数据透露给要视见这些数据的进程B。

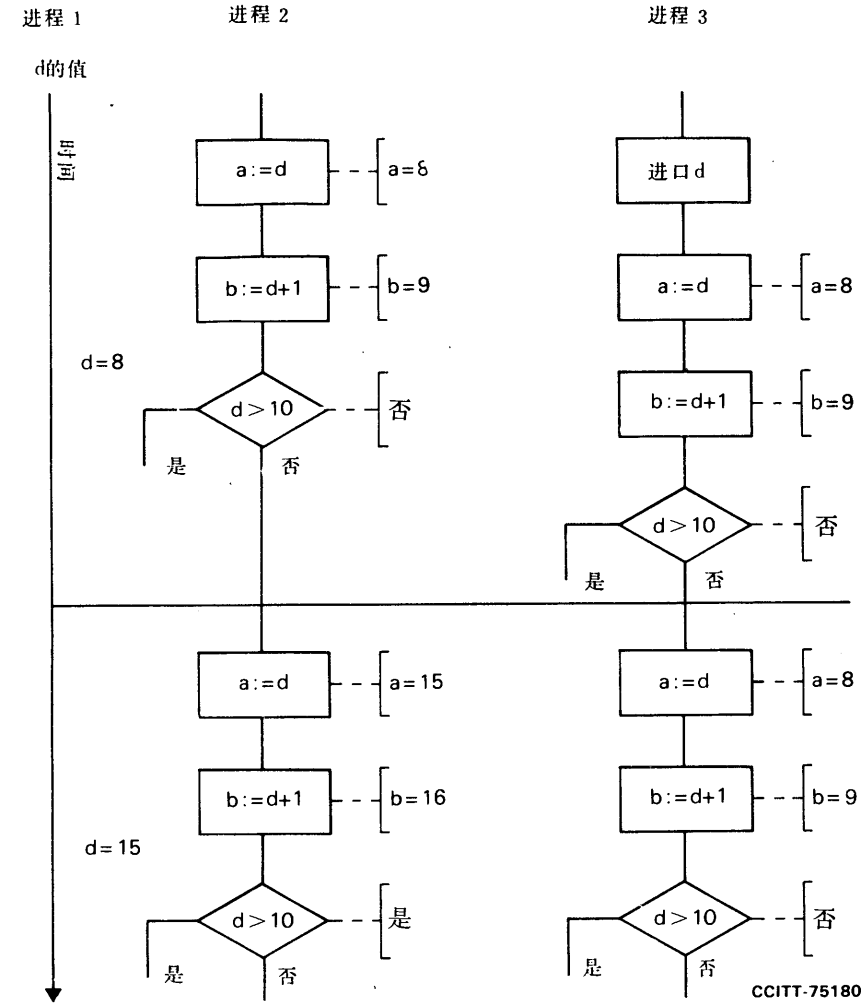
D．4.6.3.4.2　创建问题

如果企图在进程和数据被创建之前就视见该进程中的这些数据，那么其结果就是一个SDL错误。用户可以用两种方法避免这个问题，任选其中之一：

—— 保证进行透露的进程实例A在进行视见的进程实例B之前就已创建，并已预置了有关的数据；或

—— 保证B在A已被创建并预置了有关数据之前不进入要使用共享数据的跃迁。

获得前一种情况的简便方法是使A成为B的父亲（或先辈），或在创建系统的同时创建A（隐式创建）。在后一种情况下，可安排B中的有关跃迁只能被来自A的信号所触发。



注－进程 1 是d的拥有者进程，它与进程 2 共享d，且d可出口。进程 3 将d作为可进口的值。

图 D—57
使用共享值和使用可出口值间的区别举例

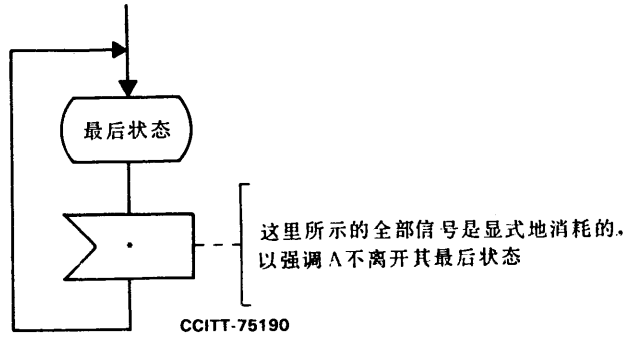
D.4.6.3.4.3 进程停止问题

进程A 到达一停止符号后，就不能再透露数据。此后要视见数据的任何企图将是一次SDL 出错。用户可用两种方法避免这个问题，任择其中之一：

——根本不在A 中使用停止符号。在图D—58中，可看出进程A 处于休眠状态，其共享数据可被看见，但不能进一步改变；或

——保证B 获知A 行将停止，並不再有任何视见有关数据的打算。使用信号的一个例子示于图D—59。

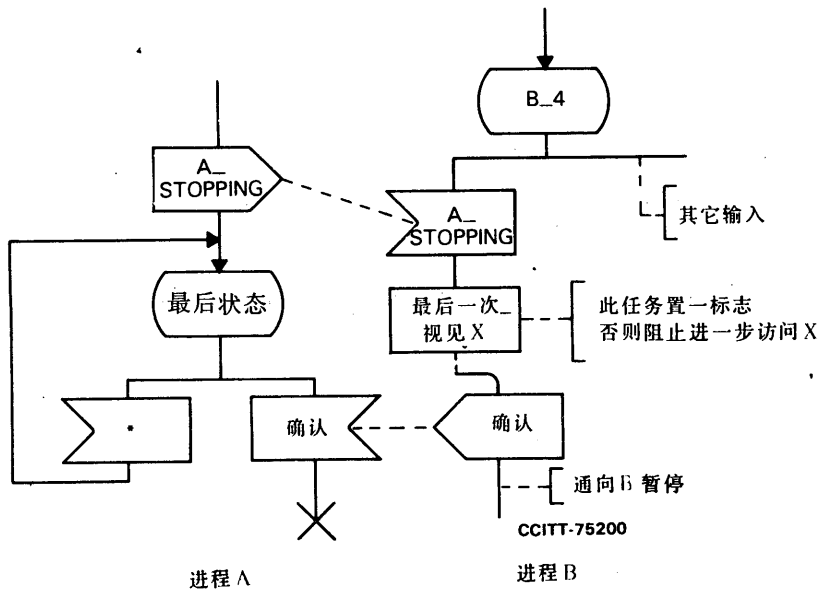
对于实现者来说，第一种解决办法的缺点是A 没有释放它所用的数据存贮区。



注-进程A拥有被透露的数据X。

图 D—58

具有休眠状态的进程，仍能透露数据但不能改变它



注-进程A拥有它向进程B透露的数据X。

图 D—59

进程A在停止之前通知进程B并取得同意

D.4.6.3.4.4 由B的多个实例引起的问题

如进程B有几个实例同时生存，例如B 1、B 2、B 3……全都去视见进程A的数据，则早先在§D．4.6.3.4.2.中的问题的解决办法要略作修改。

为了避免B的任一实例企图在A存在之前并在A预置有关数据之前就去视见A，最简便的方法是使A作为所有B的父亲。

另一种方法，若A必须向每个B发信号，则只能由父亲（B）将每个B的存在和进程标识符通知A以后，A才能这样做。

对§D．4.6.3.4.3 中进程停止的解决办法也必须略有修改。很清楚，图D—58中的休眠状态解决办法仍然是正确的。

图D—59中的解决办法不再适用了。可按下列要点构成一种解决办法。

a） A 先于B 的父亲而存在。

b） 每当创建一个新的B 时，B 的父亲发一信号通知A 。

c） 当每一个B 行将停止时（或者当它将不再去视见A 的数据时），它发一信号去通知A 。

d） A 一直在进行计数，它知道有多少个B 生存着和正在视见，或潜在地可以进行视见。

e） A 向B 的父亲发一信号，要它停止创建更多的B ，並得到认可不再创建B 。

f） 当潜在地可以进行视见的B 的数目计数到零时，A 可以停止。

这个办法涉及A 进程的部分示于图D—60。

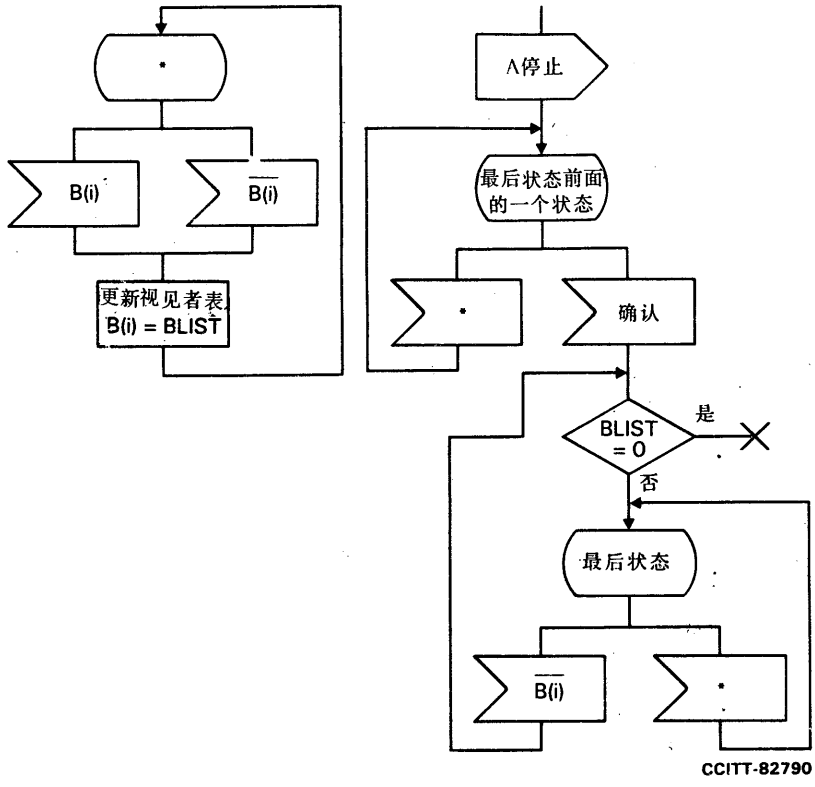


图 D—60

进程A 的两个图

D．4.6.3.4.5 由A 的多个实例引起的问题

如进程A 有几个实例同时生存，例如A 1 、A 2 ……，而B 想要去视见数据，则B 必须确定是哪个A 掌握着有关数据。这就是说，B 必须知道有关的A 的进程标识符，且必须肯定A （其数据要被视见）的存在。

对这些问题的一个可能的解决办法可以构成如下：

a） A 的父亲向B 发一信号，信号含有其子孙之一（比如A 6 ，它掌握着B 想要视见的数据）的进程标识符。

b） B 视见该数据。

c） A 6 向B 发一信号，通知B A 6 行将停止，且不应再视见数据。

d） B 同意了並用一个信号回答A 6 。

e） A 6 停止。

D.4.6.3.4.6. 由A和B的多个实例引起的问题

涉及进程A 和B 两者的多个实例，最常见的实际情况是成对的A 和B 之间存在数据共享关系。§ D4.6.3.4.2 和§ D.4.6.3.4.3 中的问题可通过下列办法避免：

a） A 的一个新实例，比如A i，一创建就立即创建B 的一个新实例，比如B j。A i 于是知道B j 的进程标识，把它看成是子孙，而B j 将知道A i，把它看成是父亲。

b） B j 视见数据。

c） 当B j 已经最后一次对数据视见完毕，就向其父亲A i 发一信号，並且停止（或通过别的跃迁继续运行）。

d） A i 停止。

D.4.6.3.4.7 共享几个数据项

考虑A 拥有几个数据项（比如 X1、X2 和 X3，它们都要被B 视见）这样一种情况。为了简单起见，假定只有一个进程A 的实例和一个进程B 的实例。

若当B 正试图视见 X1、X2 和 X3 时，A 正对它们进行更新，那么可能B 会得到一组新值和老值相混合的数据，也就是一组不一致的数据。

在有些系统中，一致性比具有更新的数据值更为重要，在这种情况下要采取措施以加强一致性，这里讨论两种可能性。

a） 不是共享 X1、X2 和 X3，而是定义一个组合结构X，它具有 X1、X2、X3 作为域和（或）子结构。

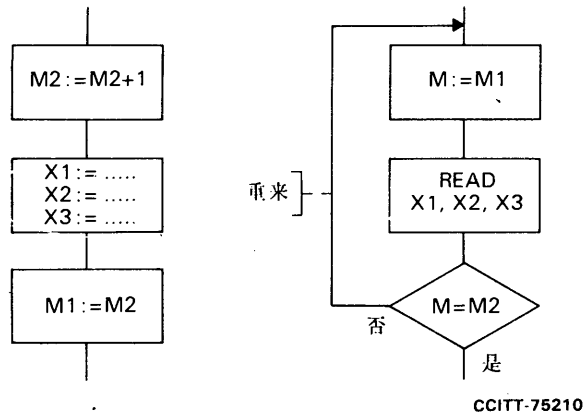
X 由A 透露，並由B 视见。

A 现在得到 X1、X2、X3 的更新值，当全部被更新后，把所有新的子域向X 一次赋值(假设SDL把赋值看作为不可分的）。

用这种办法就决不会视见到一组不一致的数据。

但问题並没有真正解决——只是转给了系统实现者。因为在大多数的实现中，只有这些域是放在一个机器字内，赋值才是不可分的。

b） 为了解决提出的问题，我们需要在S D L 中规定某些形式的锁。一个简单的不会发生死锁的例子示于图D—61，它要求引入两个标志：M 1 和M2 。



a)进程A更新X1、X2、X3.用分开的任务来强调次序先后的重要性，先是M2，然后诸X，然后M1

b)进程B读X1、X2、X3.如发现M1和M2不相等，则重来

注-进程A 向进程B 透露M1、X1、X2、X3、M2；M 1 和M 2 是INTEGER 类型，初始时置零。

图 D—61

保证一致值的一种锁

这些标志在实现时，必须使它们能被不可分地修改（即，在一个字内）。

实现者应检查更新一次数据所耗的时间和出现的频度，並估计重复检查的可能性，并且确信他将得到所要求的性能。

D.4.6.3.4.8 结论

上面几小节展示了用共享数据来规定系统功能时会出现的一些问题，并举例给出了若干避免它们的办法。问题和解决办法都没有举完，用户们会遇到其它要避免的问题，也会找到其它避免这些问题的方法。在任何情况下，所选择的解决办法应和所设计的系统的其它方面相协调，以减少复杂性。建议用户要仔细地分析采用共享数据的系统的行为。

最后，指出一点（也许是有价值的），即通过信号来发送数据，这样这些问题的极大部分都可避免。

在SDL中，数据值被认为是不可分的，也就是说，一个数据对象在它的范围内或者具有这个值，或者具有另一个值。当实现数据对象时，一个实现每次只能修改一个复合数据的一部分。当共享这类数据时，实现者应提供机制来保证一致性（如锁等）。

注　此问题通常可用可出口数据来避免，因为数据拥有者进程不能在数据变动时（这时数据会不一致）回答另一进程的申请。一个数据值在一次跃迁中可以完全被改变，由于进程只能在某一状态中被中断，因此该数据将是一致的，这样就保证了“互斥”。

D.4.7 SDL高级数据指南

D.4.7.1 数据类型定义

D.4.7.1.1 概述

SDL提供了若干预定义数据类型，然而对用户来说要定义新的数据类型也是可以的。这些类型一经定义，在声明变量、信号等时，就可象预定义类型一样地使用。

就象大多数SDL概念那样，类型定义的机制可用在若干抽象层次上，其范围从基本语法的非形式定义到类型语法和语义的完全的形式定义。

类型可以“参数化”，这样就提供一些“积木块”，从这些“积木块”又可建立更广泛的类型，这些就叫做类型生成元，并且象SDL进程那样，必须要通过这些“积木块”的具体实例来建立一个或多个性能稍有差别的近似类型。内部类型数组就是这样的一个例子，因为数组的“内容”只是在使用它时才提供。

每当一个非常精确而又无二义性的规格说明是必不可少时，总是应当使用定义新的数据类型的功能。往往即使是对十分熟悉的类型，仅仅指出一个类型的名字是不能使一个规格说明的所有读者都能确定该类型的确切性能的，在这种情况下就应采用高级SDL数据概念。

D.4.7.1.2 抽象类型介绍

SDL中的所有数据类型都定义为抽象类型。一个类型定义一组值和一组可作用于这些值的运算。例如，一组布尔型的值是True（真）和False（假），而其运算是传统的布尔运算—Not、And、Or等。虽然乍一看，抽象类型似乎有局限性，但在现代语言理论中，它的概念是广泛被接受的，并且允许用户定义所需的任意数据类型。传统编程语言中的数据结构仅仅是抽象类型的一个子集。各种运算的行为可以通过“注释”正文非形式地进行定义，或用一组形式规则来定义。

这种办法允许类型的定义与它们的实现完全无关，因而SDL中数据类型的使用不影响实现技术的最终选择。

D.4.7.1.3 数据定义

数据定义可分成三个不同的类别：数据类型定义、数据类型生成元和同义词定义。

D.4.7.1.3.1 同义词定义

同义词定义允许一个名字等效于一个值，该值由一个表达式（该表达式依赖于上下文）产生。“强定类”要求表达式的类型是明确无二义的，否则该表达式类型必须加显式说明（例如表达式“4”是二义性的，因为它可以是实数型也可以是整数型）。

同义词定义通常在下列情况时作为“简写”符号来使用，即在一个表达式被用于多个地方的时候，或者为了便于修改一个经常使用的常量而设置一个集中修改点的时候。

例如：

```
SYN Single Integer = 1
SYN BlockSize = 512 * BitsPerByte
SYN Legs = (4 * Dogs) + (2   Birds)
```

D.4.7.1.3.2 数据类型定义

数据类型定义机制体现了SDL中形式地定义新类型的主要技术，它提供了两种可供选择的方式：*newtype*（新类型）和*syntype*（同义类型）。

一个*syntype*类型定义一种类型，它是某些现存类型（称为父本类型）的值的子集，某典型用法如一个编程语言的“范围限制”，以便能对一个变量可能取的值进行较为严格的控制。

这就使各类型间能够“结构等效”，也就是说，如果各个类型有同一基础结构（父本类型），即使它们有不同的名字，它们也是相容的。

例如：

```
SYNTYPE Digit = Integer
    CONSTANTS 0:9
END Digit

SYNTYPE WarmColours = Colours
    CONSTANTS Yellow, Orange, Red
END WarmColours
```

一个*newtype*类型引入一种完全新的数据类型，它可以也可不以现有类型为基础。新的字面值和运算符名字只有通过*newtype*定义才能被引入。

*newtype*定义提供了各类型间的“名字等效”，也就是说，只有在声明一些变量的类型名字在文字上相同时，才认为这些变量具有相同的类型。

一个*newtype*类型可以独立于所有其它类型来进行定义，可为一个现有类型的扩展，或为一个类型生成元产生的具体实例，或为提供给STRUCT的内部类型生成元所产生的具体实例。在前两种情况下，一个重要的特性是能够将该类型的性质定义成形式公理。这些任选的公理通过说明组合运算的效果来定义该类型运算的语义，这样所有的运算可彼此相互定义。布尔类型被用来作为定义所有抽象类型的基础，它是所有类型全部公理的核心（虽然布尔类型在一个特定的类型的公理中不一定出现，但是递推地考察各种类型，最终将指出所有类型都可按布尔类型来定义）。

构造这些公理对SDL用户来说务必要小心，以避免两个重要的错误：不完全性和不一致性。两者都会导致具有潜在二义性的SDL规格。在第一种情况下，该组公理多半不足以完全表示被定义的类型的特性；在第二种情况下，公理间相互矛盾，这无疑会使规格说明无效。可惜这些问题的检测必须由SDL用户进行，因为SDL没有为代数类型定义中的不完全性和不一致性提供自动检测技术，用户只能依靠对给定的一组公理进行直观的估计。

D.4.7.1.3.3 数据类型生成元

一个数据类型生成元好比是一个“样板”，由这个“样板”可以建立任意数目的新类型。在整个源生成元的一个正文副本中，生成元的参数按正文被替换，于是这 个新的生成元就可被解释为该类型定义的一部分，从中可以找到该生成元产生的具体实例。

很明显，在需要若干相同类型的场合，这样就避免了类型定义的多余的重复，一部分不影响基本公理的类型除外。

D.4.7.2 数据定义举例

下面给出一些数据定义的例子，这些例子並不是详尽无遗的，它们的目的只在于让我们体会一下SDL采用的数据定义机制的功能。

除了类型定义之外，还给出了这些类型的数据对象的声明及作用于它们的运算的例子。

按照所使用的SDL形式，声明和运算将包含（GR形式）或者不 包含（PR形式）在图形符号中，然而两种形式都保持有相同的语法。

D.4.7.2.1 例1：用户计费表的定义

这里给出一个用户计费表的一种可能的定义，该用户计费表是依据自然数类型的一般性能并把容许的运算仅限于加法来进行定义的。

定义

```
NEWTYPE SUB_METER
INHERITS NATURAL ("+")
END SUB_METER
```

使用举例

```
DCL A_METER SUB_METER;


A_METER := A_METER +10
```

D.4.7.2.2 例2：一种通用类型“二维数组”的定义

这里定义了一个二维数组。该定义是通用的，既没有规定两个下标的范围，也没有对数组的分量规定任何限制，这就使该类型能用在任意范围和分量类型的二维数组实例中。

允许有三种不同的运算（除了隐含的赋值运算外）：声明，插入，抽出。

借此可以分别做到：声明哪种类型的新对象，在数组的一个指定位置插入一个新值，以及从数组的一个指定位置抽出一个值。

按照数据定义的语法，对每个允许的运算符在该定义的*OPERATORS* 部分中给出它的定义域和公共域。公理部分给出运算符的语义。

每个运算符名字后面如跟上一个惊叹号（在这种情况下是它们的全体），是指在定义中使用该名字，而当使用该类型的对象时在具体语法中用的是一个不同的名字。

---

1) 当不指明结果类型（箭头后面）时，其结果是带撇（′）的参数被代替。

定义

```
GENERATOR BIDI_ARRAY (TYPE index1, TYPE index2, TYPE A_TYPE);
    OPERATORS
        INSERT! : BIDI_ARRAY', index1, index2, A_type → ;
        EXTRACT! : BIDI_ARRAY', index1, index2 → A_type;
    AXIOMS
        EXTRACT!(DECLARE!(v), I1,I2)=Error!;
        EXTRACT!(INSERT!(A,Ip1, Ip2, E1),Ipr,Ipc)=
                    if Ipr=Ip1 and Ipc=Ip2 then E1 else
                    Extract!(A,Ipr,Ipc)fi;
END BIDI  ARRAY
```

使用举例

```
SYNTYPE INDEX = INTEGER
    CONSTANTS 1:20
END INDEX

NEWTYPE
    ARRAY2 BIDI_ARRAY (INDEX; INDEX; SUB_METER);
END ARRAY2


DCL DOUB_COL ARRAY2;
DCL A SUB_METER;




A :=DOUB_COL(5,16)



DOUB_COL(5,16):=A
```

D．4.7.2.3　例 3：　比特类型的定义

newtype bit 被规定为只具有值0或1。引入运算符flip 是为了置位（或复位）比特的值（要失去公共域），而所有其它运算（Not，And，Or，“+”）都返回一个值。

定义

```
NEWTYPE bit
    Literals (0,1);
    OPERATORS
        NOT : bit → bit;
        AND : bit,bit → bit;
        OR : bit,bit → bit;
        "+": bit,bit → bit;
        FLIP : bit' → ;
AXIOMS
    Flip (0)'=1;
    Flip (1)'=0;
    Not (1)=0;
    Not (0)=1;
    And (A,B)=if A=0 then 0 else B fi;
    Or (A,B)=if A=1 then 1 else B fi;
    "+" (A,B)=if A=0 then B else if B=1 then 0 else 1 fi;
END bit
```

使用举例

```
DCL CNTRL_BIT BIT;
DCL TMP_BIT, DUM_BIT BIT;
.
.
.
.
TMP_BIT :=NOT(CNTRL_BIT)
DUM_BIT :=OR(TMP_BIT, CNTRL_BIT)
.
.
.
.
FLIP(CNTRL_BIT)
.
.
.
```

D．4.7.2.4　例 4：　字节类型的定义

字节类型已被规定是由比特组成的数组　（newtype　bitarray），作为一个数组，它继承了数组的所有性能。此外，另外一些允许的运算是：Shr，Shl，And-Mask，它们分别提供右移n位（$0 \leqslant n \leqslant 7$），左移几位（$0 \leqslant n \leqslant 7$）以及和另一字节的AND运算。

定义

```
NEWTYPE bitarray (integer, bit) END bitarray;

NEWTYPE byte INHERITS bitarray ALL
    ADDING OPERATORS
                    shr : byte', integer → ;
                    shl : byte', integer → ;
                    And-Mask : byte', byte → ;
    AXIOMS
                    extract!(shr(B,1)',J)=  if J<0 or J>7 then Error!
                                            else if J=7 then 0 else extract!
                                            (B,J+1)' fi
                                            fi;
                    extract!(Shl(B,1)',J)=  if J<0 or J>7 then Error!
                                            else if J=0 then 0
                                            else extract (B,J-1)!
                                            fi
                                            fi;
                    shr(B,n)'=  if n<0 or n>7 then Error!
                                else if n=0 then B
                                else shr(shr(B,1)',n-1)'
                                fi
                                fi;
                    shl(B,n)'=  if n<0 or n>7 then Error!
                                else if n=0 then B
                                else shl (shl(B,1)',n-1)'
                                fi
                                fi;

FOR ALL i IN INTEGER (extract!(And-Mask(B,MB)', i)
                        = IF i<0 OR i>7 THEN error!
                          ELSE
                          and (extract!(B,i),extract!(MB,i)) FI;
                      /* 当两个字节相加时，结果字节中的每位等于把原来两个字节中对应位相加 */

end byte
```

使用举例

```
DCL F_BYTE, S_BYTE, T_BYTE BYTE;
DCL TMP_BIT BIT;
.
.
.
shr (F_BYTE, 2)
.
.
shl (S_BYTE, 3)
.
.
.
And-Mask (F_BYTE, S_BYTE)
.
.
.
TMP_BIT := F_BYTE (7)
.
.
```

D．4.7.2.5　例 5：　字节类型的一个简化定义

SDL　数据定义允许用户“非形式地”定义数据类型，特别是希望用逐步求精法来定义系统时，这样一种功能可能是很有用处的。这个例子指出了“非形式地”定义字节类型的一种方法，该字节类型与例 4 中定义过的相同。（注——采用了类型bitarray的定义。）

定义

```
NEWTYPE byte INHERITS bitarray ALL
        ADDING OPERATORS
          shr : byte', integer → ;
          shl : byte', integer → ;
          And-Mask : byte', byte → ;
    /*  shr 提供右移n 位、   0 ≤ n ≤ 7 */
    /*  shl 提供左移n 位，   0 ≤ n ≤ 7 */
    /*  And-Mask 对两个不同字节中的对应位提供and 运算；结果存储在第一个操作数中 */

end byte
```

D．4.7.2.6　例 6：　一个用户的定义

这里给出一个用户的定义，这是一个很简单的例子。直到目前为止，一个用户是一个结构（在许多编程语言中称为记录，或更形式地称为若干域的集合，每个域可以有不同的类型，并且当需要时可进行选择），这个结构包含一个号码域（用户的电话号码）和一个状态域（用户的状态），这两个域是早先定义的。

允许施加于一个用户对象的操作只是连接和断开；它们的作用是分别把用户的状态置“忙”或置“闲”。

定义

```
NEWTYPE digitstring string(digit) END digitstring;
NEWTYPE status_id LITERALS(free, busy) END status_id;

NEWTYPE subscriber
    STRUCT    number digitstring;
              status status_id;

    OPERATORS
              connect : subscriber', subscriber" → ;
              disconnect : subscriber', subscriber" → ;
```

```
AXIOMS
        S=connect(S1,S2)' => S1(status)=busy;
        S=connect(S1,S2)" => S2(status)=busy;
        S=disconnect(S1,S2)' => S1(status)=free;
        S=disconnect(S1,S2)" => S2(status)=free;

end subscriber
```

使用举例

```
DCL SUB_EX, CALD_SUB SUBSCRIBER;
DCL NUM DIGITSTRING;
.
.
.
SUB_EX (STATUS) := FREE
.
.
.
NUM := SUB_EX (NUMBER)
.
.
.
CONNECT (SUB_EX, CALD_SUB)
.
.
.
```

D．4.7.2.7 —8　例 7和例 8；　用户的一个更详细的定义

这里给出非形式定义数据类型的另外两个例子，它们涉及到结构构成的可能性。每一个新的类型把一个旧类型当作它的一个域来定义，这样一个特性使得用户能集中考虑数据类型的特殊方面，而把其它方面推迟到更为详细的层次去考虑。

定义

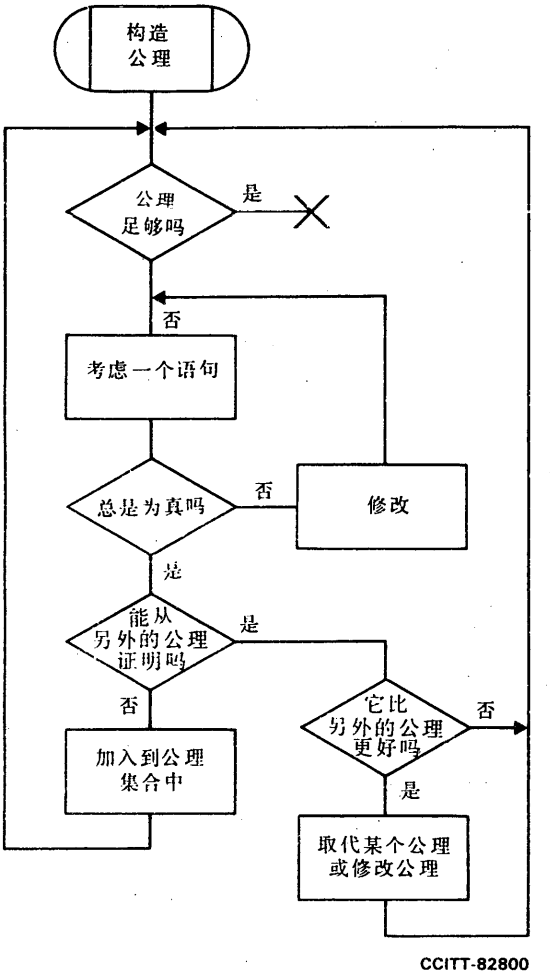
```
NEWTYPE subscriber_rev1
    STRUCT   first_app subscriber;
             counter sub_meter;

END subscriber_rev1

This definition adds the metering facility to a subscriber, here called subscriber_REV1.

NEWTYPE subscriber_REV2
    STRUCT   old_sub subscriber_rev1;
             enabling Three_cond;

END subscriber_rev2

This adds enabling conditions to a subscriber (field enabling); The Three_cond type may be defined as:

NEWTYPE Three_cond LITERALS (LOCAL_AB, LONG_AB, INT_AB) END Three_cond;
```

D．4.7.2.9　例 9：　为一个NEWTYPE 构造公理的过程

当引入新的数据类型时，重要的是建立足够的公理，这意味着：

— 在该组公理中，每个运算符至少出现一次；
— 真语句（公理以外的）是可证明的；
— 不可能找到不一致性，

这样才能满足新的数据类型的要求。



D．5　文件编制

D．5.1　什么是文件？

ＩSO　把文件定义为“用一种可检索的格式存储在某一媒质上的有限的和相关的信息”,因此应当把它看作是一个严格确定的逻辑单位。文件是用来传达与系统（用ＳＤＬ来说明规格或加以描述）有关的所有信息的。

当用纸作为存储文件的物理媒质时，文件这一术语往往不正确地用来表示纸张而不是它的逻辑内容，随着磁存储媒质使用的日益增多，这一术语正恢复到它原来的含义。

D．5.2　引言

本节D．5讨论文件编制情况，这里涉及到的是文件的逻辑组织而不是它们的物理组织，后者留待用户自己去处理。文件的逻辑组织和物理组织两者在要求上有相似之处，这意味着在下面正文中可以提供某些有用的线索，以帮助用户建立文件的物理组织。

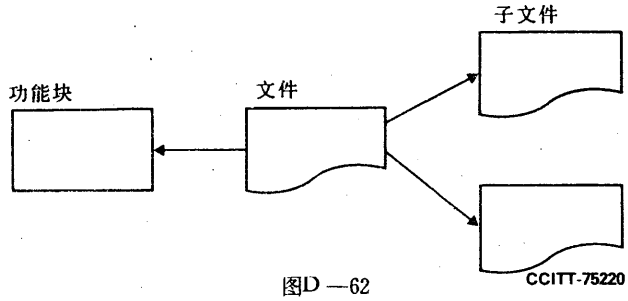
D．5.3　为什么需要文件？

把与系统有关的信息分解成若干文件，就能使系统更易于理解和更易于管理。由于ＳＤＬ没有文件的形式概念，用户不得不使用同一种非形式机制或从另一种语言（最好是ＣＨＩＬＬ）中借用文件的形式概念。只有在用单一正文串（使用ＰＲ形式）来规定或描述一个完整的系统时，才没有必要把系统分解成几个单独的文件。

D．5.4　文件结构

可以把包含整个系统定义的一组文件看做是一组结构。

一个文件结构（在这里文件要参考子文件）附在系统结构的各个功能块上，该系统结构包含了系统层次，见图D—62。



图D—62
文件结构

D．5.5　引用机制

所有的文件彼此应精确地联系起来，特别是由于更新其中一个文件会影响其它的文件。

每个文件必须有一个唯一的标识，并且必须清楚地规定它的边界。边界可以通过给出属于同一文件的页数来表示。对于一个一页的文件，例如一个功能块交互作用图，其边框符号能表示其边界。

同一文件能作为几个（在同一系统中的）功能块的子文件出现。

§D．9（“举例”）一节给出两种引用的方法，一种方法是基于PR语法，另一种方法是使用GR图中的注释机制，更详细的内容见§D．9。

D．5.6　文件的类型

GR语法中引入的各种类型的图，通常是一些与已知定义相对应的文件。它们也应当指明这些文件的边界。如前所述，合在一起能覆盖一功能块定义中的所有定义的一组文件应当形成一个文体结构。在这个结构中，各种文件内至少应安排有进程定义；如果使用功能块交互作用图，那么在同一文件中也应当包含信道定义；在单独的文件中，各信号定义和数据定义最好应组合在一起。

过程定义和宏定义同样应当形成进程文件的子文件。

在称为信号表X，Y，Z等的单独文件中，可以安排形成一个信号表的那些信号的定义。图D—63给出一个功能块的一种可能的文件结构。

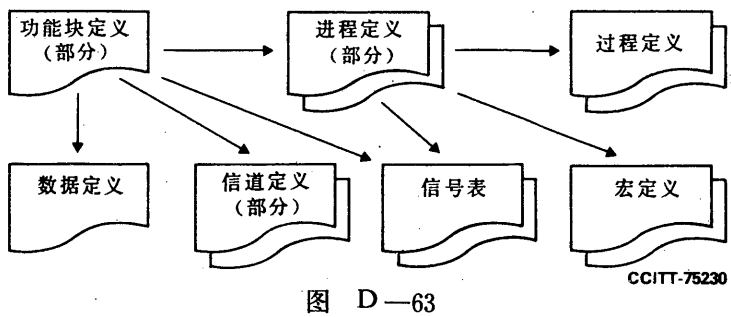


图　D—63
功能块的一种可能的文件结构

D．5.7　GR和PR的混合运用

进程定义可部分地用GR形式来定义，进程流图则或者包含在进程图中（以GR形式），或者包含在进程体中（以PR形式）。为了能够把以GR形式给出的进程流图定义和以PR形式给出的其它部分定义混合起来，进程流图定义应该形成它自己的一个文件。

一个系统定义或功能块定义也能部分地用GR形式来定义，这时功能块交互作用图应当形成一个文件。由于功能块交互作用图仅仅包含一个系统或功能块定义的若干部分，故其余部分应在一些单独的文件中给出。

当把GR和PR混合运用时，PR表示法必须分解成若干个文件。可惜，PR语法不允许这样做，因此分解应以一种非形式的方式进行。

在§D．7中讨论了一些方法，用以构成一个PR表示的各个语法上相连接的部分。

D．6　使用SDL/GR来表示系统的指南

SDL用户指南D．6章阐述SDL图形语法表示法即SDL/GR的用法。

D．6．1　为什么要使用图形语法？

图形语言具有清晰地显示系统的结构和使控制流易于形象化的优点。作为一个设计工具，SDL应当具有一种使得用户能清晰而简明地表示其思想的形式，图形语法体现了这点并且更符合扩展有限状态自动机的惯例表示法。而正文短语表示法则最适合于机器使用。

SDL图形表示法的语法（SDL/GR）是SDL的最初形式，它是在1973～1976年期间产生的并首次在Z．100系列建议的1976年文本中出现。

图形表示法的语法来源于一些图形语言，这些语言是各个组织机构为它们各自的需要而研制的。

D．6.2　SDL/GR图

在这一节加以说明的图有：

功能块交互作用图：描述一个系统或一个功能块的内部结构；

功能块树图：用各功能块和子功能块来描述一个系统的划分；

进程树图：用来定义将一个进程划分为各个子进程；

信道子结构图：定义一个信道的内部结构；

宏定义图：它规定一组符号，每当出现宏定义引用时，应当在解释它之前用该定义来代替宏引用；

过程图：规定一个过程执行的行为；

进程图：规定该进程定义的所有进程实例的行为。

信号定义和数据定义是用SDL/PR语法定义的（见§D．7.3.2），从使用该信息的图中对它们进行引用。

D．6.2.1　一般准则

绘制流图的一般准则是：

— 应该把它们画成符合正常的阅读顺序，从顶到下，从左到右；

— 图形不应包含太多的细节。把大的图划分成若干子图往往是合适的，这些子图包含各个不同的部分或方面；

— 图形的连贯部分最好应能包含在同一页上；

— 注释可以用GR注释符号或PR注释语法插在一个图形的任何地方；

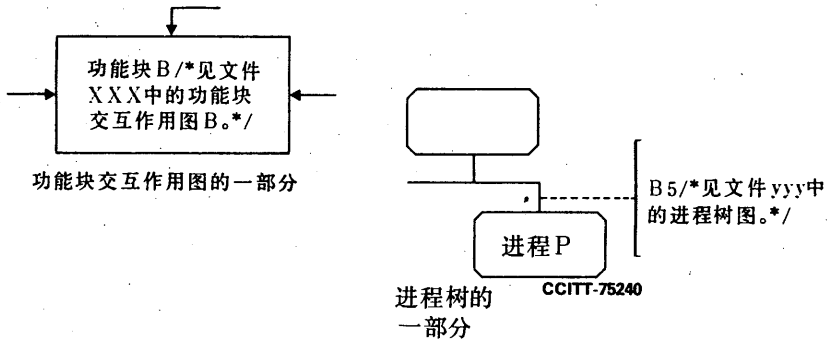
— 正文最好应放在与之相关联的符号中，如果做不到，那么可以把正文安排在一个连结到该符号的正文扩展符号中。

D．6.2.2　引用

在现行的一组SDL建议中，没有提供手段来做各GR图本身之间的引用，各SDL/PR定义之间的引用或GR图和PR定义之间的相互引用。为了形成一个系统的完整的SDL表示法，也为了构造一个系统的文件编制，必须要有一种进行引用的机制。

进行引用的机制可用若干方法产生，这取决于所选用的文件编制的组织和采用的方法学。既然在建议中没有提供引用机制，那么就由用户自己决定选择一种引用机制。

在GR语法中使用引用的一个例子是采用注或注释语法，如图D－64所示。



图D—64

把注或注释用作为引用的例子

可能有一些其它的方法，在§D．9的例子中可以找到其它的例子。

D．6.2.3　样板

SDL图形表示法的符号样板已在本卷封底页提供，图D—65给出了样板的一个简图。

图中的符号用了三种尺寸，也就是20×40mm、$20/\sqrt{2}\times 40/\sqrt{2}$mm和$10\times 20$mm，直接展示了输入、输出、判定、替换、进程、起动、任务、状态、保存、连接符及停止符号，对尺寸最大的符号还标明了内部各尺寸间的比例。

过程调用、宏及创建各符号可以在任务符符号内画附加水平线或垂直线（在该图中已标明位置）来构成。

过程定义和宏入口处符号可以在起动符号上加画附加垂直线（已标明位置）来构成。

宏出口处符号可以由连接符符号加画附加垂直线(已标明位置)来构成。

返回符号是连接符和结束进程符号的组合。

注释符号可以用任务符号的任一端来画出。

允许条件符号可以用停止符号画出。

样板中没有包含代表“全部”的符号（星号*），因为它应与符号有关的正文一起打印。

信道符号和引到正文扩展符号的线条是实线。

信号线和引到注释符号的线条是1：1的虚线。

以上介绍的所有符号都能在图D—209和D—210中见到。

D．6·3　SDL/GR的使用

对SDL/GR所用的全部符号的描述见图形语法概要（建议Z·100至Z．104的附件C）。

D.6.3.1　用SDL/GR表达结构

这一节说明了怎样把一个系统划分成若干功能块和信道，并任选地对各功能块和信道作进一步的划分。

应该注意，当把一个系统划分成若干越来越细的层次时，可能会含有一些替换图形(见建议Z．102)，这时若有两个可替换的描述，则可把不太详细的那个描述看成是一个概述。

D.6.3.1.1　功能块交互作用图

功能块交互作用图利用信道和功能块展示一个系统或一个功能块的内部结构，可以表示功能块中包含的一组进程。

也可以表示能创建其它进程的进程。

该图由以下几部分组成：

— 边框符号（对功能块是必选的，如用来表示系统则是任选的）。

— 功能块符号

— 信道符号

— 信号表

以及任选符号

— 进程符号

— 信号路径符号

— 请求符号

— 环境符号

图D—66是表示一个小系统（S）的功能块交互作用图的简单例子，它由两个功能块（B 1 和B 2）组成。应注意，对一个系统来说，包围的框符号是任选的。

按照自顶向下、自左至右的惯例，名字最好安排在命名对象的左上角，或在它的旁边。引用的定义可以安排在图的外边，如例中的信号表L1。

PR语法中描述的信号定义和数据定义，或者放进图中，这时最好放在左上角，或者安排在图的外边。如果定义安排在图的外边，在哪里找到它们就不明显，因此该图应包含一个可在哪里找到它们的引用标记。

在功能块符号中可以标明它所包含的进程，如图D—67所示。

为了作出更易读的图，可以对图进行划分，这样可以分别地展示各功能块的内部，例如，图D—67所示的片段可以形成该功能块交互作用图的一个子图，单独加以表示。

在同一功能块交互作用图中展示若干层次结构往往是有用的，为做到这一点，可以用一功能块的功能块交互作用图来替代图中该功能块的符号，图D—68举例说明了这点。

这种做法可以重复任意次。

为了有一个结构良好的、易读的图，应注意下面几点：

—图不应作得太复杂，以便一页能容纳得下；

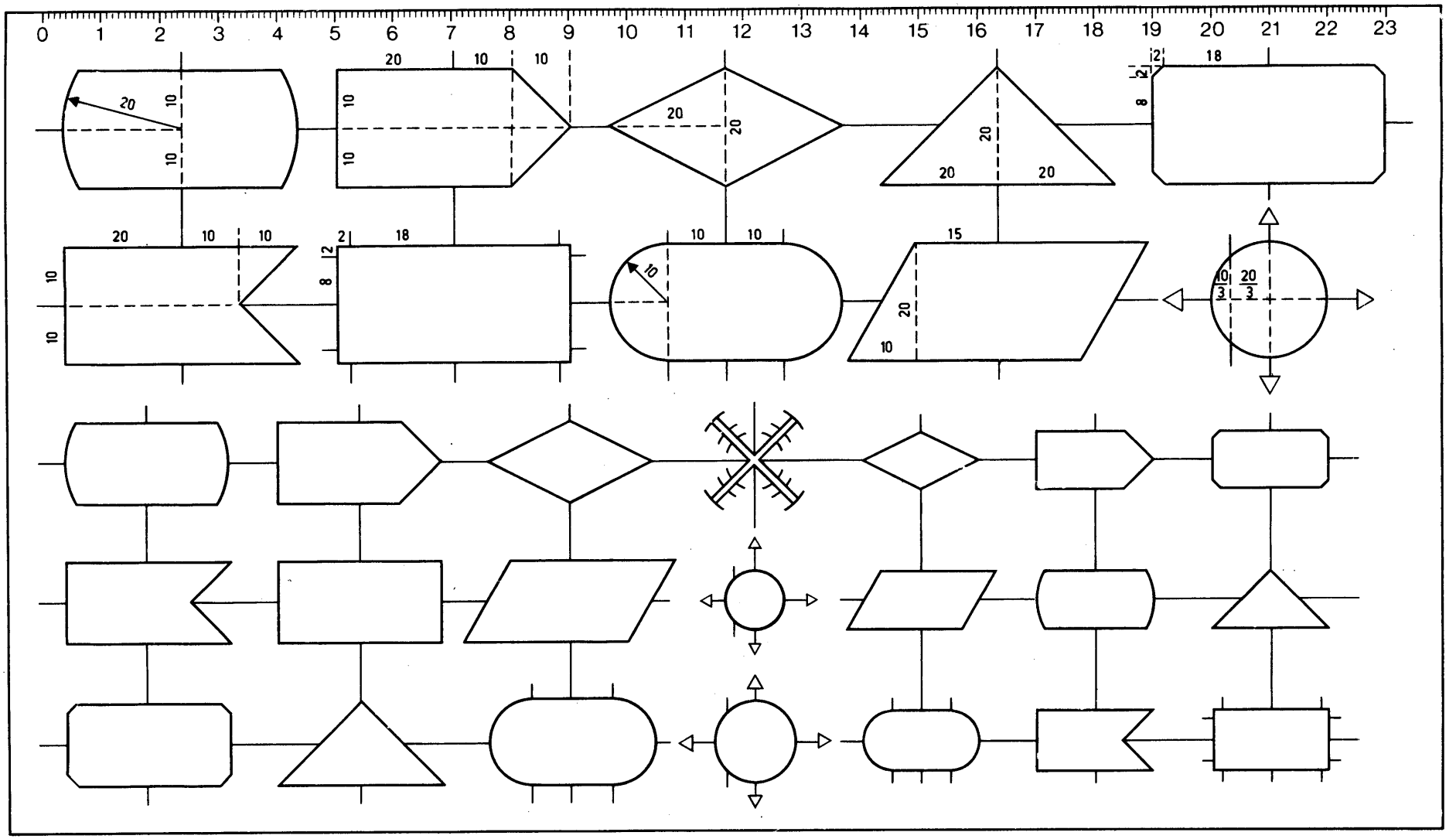
— 运用图可划分成子图的可能性，来减少细节的数量，从而使图更易读；

— 可以用宏作为工具，以简化大图的复杂性。

D.6.3.1.2　功能块树图

功能块树图利用功能块和子功能块来表示一个系统的结构，其目的是使读者对一个系统的总结构有概括的了解。

功能块树图是由一些功能块符号和一些“划分”线所组成的分层树。



注 1 -高度：长度 = 1：2

注 2 -所示的三种尺寸，也就是长40mm、28mm和20mm。40mm/$\sqrt{2}$ = 28mm，28mm/$\sqrt{2}$ = 20mm等。
这使照相缩版能将A 3纸页缩到A 4纸页，而使各符号的大小兼容。

图 D—65

SDL样板简图

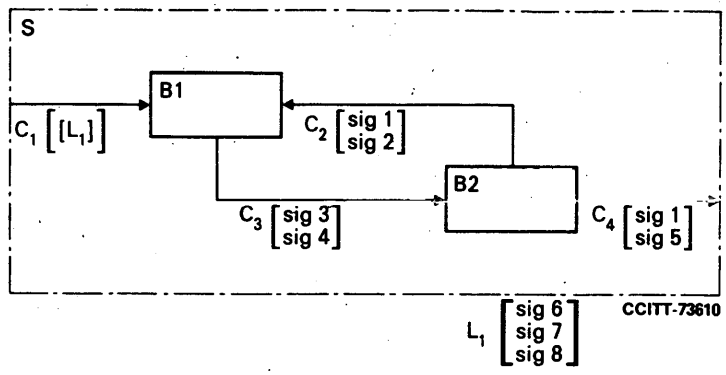


图 D—66

表示一个系统的功能块交互作用图的例子

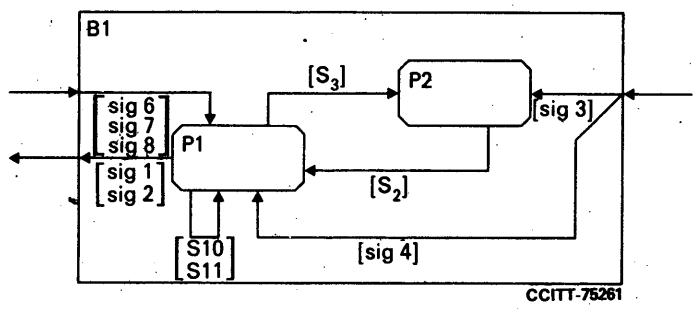


图 D—67

功能块交互作用图的片段

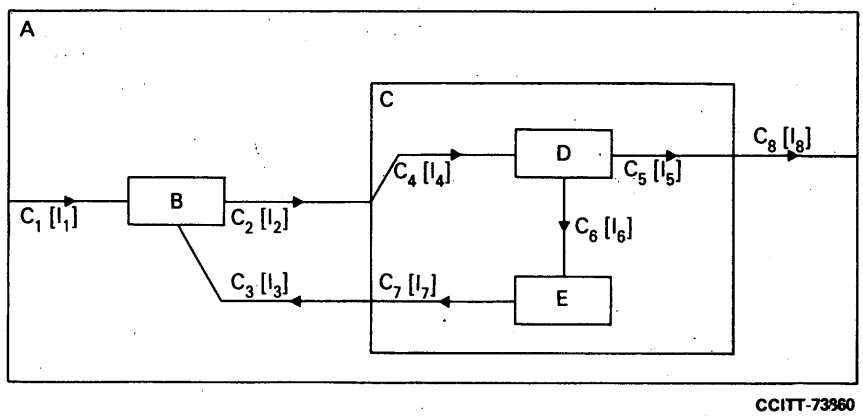


图 D—68

表示若干层次结构的例子

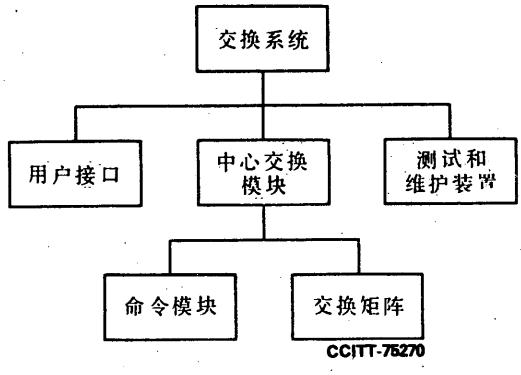


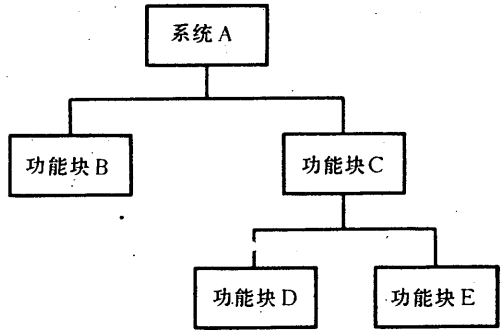
图 D—69

功能块树图举例

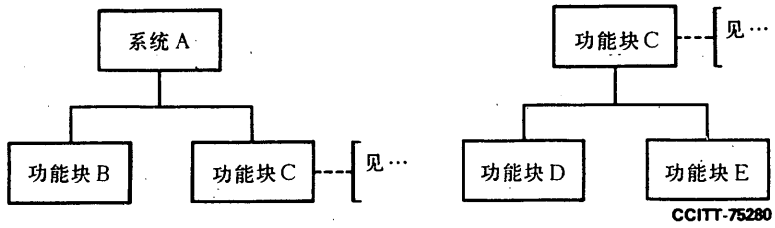
该图中所有功能块符号的大小最好应画成一致，这就使图中同一划分层次内的各功能块並排出现，在图中呈现相同的高度，如果图大，一页画不下，那就应将它划分成“局部”功能块树图，如图D—70所示。

把一个功能块树图划分成几个局部图常常是有用的。

划分成若干局部图时，应使得以系统作为根的第一个图被截断，使一组被进一步划分的功能块看起来象未经划分的一样。原图被截断处的各功能块则在表示进一步划分的图中作为根出现。



a) 未经划分的图



b) 划分后的图

图 D—70

划分功能块树图举例

如果用了局部图，而进一步划分一个功能块和（或）在什么地方找到接续的图又不很明显，则应使用注释符号插入引用标记。

D.6.3.1.3　　进程树图

进程树图描述一个进程划分为子进程，並指明各进程配置的位置。该图是一棵由进程符号和“划分”线组成的分层树，进程的配置用注释符号来描述。

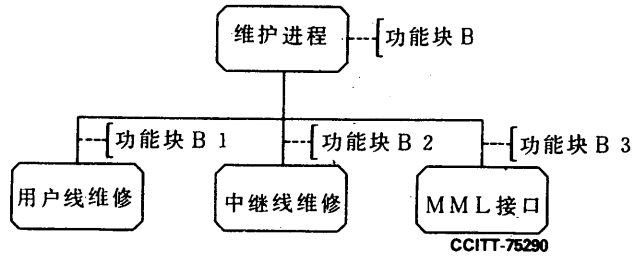
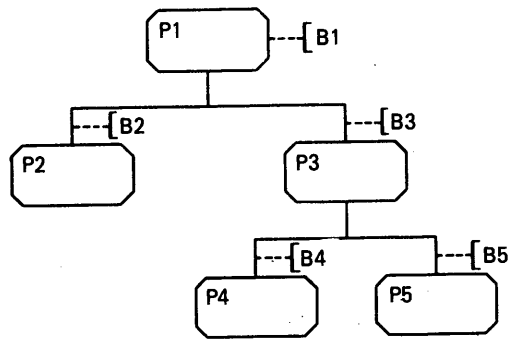


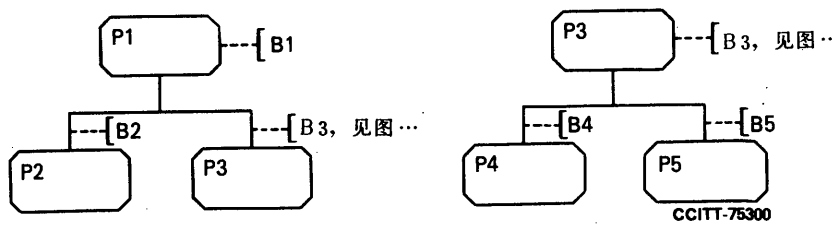
图　D—71
进程树图的例子

要使进程树图完整，其根应是一个进程，该进程不是任何其它进程的子进程，所有的叶进程也不应当被进一步划分。最好应把图形画成使所有进程符号大小一致，以使图中同一层的所有子进程分布呈现相同的高度。

如果图形大得一页容纳不下，或者有了易读性的目的，图形可分解为局部图。通过使一组进一步被划分的进程在被始图中起未被划分的进程的作用，可获得一组局部图。这些进程在另外的描述进一步划分的图中作为根出现。



a)　未经划分的图



b)　划分后的图

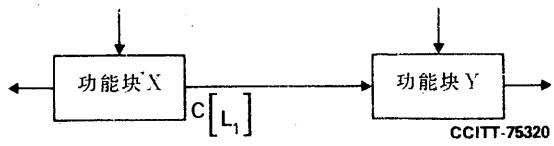
图　D—72
局部进程树图的例子

如果使用局部图，而一个进程进一步被划分不明显或找到接续图的地方不明显，就应运用注释符号插入引用标记。

D.6.3.1.4　　信道子结构图

信道子结构图描述一个信道如何被划分成子成分，该图类似功能块交互作用，所有在D.6.3.1.1给出的准准则对信道子结构图也是有效的。

在出现被划分的信道的功能块交互作用图中，应该注明对描述该划分的信道子结构图的引用，见图D—73和D—74。



/*见文件…中的信道子结构图*/

图　D—73

包含被划分信道的功能块交互作用图的片段

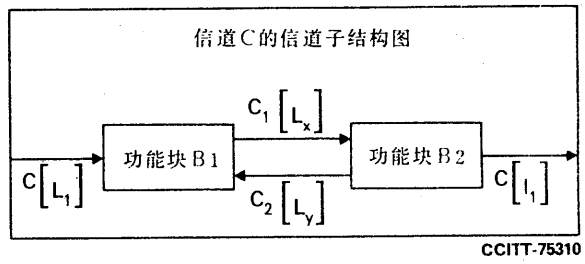


图　D—74

信道子结构图的例子

如果信道划分明显，並且找到子结构图的地方也明显，那么就可省去引用标记。

D.6.3.2　　信号定义

对信号定义没有专门的图形语法，在使用信号定义的SDL/GR图中可使用SDL/PR语法（见§D/PR正文可以插在图中，但正文最好应当只是从图中去引用。

可以引用信号定义的图形是：

— 功能块交互作用图；

— 信道子结构图。

当定义在图中出现时，它们最好应出现在图的右上角，如图D—75所示。

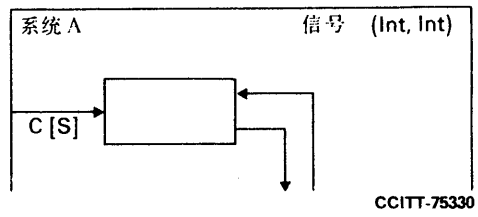


FIGURE D-75

包含信号定义的功能块交互作用图的片段

在一个图中有大量的PR正文会降低图的易读性。

最好是用一个对PR正文位置的引用来代替该PR正文，例如：

/*信号、类型及过程定义见文件…*/

D.6.3.3　数据定义

SDL/GR中对数据定义没有专门的图形语法。当GR中需要数据定义时，就使用PR语法（见§D．7 3．4）。

有两种类型的数据定义：

— 数据类型定义；

— 变量定义。

至于信号定义，其PR正文最好是从GR图中去引用。

D.6.3.3.1　数据类型定义

数据类型定义可以在下例的SDL/GR图中加以引用：

— 功能块交互作用图

— 信道子结构图

— 宏定义图

— 进程图

— 过程图

插入信号定义的准则也适用于数据类型定义。

D.6.3.3.2　变量定义

在下列SDL/GR图中可含有变量定义：

— 进程图

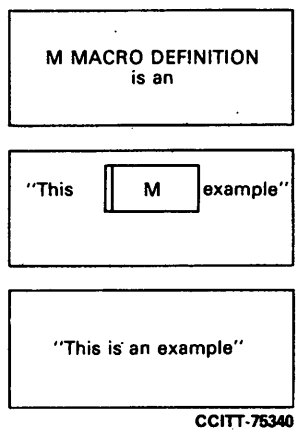
— 过程图

插入信号定义的准则也适用于变量定义。

D.6.3.4　宏定义图

宏定义图定义一组符号，对宏的每次引用应在解释之前用这组符号来替换该引用。

该图可以有一个或多个入口符号，后面跟一流线进入该图；还有一个或多个出口符号，跟在从图中引出的流线之后。除此之外，一个宏定义图可以包含有任意一组图形符号和正文的任意组合。宏的正确性和含义只有在替换发生之后才能判定。



a）宏定义

b）在上下文中的一次宏引用

c）替换之后

图　D—76

使用宏的例子

对于绘制结构良好的图的一般准则也适用于宏定义图。

D.6.3.5　过程图

过程图规定在每次调用该过程时要执行的行为。SDL 中的过程类似CHILL 和其它编程语言中的过程。引入过程的目的是：

a）使进程流图的结构构成有可能进入一些细节层次；

b）允许用一单项来表示一个可以孤立地加以对待的多项的复杂组合，以保持规格紧凑；

c）允许公用的项的组合被预定义和重复使用。

由于过程定义可以包含在系统定义、功能块定义、进程定义及过程定义中，同一过程可以由若干进程调用，允许使用公共过程库。

定义过程的图是与进程图类似的，其差别是用过程启动符号替换进程图的启动符号，以及用返回符号替换停止符号。

用于进程图的准则也可应用于过程图（见§D.6.3.6）。过程图的一个例子如图D—77所示：

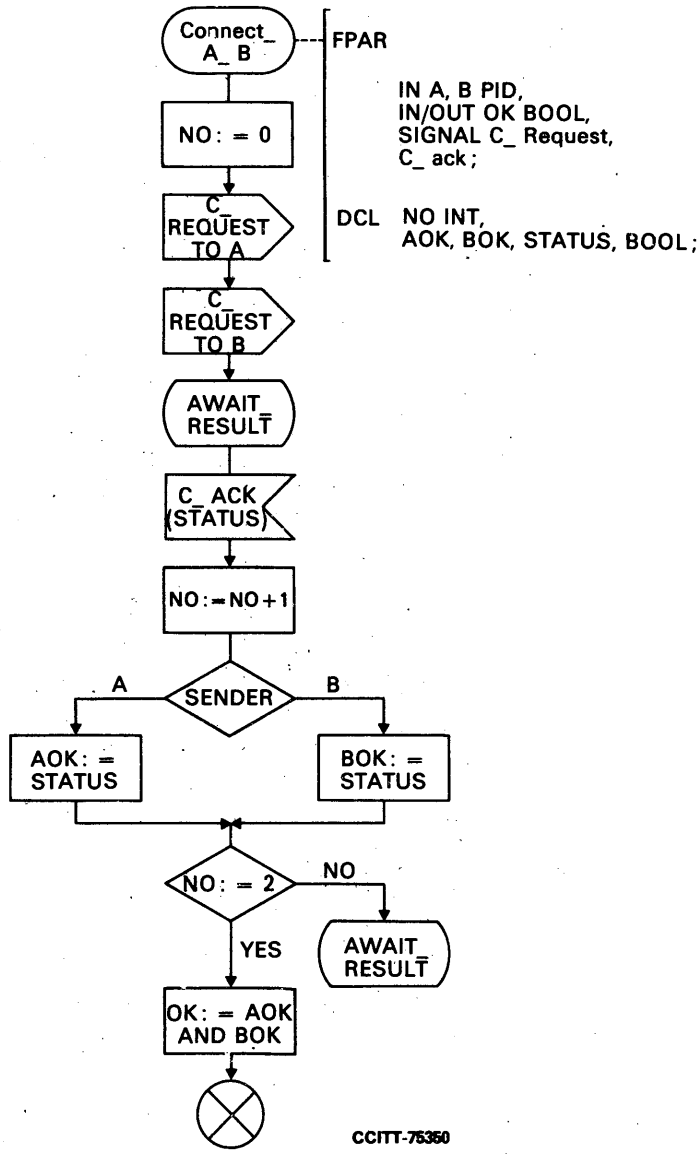


图　D—77

过程图举例

D.6.3.6 进程图

进程图使用SDL/GR定义一个进程的行为，当被描述的进程大而复杂时，用辅助的概述文件作为先行文件往往是有效的。在本节的结尾给出一些这种文件的例子。

D.6.3.6.1 SDL/GR进程图的各种型式

在一个SDL/GR图中，全都用显式的动作符号来描述跃迁，则称之为面向跃迁型的SDL/GR图。

另一方面，如果用状态图来描述状态，而用状态图的区别来隐含跃迁动作，则称之为面向状态型的SDL/GR图。

同时采用两种型式，则称之为组合型的SDL/GR图。

图D—78至D—80给出了这三种型式的例子。

当动作次序是重要的而详细的状态描述是次要的时候，面向跃迁型是适用的。

当每次跃迁中动作次序不重要时、当希望用图形说明时以及当希望用简洁的表示法时，面向状态型是合适的。为了有效地运用它，定义了一些合适的图形元素（见§D.6.3.6.22）。

当内含重复信息有利时，组合型是合适的。

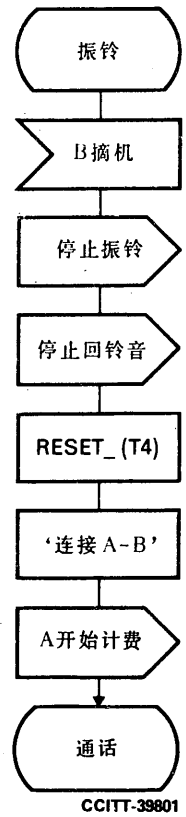


图 D—78
面向跃迁型

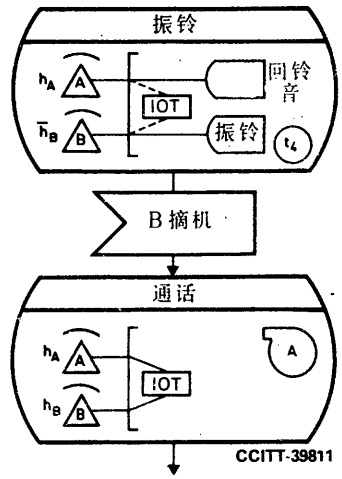


图 D—79 ·
面向状态型

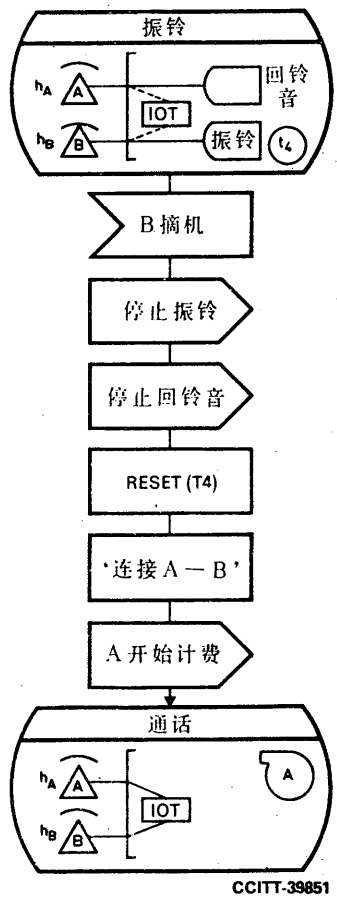


图 D—80
组 合 型

D.6.3.6.2　符号与连接规则

进程图中允许使用的符号及其连接方法见建议Z.100至Z.104的附件C。

D.6.3.6.3　结构良好的图

下面是绘制一个结构良好的图的一组通用准则。

D.6.3.6.3.1　一个图的启动与结束

进程实例可以显式地创建，在这种情况下，进程图应先画起动符号。对没有显式实例创建的进程，则有一起动状态，该状态用它处于进程图顶部的位置来标识。

通常画SDL图的流程最好是从一页的顶部画到底部。

一个图可以分解为若干页，从一页到另一页的流程只能通过连接符（§D.6.3.6.16）或通过同一状态的重复出现（§D.6.3.6.5）来建立。

D.6.3.6.3.2　一个图的几种安排

进程的挂起是用状态符号表示的，挂起是否应被强调（通过在每个状态上将图断开），让SDL用户自己决定。按照SDL规则有三种可能的方法：

方法A

在每个状态符号处图被断开，然后再从该状态符号开始将图继续下去。

使用方法A时，完整的SDL图由n个图组成，这里n是进程的状态数。每个图从一个不同的状态开始，展示发自该状态的所有跃迁，并结束于这些跃迁所达到的状态。同一状态可以出现在若干图中。

其例子如图D—81所示。

应注意，跃迁连同输入只用有向线来表示。

如果进程有起动动作，则起动符号和起动的跃迁也作为一个单独的图画出（见图D—81*b*））。同样，如果进程有停止动作，则引向停止符号的各个状态和跃迁也将分别画出（见图D—81*c*））。

方法B

图不被断开。

在方法B中，完整的图画成一完全连接的流图。图D-81的例子作为一个完全连接的流图重新画成为图D-82。

方法C

在某些状态符号处把图断开。

如果完整的SDL图只是在某些状态上被断开，则可有若干等效的方法重画图D—81，其中之一由两个图组成，表示在图D—83中。

在方法C中，某些状态将在若干图中出现。运用这种方法时，将该图中逻辑上应归在一起的各部分之间画上连接线当然是合理的。例如，进程的一小部分表示了最常用情况下的行为，这是很通常的，这个部分就应作为一个单独的图画出。

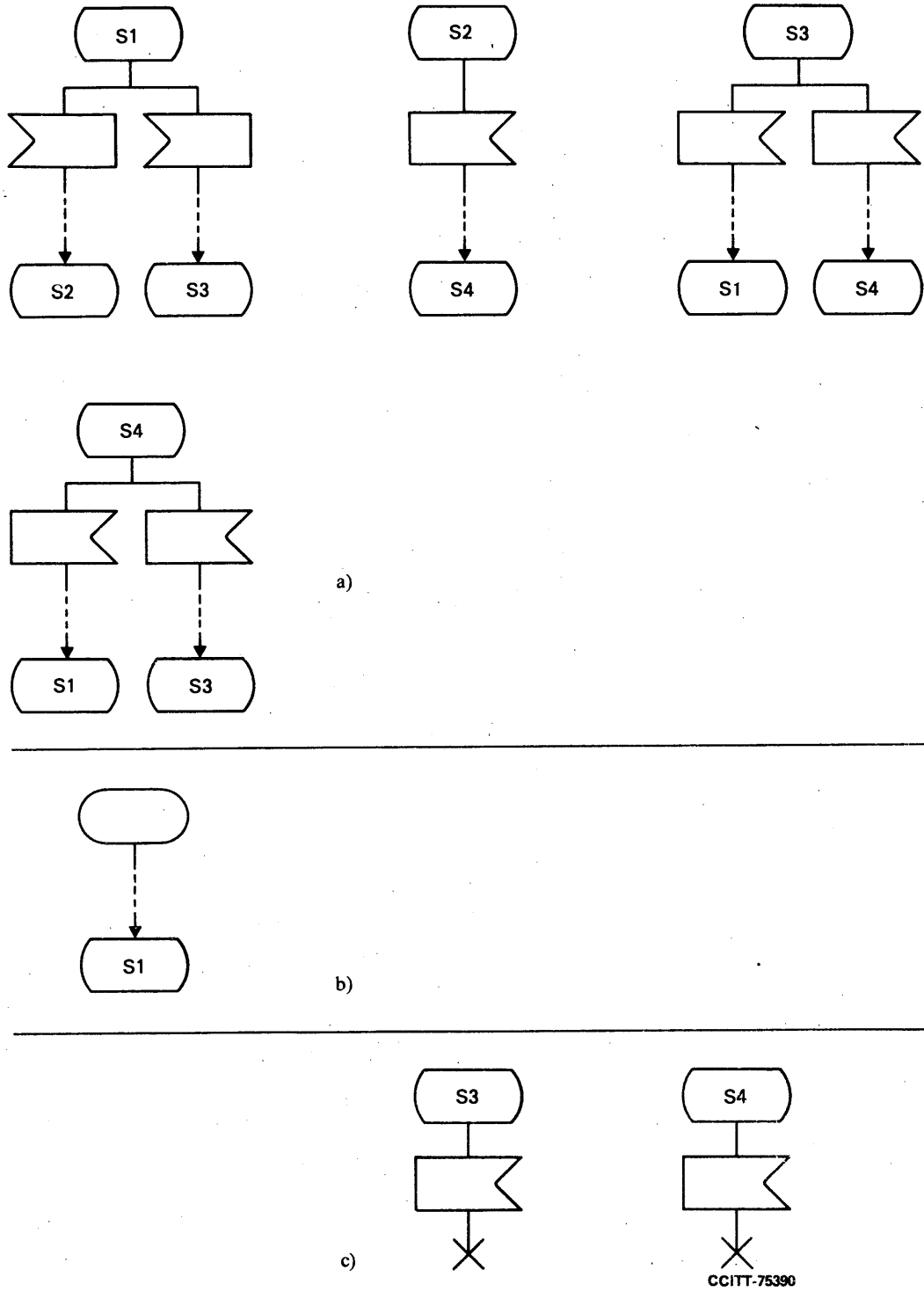


图 D—81

完全分开的流图

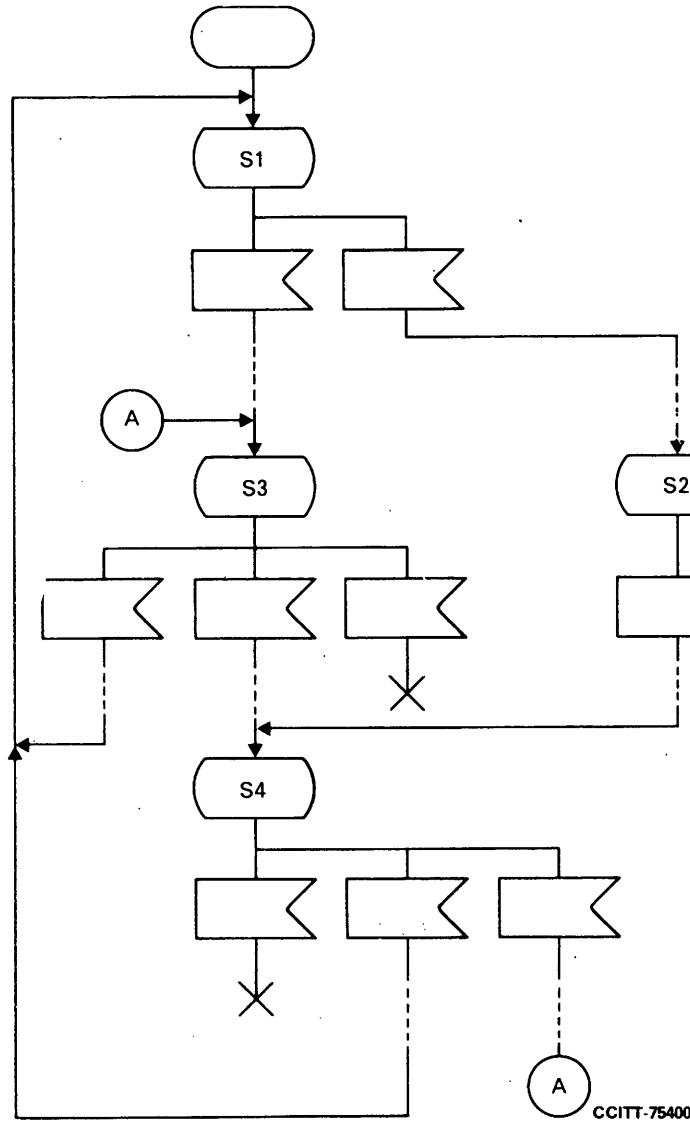


图 D—82

完全连接的流图

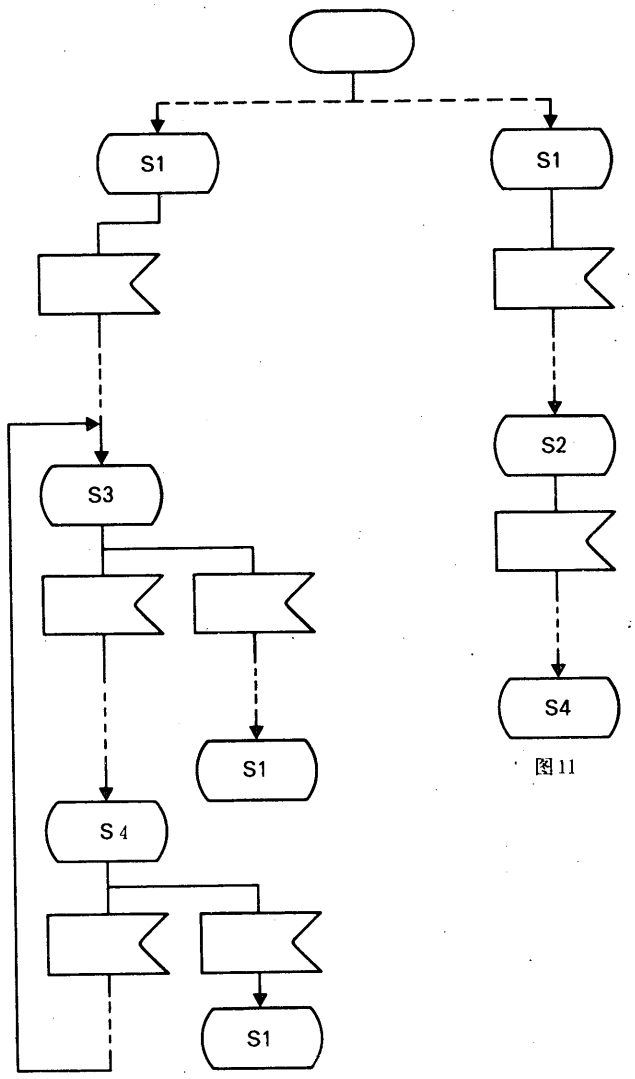


图 1

图11

图111

图 D—83

部分连接的流图

三种方法的比较

方法A 的优点在于考虑如何画一个图所花费的时间最少，还在于易于修改和易于检索信息。

方法B 的优点在于综合了整个过程，而方法C 能使进程的结构就该进程最重要部分来说符合预先确定的准则。

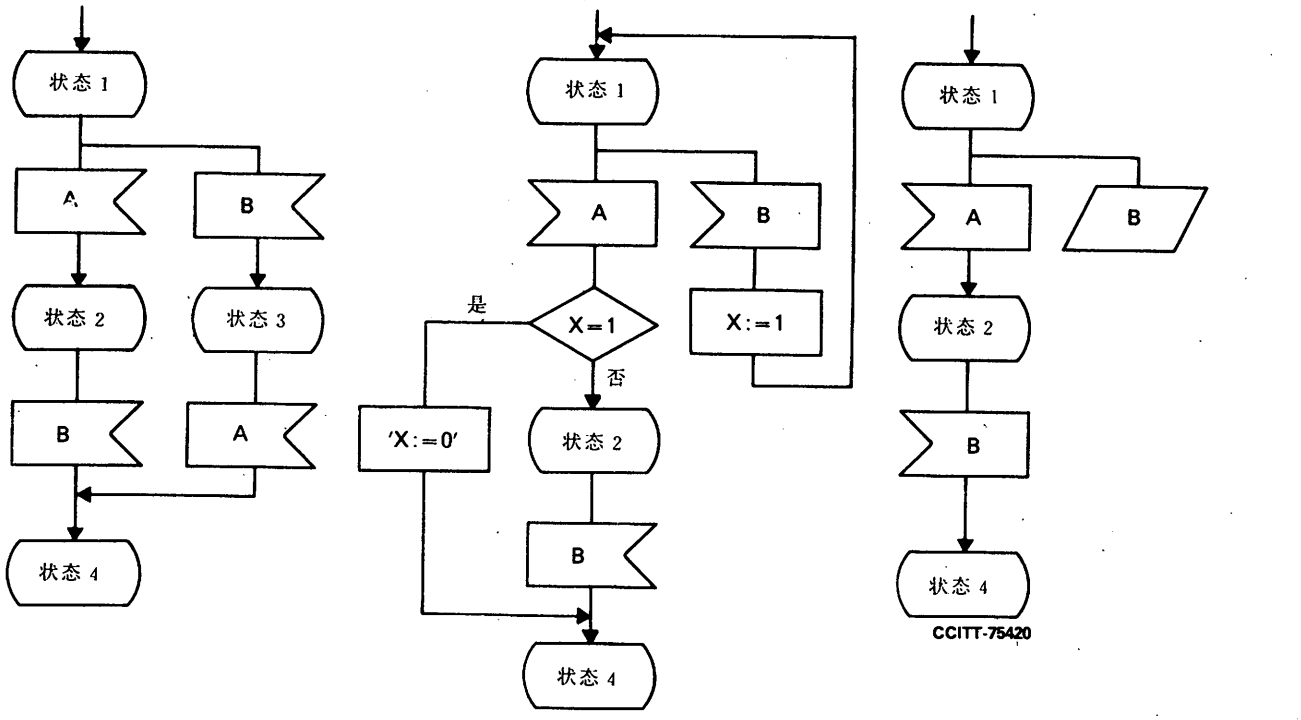
图的清晰度

因为SDL 图是作者和读者之间的一种通信媒质，所以很希望图是清晰的，图的清晰度不仅取决于安排图形的方法，而且也取决于进程的复杂性（例如状态、输入和判定的数目），这些因素本身又随结构而变化。

由于图总应易读、易理解，所以务必细心地实现一个合理的布局。

在探索最清晰的表示方法时，作者应当知道使用SDL 的各种方法。在图D—84的*a*）、*b*）、*c*）中所示的例子可以认为是表示某一部分进程的不同的方法。由于不执行任务或输出（人为地设置变量X或复位变量X 的任务除外）；这三个例子在逻辑上确实是相同的。

图*c*）被认为是最清晰的表示法。



a) 一SDL图的例子　　b) 含有一个判定的另一个SDL图　　c) 使用一个保存符号的又一个SDL图

图 D--84

三个图表示以随机次序接收输入A 和B 时，从状态 1 变为状态 4 的进程

D.6.3.6.3.3 进程图中的SDL/PR正文

在描述SDL/GR中的进程定义时，数据类型定义和变量定义是用SDL/PR语法描述的，该正文或是直接包含在图中，或是包含在一个分开的文件中。当正文不是直接包含在图中时，必须有一个对包含该正文的文件的引用说明，见图D—85和图D—86。

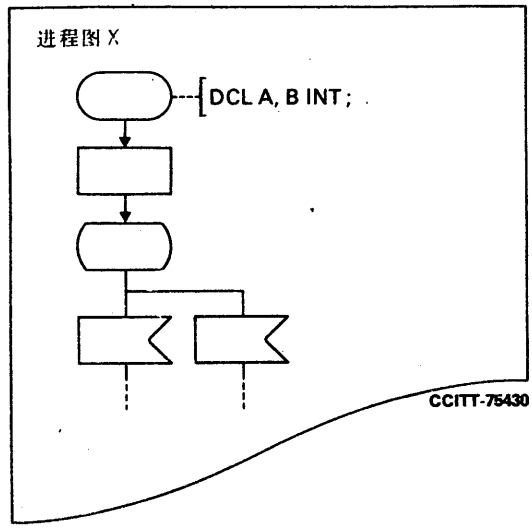


图 D—85
在进程图中包含SDL/PR正文的例子

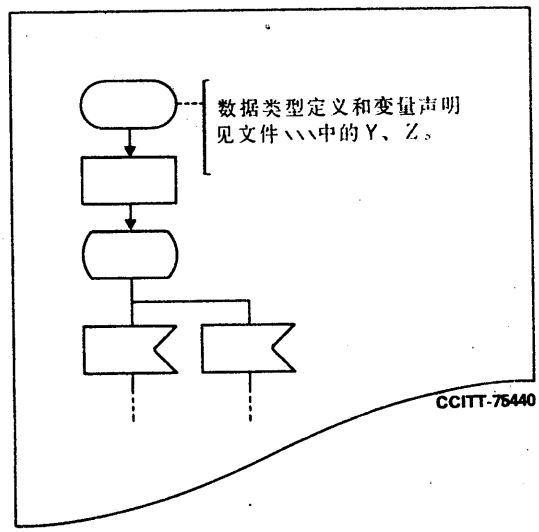


图 D—86
引用包含在别处的定义的例子

所选用的方法取决于所要包含的或引用的正文量的多少。包含大量的PR正文可能有损图的清晰度；另一方面，如果正文量少，则在图中包含正文可以更易读。

D.6.3.6.3.4 过程定义

当过程被定义为局部于进程时，该过程图可以被引用，也可以包含在进程图中。

当进程图中包含过程图时，重要的是使定义过程的流图与定义进程的流图分开，做到这一点的一种方法是在进程图内部形成章或节，见图D—87。

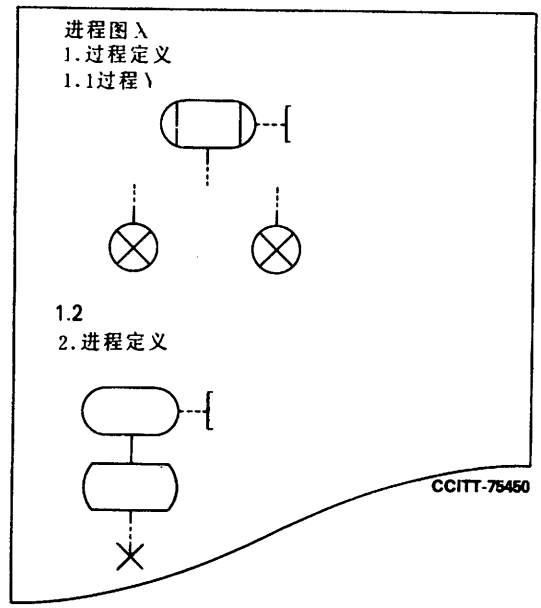


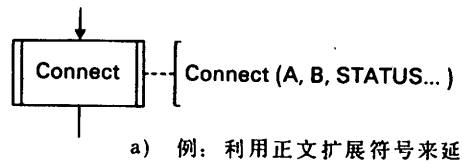
图 D—87

进程图中包含过程定义的例子

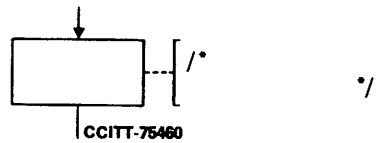
D.6.3.6.3.5 符号中的正文

通常与一符号相联系的正文应安排在那个符号的内部，然而由于正文数量和（或）符号大小的限制，这未必总能做到。

如果需要，正文可以安排在符号外部。若选择这个方案，必须使符号外部的正文与符号之间的联系很清楚，并且显然不是一个注释，做到这一点的办法是使用正文扩展符号，见图D—88所示。



a) 例：利用正文扩展符号来延伸形式正文



b) 例：利用‘注’一加入形式正文的机制

图 D—88

D.6.3.6.4　进程的创建

D.6.3.6.4.1　创建请求

一个新进程实例的创建是用创建请求符号来描述的。应当注意的是，被创建的进程实例只能在进行创建的实例所在的功能块内被创建。

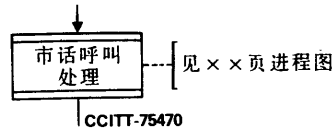


图　D—89
使用创建请求符号的例子

对被创建进程的进程图的引用，应作为一个注释插入图中，使图更加清晰。当需要实在参数时，应使用SDL/PR语法来提供。

D.6.3.6.4.2　起动

进程的起动符号描述创建一个进程实例时其行为的起点，进程的形式参数应与该符号相关联。

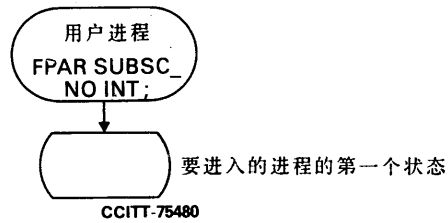
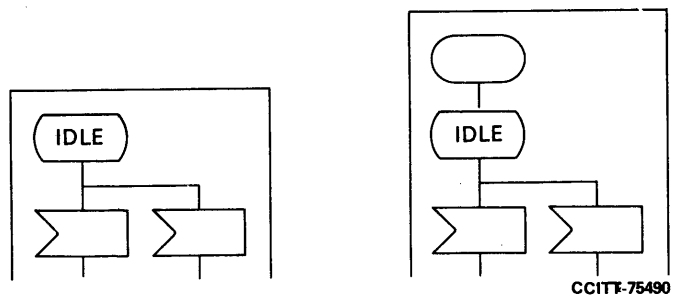


图　D—90
使用起动符号的例子

为了不使旧的SDL图无效，所需的起动符号可以隐含；这时我们假设它出现在进程图中的“第一”个状态符号之前，这里“第一”指的是出现在图的第一页顶部的第一个状态符号。



a)　图的样子　　b)　对图的解释

图　D—91
隐含起动符号

起动符号应作为一个图的第一符号出现，把它隐含在图的内部，可能会使读者产生混淆。

D.6.3.6.5 状态

状态用状态符号表示，并且有相关联的输入符号，还可能有保存符号。状态的一个例子如图D—92所示。

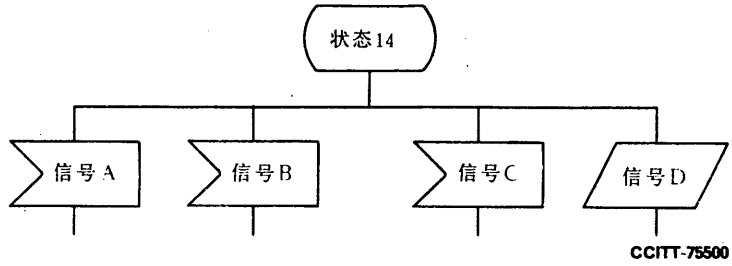


图 D—92

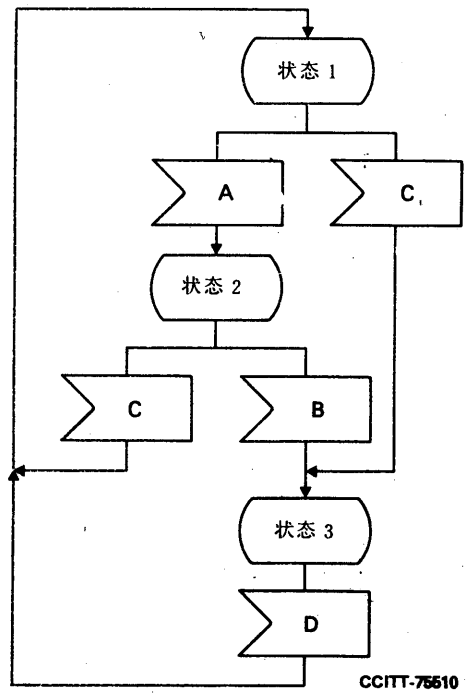
使用状态符号的例子

D.6.3.6.5.1 状态的重复出现和下一状态符号

为了简化作图或有助于更好的理解，在一个SDL图中相同的状态可以出现若干次。在这种场合，可认为该图与合并同一状态的所有重复出现而得到的图是完全等效的，图D-93和图D-94给出这样的例子。在图D—93 *b*）中，当一个状态符号处在下一状态符号的地位时，被用作为连接符连接到具有同一名字的主状态。在图D-94中，一个状态用重复的几个符号表示，而每个符号仅具有输入（或保存）的一个子集。

在图D—93中，*a*）和*b*）逻辑上是等效的。图*a*）中每个状态只出现一次，而图*b*）中每个状态重复出现几次。图*b*）中各状态有一个主状态符号，在那里指明了所有与它有关的输入（和保存）符号。图中凡是能从其它地点到达该状态处（作为一个跃迁的终点），都把它显示为一不带输入或保存的状态。一个从下一状态符号对该状态符号引用的注释将改善清晰度，特别当各个重复的状态符号出现在不同的页上的时候。

图D—94运用一个状态的重复出现来建立输入（和保存）的全集。图中示出该状态的每次出现仅附有其中的一个子集。



a)

图 D—93

一个完整的图

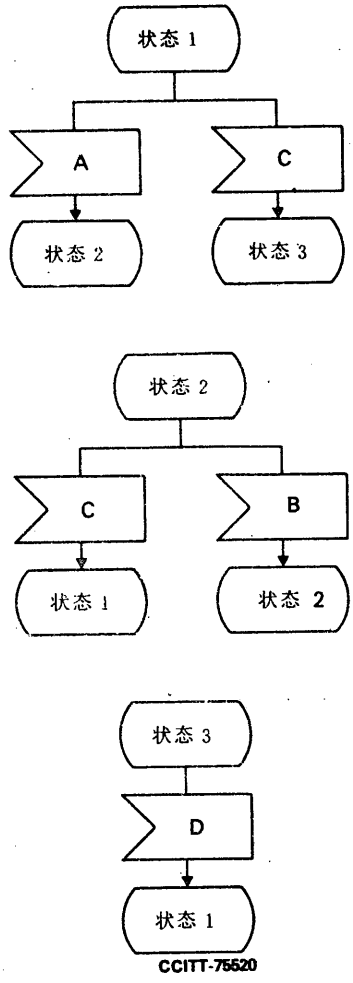


图 D—93*b*

同图*a*），但带有多个主状态和用作为连接符的后继状态

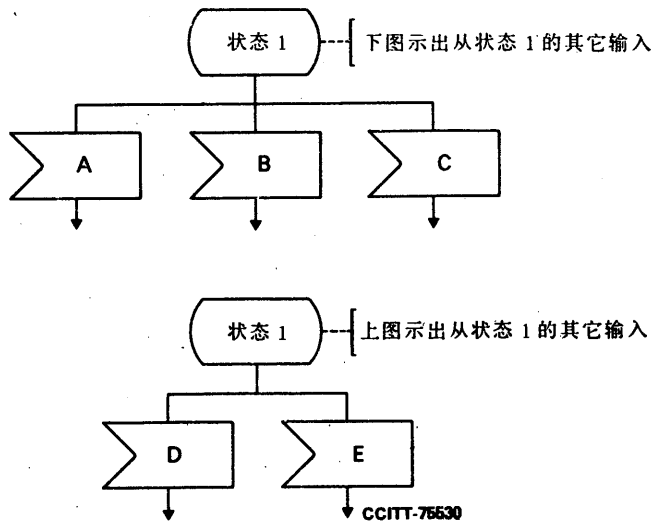


图 D—94

当不能从同一符号清晰地画出全部输入时，采用一个状态的重复出现

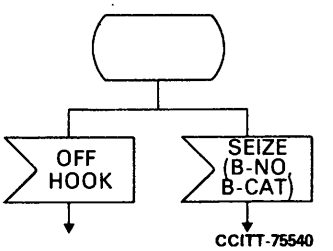
在状态有较多的输入或保存时，这种方法已成功地使用了，但如果读者未意识到有重复出现，就可能有对图形产生误解的危险。要避免这种误解，应当给只示出输入（或保存）的子集的那些状态加上注释，该注释指出具有有关输入（或保存）的另外一些状态，如图D—94所示。

利用状态的重复出现，可以方便地把读者的注意力集中在某些方面（如处理信号的正常顺序），而把其它的方面推后到另外的页（如告警情况处理）。

D.6.3.6.6　输入

输入符号连接到一状态符号，表示输入概念。

当用信号（将被该输入接收）传递数据时，用SDL/GR语法在输入符号中给出需要共享信号中的值的局部变量表。如果没有数据可传递，该表可以省去；如果传递了数据而表又被省去，则应解释为在接收时数值将被丢掉，见图D—95的例子。



注－在引到输入符号的线上不应有箭头

图　D—95
使用输入符号的例子

如果在一个状态后面有若干个信号可以引起同一跃迁，那么所有的输入可以组成一个信号表，用同一输入符号来表示。在信号表中出现所有信号的名字（和它们的变量表），并用逗号隔开。

在早期的SDL文本中，外部输入和内部输入之间有一差别，即外部输入接收从功能块（包含此进程的功能块）外部发送来的信号。由于从功能块内部和外部始发的信号并无语义差别，这些概念已经取消了。早期用来表示内部输入的专门的符号，如果在图中发现，就按一个输入符号进行解释，见图D—97。

D.6.3.6.7　保存

保存符号表示一个状态的保存信号表。要保存的信号应当在符号内写出名字。如果有若干保存符号连接到一状态符号，那么它们将被看成是一个保存符号，其中所包含的名字表是所有符号中提到的名字的并集。

名字表中的名字应当用逗号隔开。此外还有一种使用“星号*”的简化表示，将在D.6.3.6.21中加以说明。

D.6.3.6.8　输出

输出用来发送一个信号并给它提供待传送的值。输出用输出符号来表示。待传送的值表用SDL/PR来规定，此表最好应包含在符号中。该符号也必须包含接收信号的进程实例的地址，见图D—99中的例子。

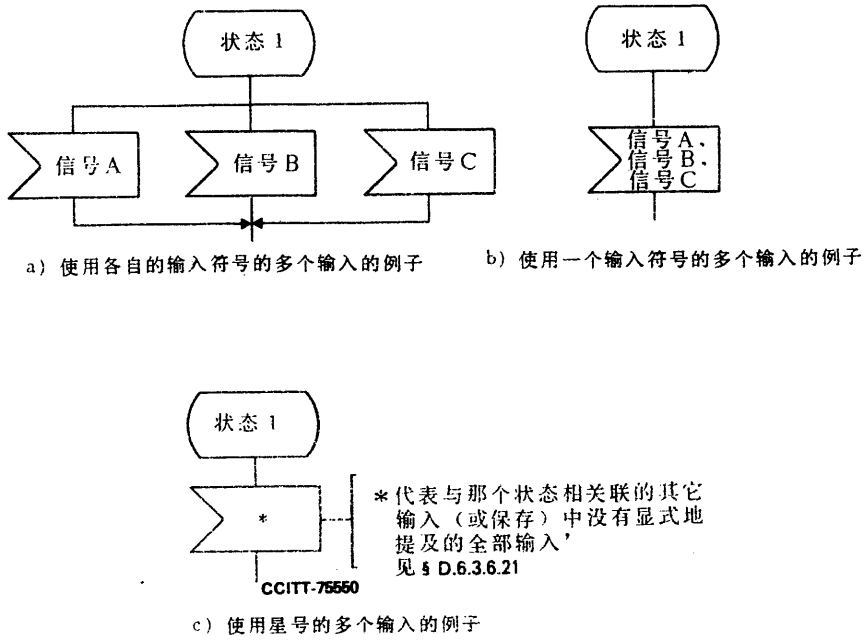


图 D—96

多个输入的可供选择的表示法

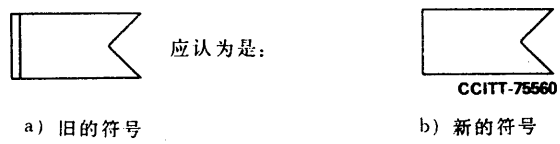


图 D—97

外部输入和内部输入的等效

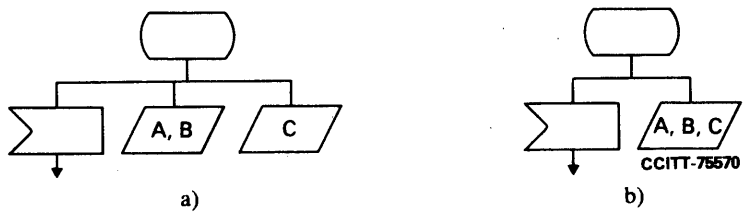
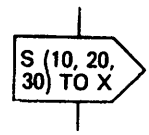


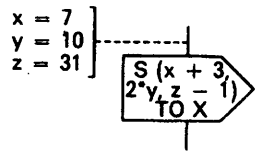
图 D—98

同一保存表的不同表示法



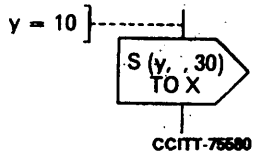
注-信号S 有三个与它有关的值，10、20和30。

a)



注-在解释S 时，X、y和z（在该例子中）分别有值 7 、10和31，S 发送的值为10、20和30。

b)



注-在解释S 时，y（在该例子中）有 值为10，S 发送值10、一个未定义的值和30。

c)

图 D—99

带有有关数据的输出

在较老的SDL 文本中，和输入一样，在外部输出和内部输出之间有一个差别。在图中发现的任何旧的内部输出符号应解释为一个输出符号，见图D—100。

a）旧的符号　　b）新的符号

图 D—100

内部输出和外部输出的等效

D.6.3.6.9　允许条件

允许条件用允许条件符号表示，该符号是输入符号和表示布尔条件的图形语法的组合。

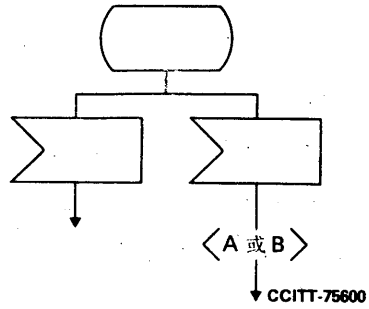


图　D—101

这个组合应被认为是一个符号，这就是为什么在连接两部分的线上不应有箭头的缘故，见图D—101。

D.6.3.6.10　连续信号

连续信号用连续信号符号表示。
包含的表达式应是布尔表达式。

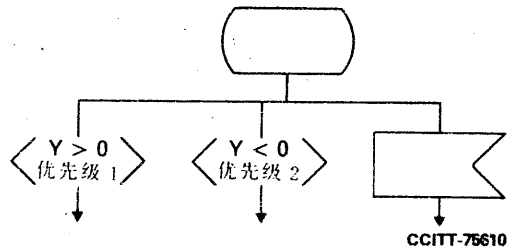


图　D—102
允许条件的例子

如果有一个以上的允许条件与一状态相关联，那么优先子句是必须要有的。

D.6.3.6.11　任务

任务要用一个任务符号来表示。任务的描述或者用SDL/PR正文，或者用非形式正文，它们最好应包含在任务符号中。

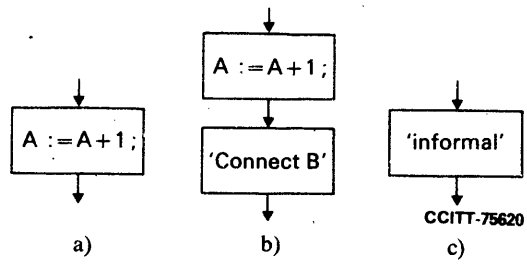


图　D—103
任务符号的例子

在既包含形式语句又包含非形式正文的图中，非形式正文应当用单引号括起来。

D．6.3.6.12　判定

判定用一个判定符号表示。

与某一判定有关的问题的正文放在该判定符号内，问题不必包含问号，仅需写出待求值的表达式，如no、A ＋B 、Z 。判定符号必须有两个以上的分支，问题的回答直接安排在相应分支的右侧或顶部，各分支合在一起必须包含所有可能的回答。可以用答案的值（如17、12）或用一操作符作前缀（如＞18，≠ 34，＝25）来表示各个回答。一些可能的格式表示在图D －104中。

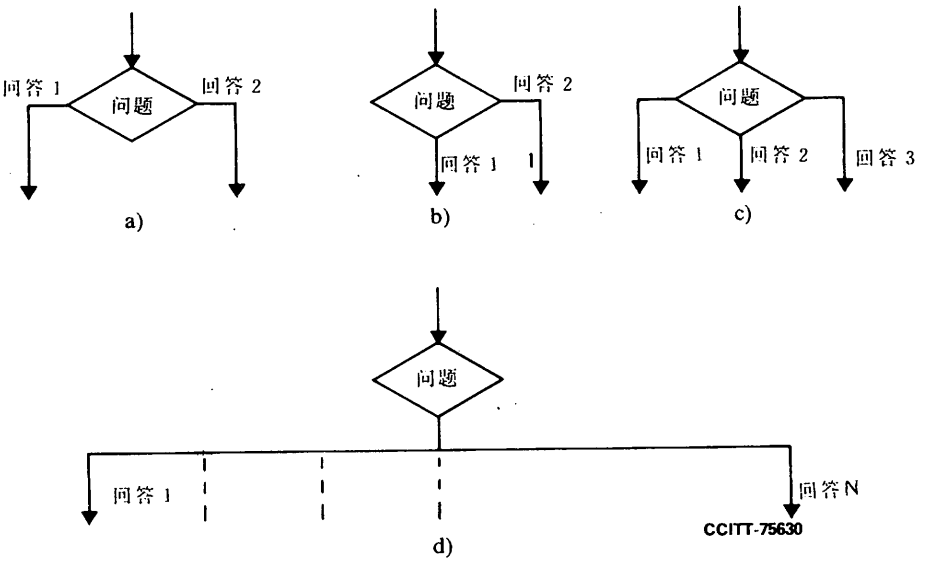


图　D －104

判定符号的一些可能的格式

许多问题和一特定的条件有关，在该条件下可能的回答仅有“是”或“否”,也就是 TRUE 或FALSE。

例如：

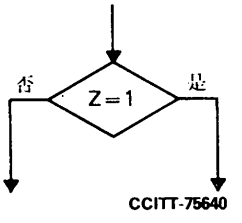
B 用户忙

电话机“挂机”

数字 2 被接收

Z ＝ 1

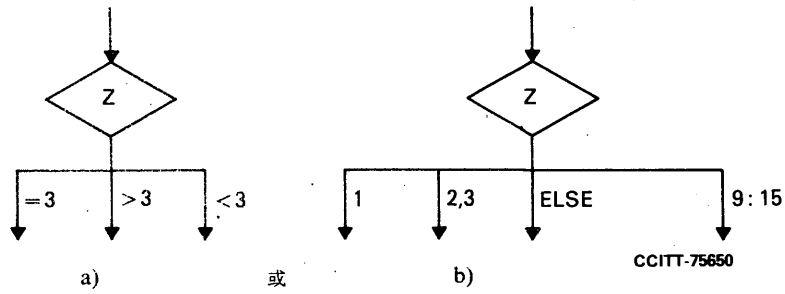
图D －105给出一个这样的例子。



图D －105

一个简单的二向判定的格式

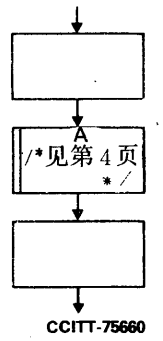
在有两个以上回答的情况下，图D－106的一些格式是合理的，它假定2是一个正整数，图中的ELSE回答是指不包含在其它回答中的所有回答，也就是与这组值（1、2、3、9、10、11、12、13、14、15）不同的值。



图D－106
问题和回答的格式

D.6.3.6.13 宏

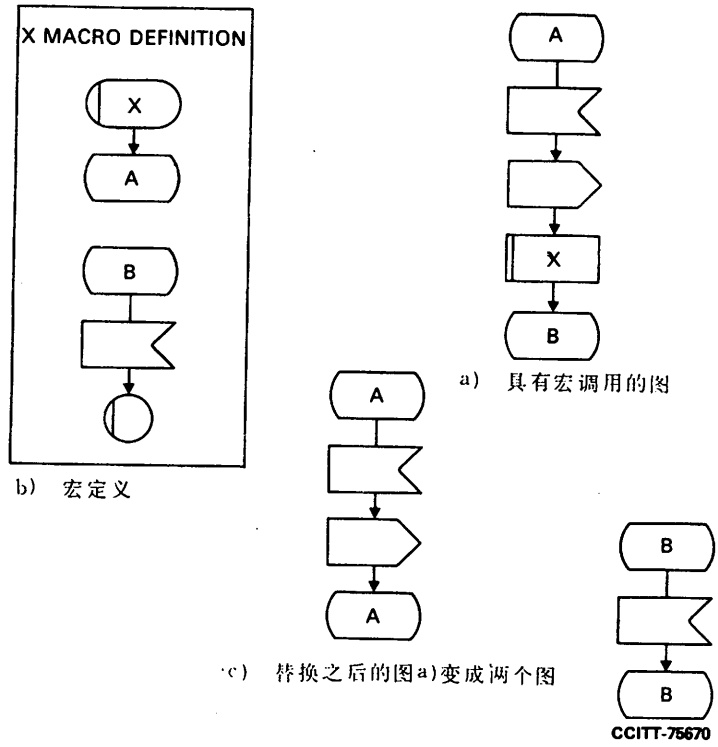
宏符号可以安排在图中任何地方。图的正确性只有在宏符号被它的定义的内容替代之后才能确定。然而，宏最适宜于用在进程图中能出现任务符号的地方。



图D－107
宏符号的使用

当使用宏时，在宏符号中指出应在何处找到定义是很有用的（图D－107给出了一个例子）。

在使用宏时，用户应当意识到宏可能隐藏了某些副作用。由于宏通常出现在任务符号的位置上，读者可能会采用任务的语义。

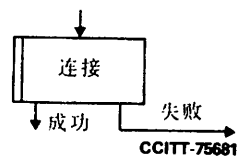


图D－108

不好的宏的使用的例子

图D－108表示一个宏调用的置换如何改变了图中的序列。当宏含有状态时，请注意：包含宏引用的跃迁可能被分解成若干个跃迁。

在宏包含若干入口处和（或）出口处时，它们应当用标号明显地加以标识，标号最好安排在走出流线的右边。

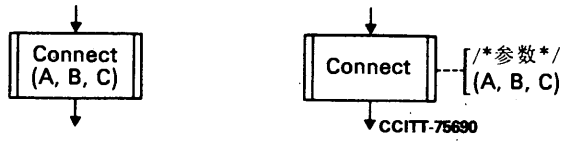


图D－109

有若干出口处的宏

D.6.3.6.14 过程

过程调用符号表示对一个过程的调用。为调用提供的参数最好应放在符号中，如果这不可能或不实际，那么可以把它们安排在旁边的一个正文扩展符号中。



图D－110

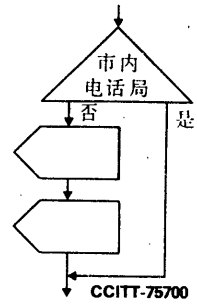
使用过程调用符号的例子

过程调用符号可以出现在能安排任务符号的任何地方。

过程是一个工具，可用来减少图的复杂性，也便于利用标准的解法。

D.6.3.6.15 任选

任选符号用一个图来描述若干可供选择的进程行为。如果若干进程行为只有小的差别，这是一个有用的功能。该符号包含一个任选表达式，这个表达式应当在进程被解释之前就能求出值，也就是说，它不是一个动态条件。这个表达式还应使它的值能对应于一组离散的选择对象，能用这些值对走出流线作标记。

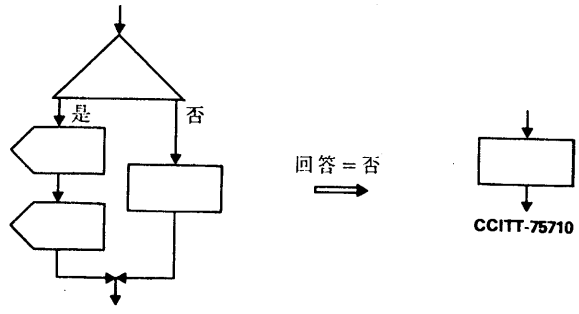


图D－111

使用任选的例子

任选对象最好应置于走出流线的右边。

要等到全部任选表达式被求值以后才对图进行解释，然后图就被看成是似乎已删去了不能到达的全部分支。



图D－112

任选的解释

D.6.3.6.16 连接符

概述

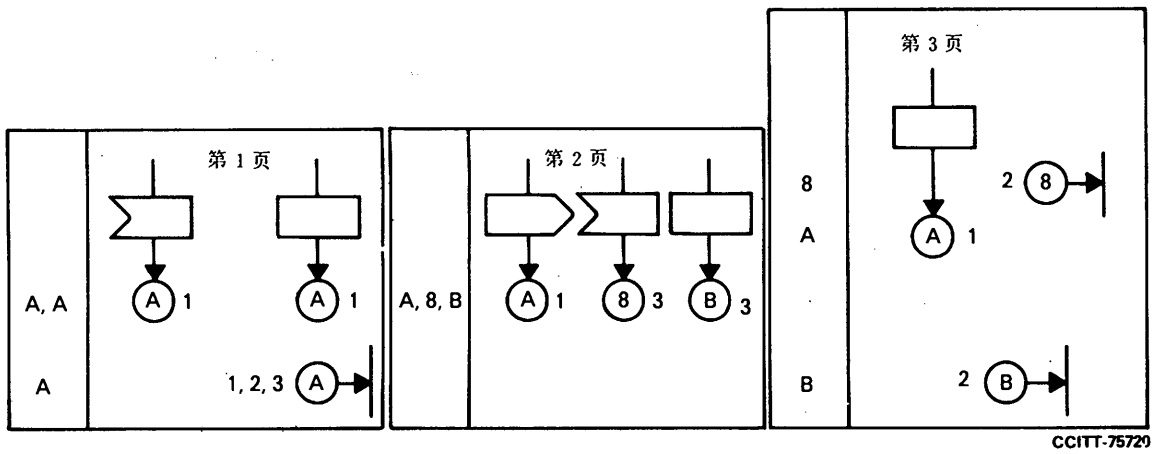
在画一个大的SDL图时，由于地方不够，也可能需要把图进行分解，连接符就是用于这个目的的。

连接符也可用来避免流线的交叉，流线的交叉会使图多少有点不清晰。通常SDL图的可取画法是从一页的顶部画到底部。

使用连接符表示隐含的流线

任何流线可以被一对相关联的连接符断开，设定其流向是从出连接符到入连接符。每个连接符有一个名字，相关联的各连接符名字相同，对每个名字只存在一个入连接符，而可以有一个或多个出连接符。

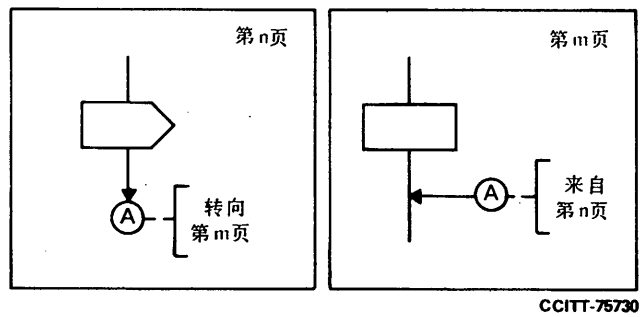
最好是在每个出连接符附近给出相应的入连接符的参考页码，而在每个入连接符附近给出相应的出连接符的参考页码，见图D—113。也希望在页的边上有一连接符的参考栏，用来指明连接符的水平位置和顺序。



注－另一个同样可接受的画参考页码的方法是使用注释，如图D－114所示。

图D－113

画参考页码和连接符参考的方法



图D－114

画参考页码的另一种方法

连接符只能用于同一进程内的连接。

如存在若干主要流程路径，则除了在各页的顶部和底部外不要用连接符把它们断开，以使把注意力集中于主流程。

对SDL图内的次要流程路径则能用一出连接符断开，相关联的入连接符可画在别的合适的页上，比如主要流程路径结尾的下一页。

D.6.3.6.17 发散和汇聚

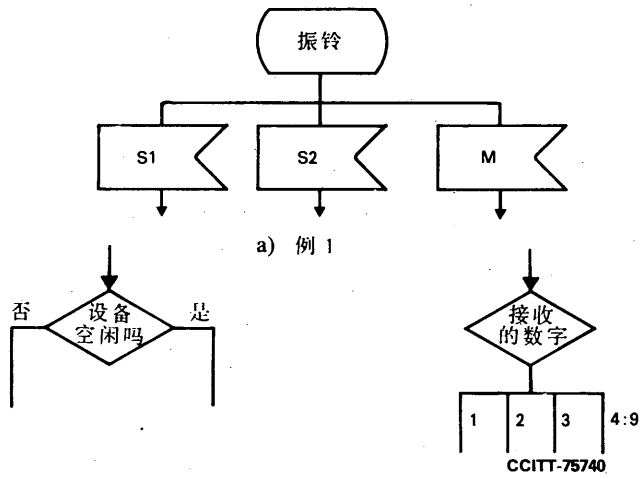
发散

在SDL图的一次跃迁内发散只可能发生在：

a）一个状态符号和与它相关联的输入和保存符号之间，或，

b）紧接在一个判定符号之后。

发散的一些例子如图D-115 所示。



图D-115

发散的例子

汇聚

汇聚点不能在一状态符号和一输入或保存符号之间出现，但可以在SDL图中任何其它地方出现。

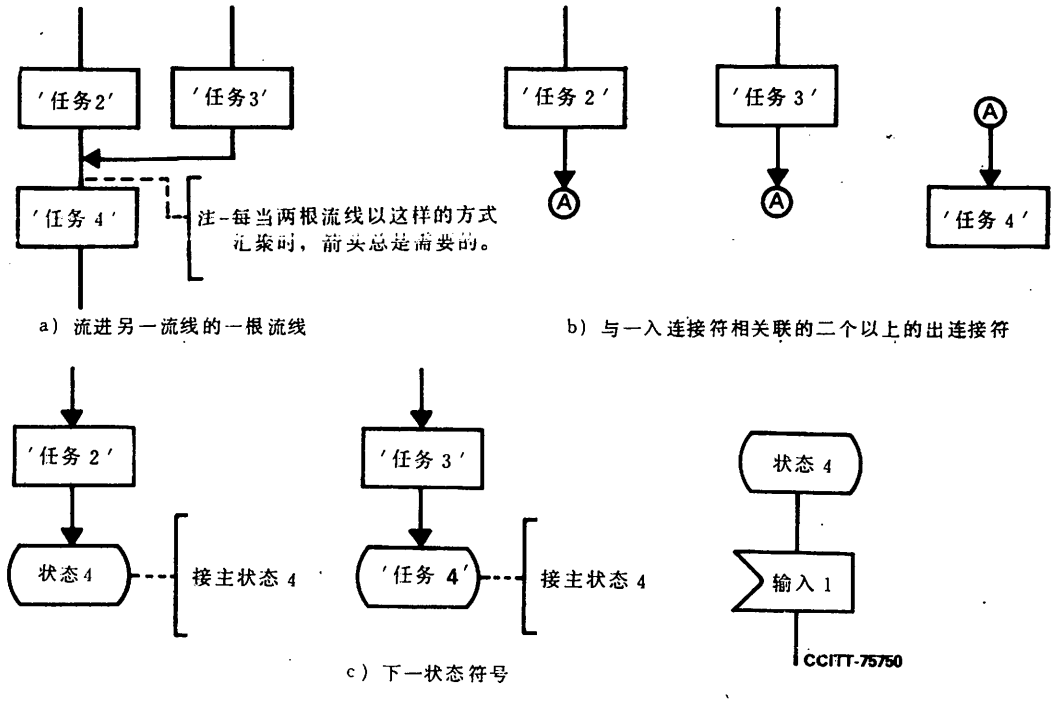
汇聚可以以图D-116所示的任何一种方式出现。

在一连串符号和与之相关联的正文在同一SDL图中反复出现的场合，运用汇聚能够减少图中符号的数目，如图D-117所示。

D.6.3.6.18 注释

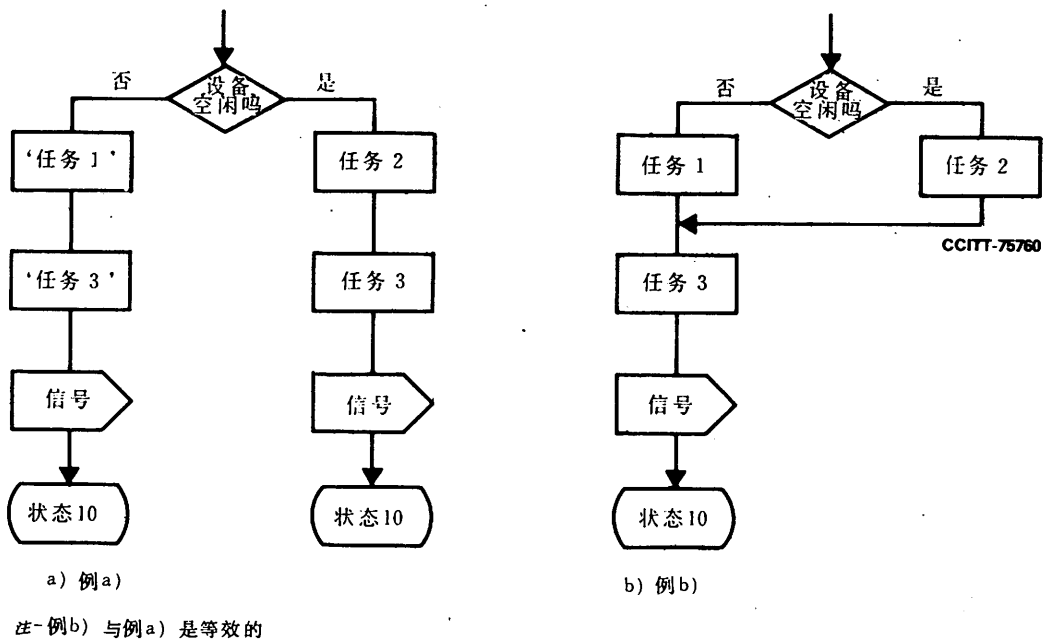
可以把注释加入SDL图中，使图的某些部分易于理解。注释是用来帮助读者的，它对SDL图的解释不起作用，因此注释应当对图的语义既不矛盾又不增加任何东西。

注释通过一单方括号附加到一流线或-SDL 符号上，该方括号用一根虚线连接到该流线或符号，见图D-118。



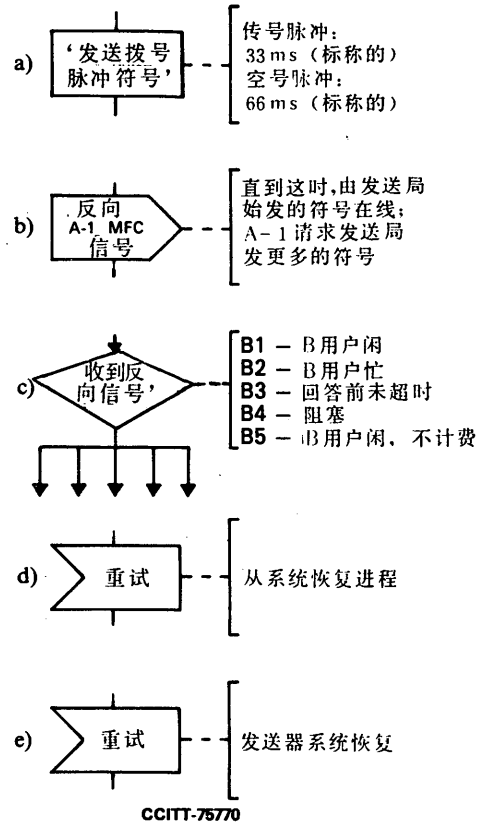
图D－116

汇聚的例子



图D－117

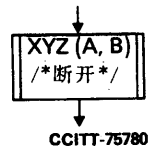
汇聚的使用



图D－118

使用注释的例子

注释也可以使用SDL/PR语法（/ * ……* / ）插入，如图D－119中所示。



图D－119

在SDL/GR中使用SDL/PR注释语法的例子

D.6.3.6.19 正文扩展

通常与一图形符号相关联的正文应安排在那个符号内，然而这往往是不可能或不实际的。一个替代的方法是把正文安排在一正文扩展符号内，而将该扩展符号再连接到相关联的符号，见图D—120 。

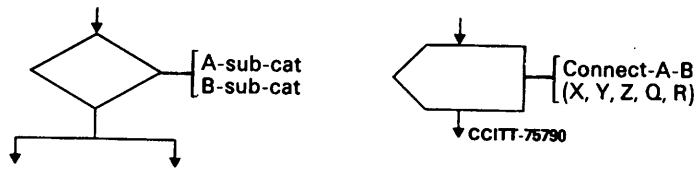


图 D－120

使用正文扩展符的例子

由于正文扩展的语法与注释的语法相类似，因此把连接线画成实线很重要，这样就和注释符号的虚线区别开了。

如果在正文扩展符号中使用非形式正文，那么该正文应当用单引号括起。

D．6、3、数据

通常，SDL /GR中的数据结构采用SDL /PR 语法。当在任务、输出、判定或输入中使用数据时，要将相应的SDL /PR 正文与符号相关联。

数据类型定义和变量声明应包含在单独的文件中为进程图所引用，也可包含在进程图中。这些定义应当与该图的起动符号相关联。

图D－121

在SDL /GR 进程图中的数据定义举例

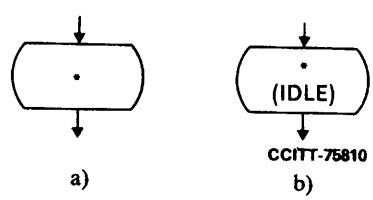
运用正文扩展符号使数据定义与该符号相关联，如图D－121 所示。

D.6.3.6.21 简化符号

为了使图更易读和避免长的名字表，引入了下列简化符号，所使用的符号是星号（*）和短横（－）。一般“*”具有“全部”或“除…之外全部”的意思，而“－”具有“相同”的意思。

简化符号可用在STATE、INPUT、SAVE及NEXT－STATE 等符号中。

在状态符号中，星号（*）可以单独使用，或者和一用括号括起的状态名字表一起使用。

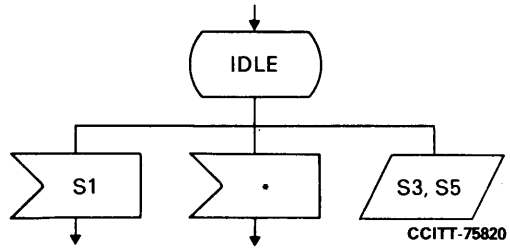


图D －122

在状态中的“*”符号举例

图D－122中的星号（＊）对a）应解释成“全部状态”，而对b）应解释为“除IDLE外的全部状态”。这里该简化符号起到了状态符号重复出现的功能，（见§D．6．3．6．5）。

在输入符号中，星号（＊）只能用在与一状态连接的一个输入符号中（假定它不出现在与同一状态连接的一个保存符号内）。

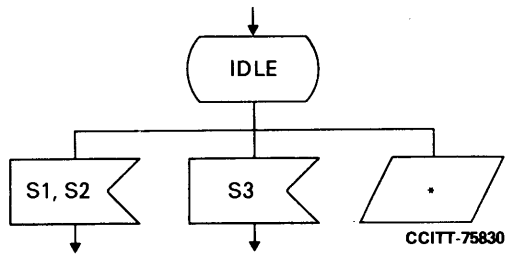


图D－123

星号（＊）在输入符号中的使用

在图D－123中星号（＊）的解释是“除了在其它输入或保存中所列的信号，即S 1、S 3和S 5以外的所有信号”。

星号（＊）在保存符号中的使用是类似的，它只能出现在与一状态连接的多个保存符号中之一，並且它不得再出现在与同一状态连接的任何输入中。

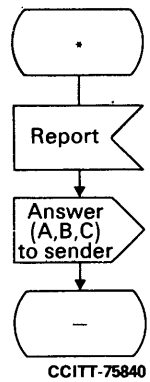


图D－124

星号（＊）在保存符号中的使用

在图D－124中星号（＊）的解释是“除了S 1、S 2和S 3之外的所有信号”。

在下一状态符号中，短横“—”可用来表示“和引起跃迁的状态相同的状态”。



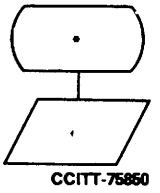
图D－125

短横（－）在下一状态中的使用

图D－125的解释是在进程的任一状态中，信号“REPORT”都能被接收，接收动作将引起信号“ANSWER”的发送，並且跃迁将终止于一个状态，此状态就是引起这个跃迁的状态。

和重复出现功能相配合，这些简化符号有很强的表达能力，应注意，用“*”和“－”简单地加到一个图上，可以以意想不到的方式改变一个图的意义。

作为一个例子，将“*”加到如图D－126所示的图上，会使得在全部状态中都不再有隐含的跃迁了。



图D－126

此图的一个效果是：在整个图的任一输入、状态或保存符号中都不允许用“*”了

D.6.3.6.22 状态图形

D.6.3.6.22.1 状态图形任选

如果我们选择状态图形任选，那么最好把每个状态图形看成是由(图D－127*a*)所示的三种类型的元素所构成。

这三种类型又往往组合起来，形成一复合图形元素，该复合图形元素能看成是与状态图形其它部分相分离的。例如(图D－127*b*）的含义是一个多频信号接收器正在等待一正向信号。

注意为了监视一个进程的时间，所建议的图形元素符号包含了有关的输入变量*ti*。

1）图形元素，如 ⟩□ （指一信号接收器）；

2）输入变量，如$\overline{f}$（指一正向信号尚待识别）；和

3）限定正文，如MFC（指多频代码）。

a）状态图形的内容

b）复合图形元素

图D－127

图形元素的构成

D.6.3.6.22.2 输入变量

输入变量是表示与进程相关联的那些条件的一个有用的方法。如果条件改变，进程将引起一次跃迁。输入变量应采用小写字母，以便于与限定正文区别开（限定正文应采用大写字母）。（输入变量改变相当于输入信号改变，这会使进程脱离当前的状态；而限定正文改变並不相当于输入信号改变。）

D.6.3.6.22.3 限定正文

限定正文应大大地缩写，並且尽可能置于相应的图形元素内，这时哪个图形元素被限定就十分明显了。

D.6.3.6.22.4 状态图形的完整性

应当把足够的图形元素分配给每个状态图形，以便指明：

a）在当前状态期间，该进程正在考虑什么资源。资源的例子是：交换通路、信号接收器、信号发送器及交换模块；

b）当前是否有一个或多个计时器正在监视该进程；

c）在该进程涉及呼叫处理的情况下，当前用户计费是否在进行，以及在此呼叫的状态期间哪一个用户负担费用；

d）分配给该进程的设备的现行类别（或状态），这些信息在何处影响该进程的行为；

e）在当前的状态期间被进程监视的那些输入变量的状态。

D．6．3．6．22．5 例子

在图D－128中，用状态图形说明了应用上述原则的一个例子。在该状态期间将可见到：

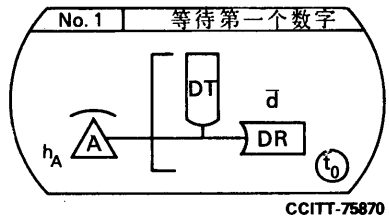
a）进程所考虑的资源由一用户话机、一数字接收器、一拨号音发送器及连接这些设备的交换通路所组成；

b）一个计时器T_0（它的现行状态是t_0）当前正在监视该进程；

c）没有进行对此呼叫的计费；

d）用户已被标识为A用户，而没有把其他类别的信息考虑进去；

e）用了下列一些输入变量状态：h_A（即话机摘机），*d*（即数字接收器正在等待一个数字）和t_0（也就是监视的计时器T_0正在运行）。



图D－128

在一呼叫处理进程中一个状态的例子

D．6．3．6．22．6 具有图形元素的SDL图的一致性检验的特性

如果遵循D.6.3.6.22.4*e*）中所述的原则，那么只要查看一下状态图形中指明的输入变量，总能找出使进程脱离此状态的一组输入。例如，通过查看图D－128的状态图形，人们能够找到三个不同的输入会使进程离开这个状态：话机状态转换到挂机（输入$\bar{h}_A$），一个数字的到达（输入*d*），或计时器T_0终止计时（输入t_0）。这样，当按很复杂的SDL规格来实现系统时，某些“意想不到的跃迁”就可以避免了。

很明显，图形使读者感到更加简洁明了，且在一定意义上使读者看到更多的信息；但同时人们又必须很仔细地查看图形，以便得出一组在跃迁中执行的正确的动作。

此外，只凭查看图形不可能知道一个动作是通过一个输出执行呢，还是通过一个任务执行（§D.6.3.6.22.8）。

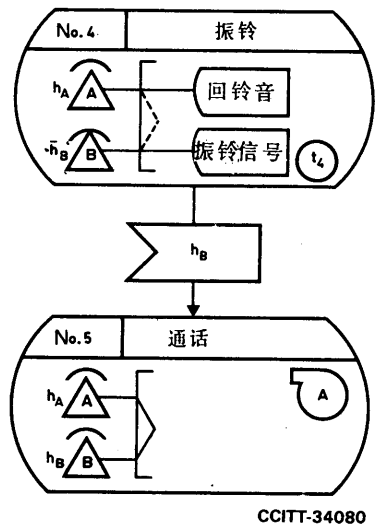
D．6．3．6．22．7　功能块边界符号的使用

在功能块边界符号外部的图形元素代表不受给定进程直接控制的元素，而在功能块边界符号内部的图形元素代表受给定进程直接控制的元素。例如，在图D－129中部分地表示了的呼叫进程能够接通或断开振铃电流，以及启动或停止计时器 T4，但它不能改变任何一个用户的话机状态。

在根据具有图形元素的SDL 规格进行逻辑设计时，只有画在功能块边界内的那些图形元素才会影响在跃迁序列中执行的处理动作。画在功能块边界外的复合图形元素通常要包含在一状态图形中：

a）因为它们指明了在给定的状态期间进程必须监视的输入变量；以及（或）

b）使图更易于理解。



图D－129
两个状态之间的一次跃迁的例子，在这里状态图形的差别隐含了全部的处理动作

D.6.3.6.22.8　任务或输出

把一个处理动作解释为一个任务还是一个输出，有时看起来是任意的。事实上，把功能块边界内的某个图形元素的出现或消失判成是一个任务还是一个输出，只有通过查阅进程信号表才能解决。

D.6.3.6.22.9　计时器符号的使用

不管是否使用图形元素，在计时器超时的时候，被监视的进程的中断总要用一输入来表示。

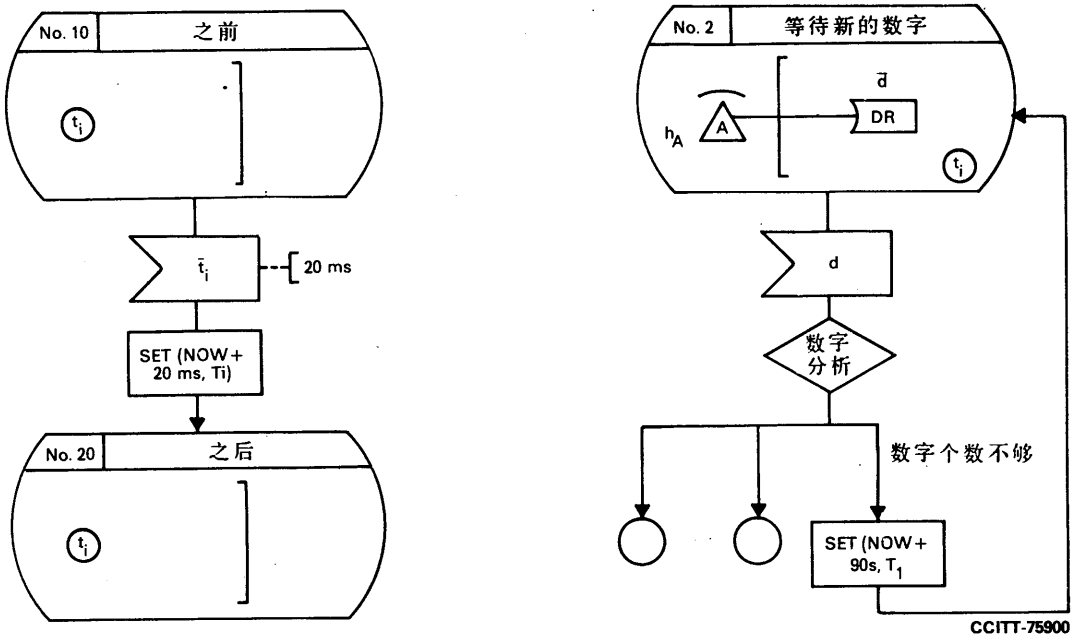
在一状态图形中，计时器符号的出现意味着在该状态期间有一计时器正在运行。按照在建议中叙述的一般原则：启动、停止、重新启动和计时器超时是按下列方法用图形元素表示的：

a）如要表示出在一给定的跃迁期间一个计时器的启动，则该计时器符号应出现在该跃迁结束处的状态图形中，而不是出现在该跃迁启动处的状态图形中。

b）如要表示出在一次跃迁期间一计时器的停止时，则该计时器符号就应该出现在该跃迁启动处的状态图形中，而不是出现在该跃迁结束处的状态图形中。

c）如要表示出在一次跃迁期间一计时器重新启动，则应在该跃迁中安排一个显式的任务符号（图D－130中举了两个例子）。

d）一特定计时器的超时可通过与一状态（在该状态中其状态图形包含相应的计时器符号）相关联的一个输入符号来表示。当然如果需要的话，多个计时器可以同时监视同一进程，如图D－131所示。



a）随着计时器到时，计时器的重新启动

b）计时器尚未到时时，计时器的重新启动

注－每个计时器Ti有两个互斥的条件t_i和$\bar{t}_i$。

图D－130
计时器重新启动举例

D.6.3.6.23　　**辅助文件**

为了帮助阅读和理解大的进程图，（给出概述的）辅助文件是很有用的。这些文件是非形式的，仅仅用来给出概述并作为SDL图的“导引”文件，用户可自由选择它们自己的合适语法。

在本章中给出两类辅助文件的例子；状态概览图和状态/信号矩阵。

D.6.3.6.23.　　状态概览图

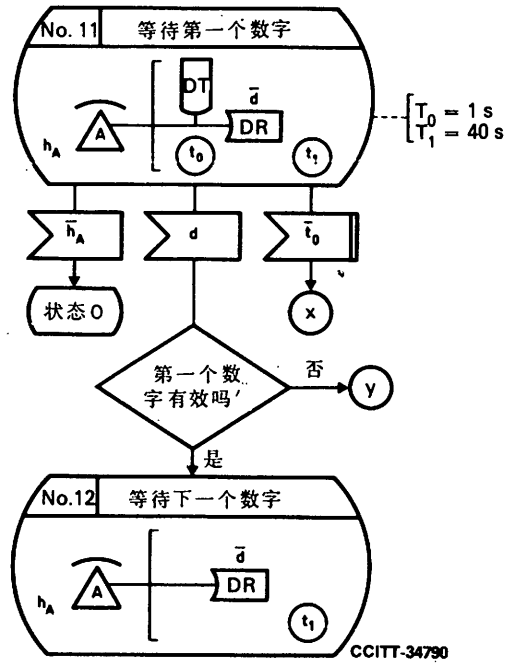
状态概览图的目的是要给出一个进程的各状态的概览及状态之间可能有何种跃迁。因为目的是给出概述，不重要的状态或跃迁可以省略。

该图由各状态符号、表示跃迁的有向弧、以及任选的启动符号和停止符号所组成。

状态符号应包含涉及到的状态的名字。该符号可以包含若干个状态名字，并可使用一星号“*”作为代表全部状态的符号。

对每个有向弧可给出引起跃迁的那个信号（或一组信号）的名字。

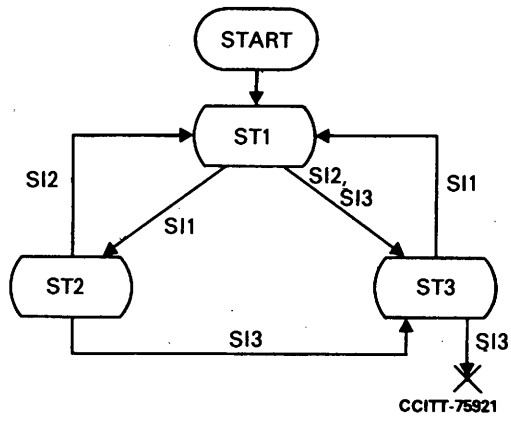
启动和停止符号的使用是任选的。



计时器T_0监视第一个数字的到达，随后从呼叫中除去拨号音，且计时器T_0停止。计时器T_1继续监视足够的数字个数的到达，以便为该呼叫选择路由。

图D－131

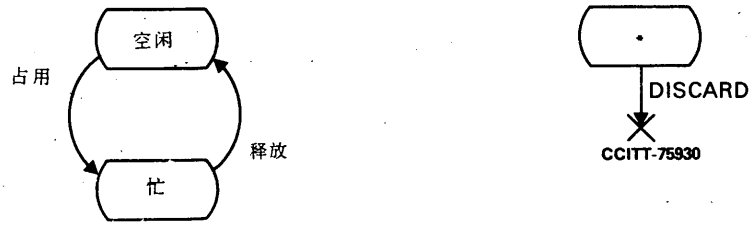
在同一状态中使用两个监视计时器的例子



图D－132

状态概览图举例

一个进程的状态概览图可分解成若干个图，每个图涉及不同的方面，例如“正常情况”,故障处理等。



图D－133

分开的状态概览图举例

为了结构良好，易于阅读，状态概览图最好应：

——符合正常的阅读方向—从顶向下，从左到右；

——篇幅不超过一页。

D.6.3.6.23.2 状态/信号矩阵

状态/ 信号矩阵可用作为一个大进程图的导引文件。它给出图中何处可找到某个状态和某个信号的组合。

此图由一个二维矩阵组成，该矩阵在一个轴上标以一个进程的所有状态，而另一轴上标以一个进程的所有有效的输入信号。在每一个矩阵元素中给出一个参考处，以便找到给定的状态—信号组合（如果存在的话）。

STATES / SIGNALS	IDLE	BUSY	BLOCKED	------
OFFHOOK	P5	–	–	
ONHOOK	–	P6	–	
SEIZE	P7	P8	–	
BLOCK	P6	P6	–	

CCITT-75940

图D－134

状态/信号矩阵举例

矩阵可分解为包含在不同页上的子矩阵,参考处应是在文件编制中用户所通常使用的参考处。

最好各信号和各状态应分群地组合在一起，以便矩阵的每个部分包括进程行为的一个方面，例如“正常情况”、“异常处理”、“维护部分”等；全应在不同的子群中找到。

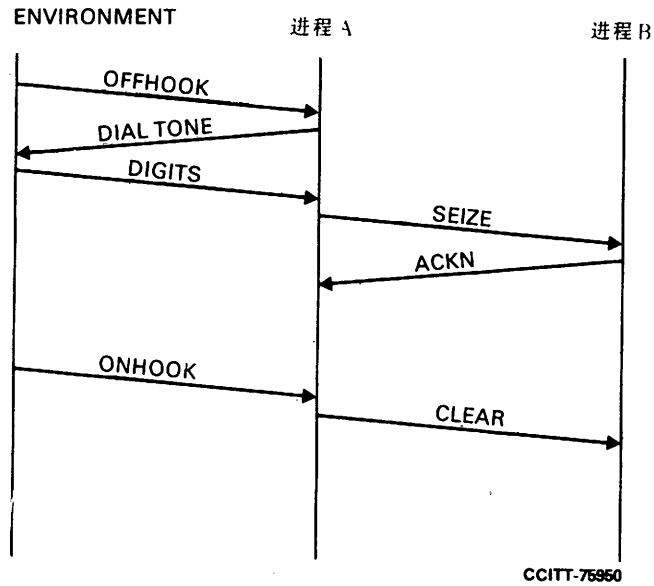
D.6.3.6.23.3 列线图

列线图可用作为相互作用图的一个补充，以表示在一个或多个进程和它们的环境之间容许交换的信号序列。

采用列线图是为了对系统各部分之间的信号交换有个概括的了解。可以用它表示信号交换的全过程或部分过程，这取决于要突出的是什么方面。

图中的竖线表示环境（通常安排在图的边上）和各个进程或功能块。

它们的相互作用是通过一组斜直线来表示，每一根斜线表示一个信号或一组信号，当用一根线表示向两个方向来往传送的一组信号时，建议用两端带有箭头的水平线来表示（例如“建立连接”）。



图D－135

列线图的例子

为了使人们能看清交换的是一组什么信息，可以在每根列线旁加注解，每根线旁应注明所包含的信息（信号名字或过程）。

在一根线的到达处可以安放一个判定符号，以指出如果指定的条件为真，则随后的序列有效。通常在这种情况下该判定符号出现若干次，以指出由不同条件值引起的不同序列。

这个图能够完整地表示交换信号的所有序列，或者只是其中的一个有意义的子集。

这个图可有效地表示划分一个进程所引起的子进程间的相互作用。在这种情况下子进程的环境就是被划分的进程的环境。

D．7　用SDL/PR表示系统的指南

用户指南的D．7章阐述了命名为PR的SDL正文短语表示法，它与编程语言有非常相似之处。

第一节（§D．7、1）包含了对SDL/PR的扼要说明；简述了把它用于机器输入的适应性；它与程序相似之处及其差别。

第二节（§D．7．2）阐述了在整个建议中如何使用“语法图”来定义SDL/PR。

第三节（§D．7．3）从用户的观点看是最重要的。它阐述了有关SDL的实际使用，特别是考虑了正文短语表示法的特点及表达能力，涉及的方面有各状态的次序、在一状态内各跃迁次序的确定、状态的重复出现、过程、宏、标号及简化符号。

D.7.1 为什么要使用PR？

SDL的正文短语表示法（SDL/PR）是在1977～1980年研究期间创造的。因为把它作为建议之前觉得需要作某些改进，所以在1980年决定把它的定义作为建议的一个附件。这一改进已在下一研究阶段完成，而现在，SDL/PR是SDL建议的具体语法之一。

起先，创造SDL/PR的目的是把它用作为一种易于把SDL文件输入计算机的方法，因为GR形式较难输入（需要图形外部设备来处理）。由于这个原因，重点是放在PR和GR形式之间一对一的映射上。此后随着图形终端的进展（性能的提高和成本的降低），GR形式也适合于输入计算机已经是可以接受的了。而这並不降低PR形式的重要性和使用，因为有些用户发现它更合他们的心意，尤其是那些惯用编程语言工作的用户。

D.7.1.1 SDL/PR很象程序吗？

这种进展导致了GR和PR之间不严格的对应关系，所以我们还可以（容易地）把一种语言映射到另一种上，不过每种语言具有它自己的特色。初看上去，PR形式非常像编程语言（见图D－136）。

```
－
－
－
－
STATE aw_off_hook;
INPUT off_hook;
TASK 'activate charging';
TASK 'connect';
OUTPUT 'reset timer';
NEXTSTATE conversation.
－
－
－
－
```

图D－136
SDL/PR举例

事实上，关键在于根据什么一段正文可看成是编程语言。

如果定义一程序为一组计算机能理解的信息，则不仅PR而且GR都是“程序”。但若把程序的定义局限于一组计算机能理解並能执行的信息和指令，我们就可发现第一个差别是：PR表示法並不要求一定能为计算机所执行（虽然不反对这样做），关键在于它能够把准确的信息从一个人传递给另一个人。

作为程序来看，可能认为是“错的PR”（因为非形式正文的不完全性），若将它看作是系统功能要求的一种表示法，就是完全有效的PR。

和程序的通常表示法的又一个差别是在PR表示法的“风格”方面。

因为PR的目的是用来支持人与人之间的通信，所以务必允许有不同的PR格局，以便能用某种格局来指导读者集中注意力于某些被认为是更重要的方面。当然，这对程序来说是不重要的，因为已假定程序由计算机解释，计算机並不集中注意力于任一特定的方面，而是必须同等地考虑所有方面，它也不试图去“理解”程序。

因为PR与程序的类似性（这更多地是一种心理上的类似，因为GR形式在“语义上”也象PR一样与程序类似），使得一些可能使用CHILL来实现由PR表达的要求的程序员更喜欢PR。因此就有强大的引诱力来寻求从PR到CHILL的一对一映射，以便用PR表述的要求能自动地转换成CHILL代码。反过来也是令人感兴趣的，因为这样能从一个CHILL程序推导出一个PR描述。

在§D．8中，举例说明了把SDL映射到CHILL的几种可能的方法。

D.7.2 描述SDL/PR语法的元语言：语法图

语法图由用流线互相连接的终结符和非终结符构成。

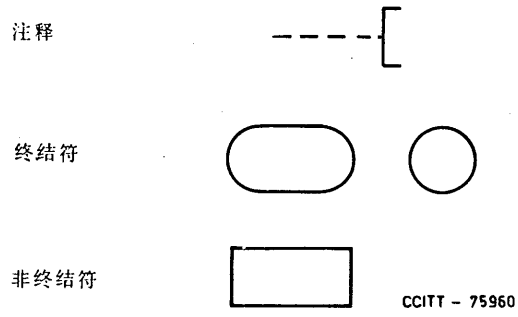
非终结符表示具有相同名字的另一语法图，它是一个简写符号，用来表示在多处使用的更加复杂的结构（也由终结符和非终结符组成的结构）。

有一专为插入注释而设置的符号。

每个符号，即每个图，必须只有一个入口处和一个出口处。

流线都是单向的，用一箭头指示流向。

终结符用两垂直边被半圆代替的矩形来表示。非终结符用一矩形表示。



图D－137

语法图中的图形符号

D.7.3 使用PR

在SDL/GR中，整个系统用许多文件来描述，这样做使它处理信息较为容易。功能块交互作用图表示系统结构及各功能块之间和各进程之间通过信道与信号进行的通信联系，而进程图在另外的文件中表示。

在PR中，系统结构和它的通信联系可以通过把系统的完整PR程序分解成不同的模块来表示，每一模块描述一个或多个进程的结构。这样，每个模块就能比较容易地进行处理。

这些模块之间可以有引用机制，见§D．5中的说明。

PR型式由一组语句组成，每个语句用“；”（分号）结束，它以语句

SYSTEM 名字;

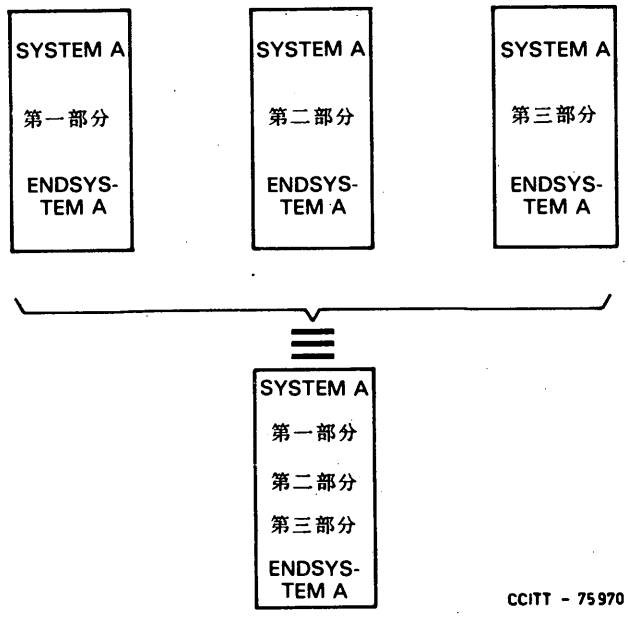
开始，而以语句

ENDSYSTEM 名字;

结束。

在这一对语句中，“名字”是任选的。

PR表示法不仅由嵌在SYSTEM－ENDSYSTEM语句之间的一组语句组成，而可以有若干组，每组以SYSTEM语句开始，以ENDSYSTEM语句结束。当且仅当各组具有相同的名字时(图D-138)，它们才属于同一SYSTEM表示。每个进程模块必须只放在一个组内，不可以使它分处于两个组。



图D—138

把一个系统表示分成三个部分

我们可以把包含在SYSTEM和ENDSYSTEM之间的各语句分别组合成一些功能块模块和信号定义、信道定义、数据类型定义、宏定义及过程定义等，如图D—139 所示。

SYSTEM .....

信号定义

信道定义

数据类型定义

宏定义

过程定义

[BLOCK...] 功能块模块
[ENDBLOCK]

[BLOCK...] 功能块模块
[ENDBLOCK]

ENDSYSTEM ....

图 D—139

在—SYSTEM模块中包含的各种语句的组合

每个功能块被嵌在BLOCK和ENDBLOCK 语句之间。和系统表示一样，一个功能块表示可以划分为两部分。依次类推，我们可把每个BLOCK模块看成是一组PROCESS（进程）模块加上信号定义、数据定义、过程定义及宏定义等。每个进程模块被嵌在PROCESS和ENDPROCESS语句之间。PROCESS模块不能再进一步划分。这在图D—140的例子中作了说明，在那里一个系统被分成两部分，一部分包含具有进程P 2 的功能块 b 1 ，具有进程P 4 的功能块 b 2 及具有进程P 5 的功能块 b 3 （图D—140a ），另一部分包含具有进程P 1 和P 3 的功能块 b 1 ，及具有进程P 6 的功能块 b 3 （图D—140b ）。这两个部分的总体表示如图D—140C）所示。

```
SYSTEM a;
BLOCK b1;
  PROCESS p2;
  -
  -
  -
  ENDPROCESS p2;
ENDBLOCK b1;
BLOCK b2;
  PROCESS p4;
  -
  -
  -
  ENDPROCESS p4;
ENDBLOCK b2;
BLOCK b3;
  PROCESS p5;
  -
  -
  -
  ENDPROCESS p5;
ENDBLOCK b3;
ENDSYSTEM a;
```

a) 第一部分

```
SYSTEM a;
  BLOCK b1;
    PROCESS p1;
    -
    -
    -
    ENDPROCESS p1;
    PROCESS p3;
    -
    -
    -
    ENDPROCESS p3;
  ENDBLOCK b1;
  BLOCK b3;
    PROCESS p6;
    -
    -
    -
    ENDPROCESS p6;
  ENDBLOCK b3;
ENDSYSTEM a;
```

b) 第二部分

```
SYSTEM a;
  BLOCK b1;
    PROCESS p1;
    -
    -
    -
    ENDPROCESS p1;
    PROCESS p2;
    -
    -
    -
    ENDPROCESS p2;
    PROCESS p3;
    -
    -
    -
    ENDPROCESS p3;
  ENDBLOCK b1;
  BLOCK b2;
    PROCESS p4;
    -
    -
    -
    ENDPROCESS p4;
  ENDBLOCK b2;
  BLOCK b3;
    PROCESS p5;
    -
    -
    ENDPROCESS p5;
    PROCESS p6;
    -
    -
    ENDPROCESS p6;
  ENDBLOCK b3;
ENDSYSTEM a;
```

c) 总体

图D—140

功能块的表示法

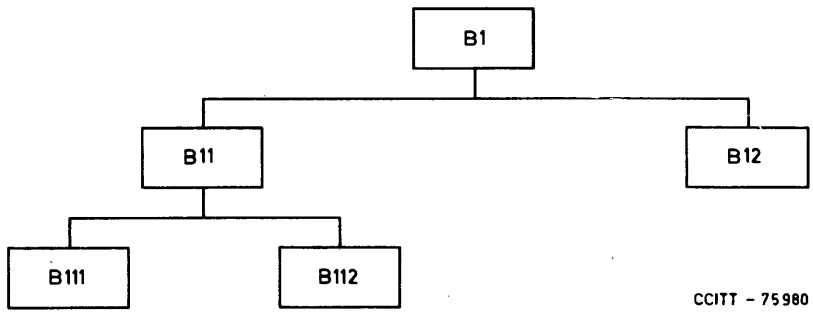
D．7.3.1　　用PR表达结构

GR中的实用文件有功能块树、进程树和信道子结构图，运用建议Z．102中所说明的功能块和信道子结构，可获得等效的PR表示。

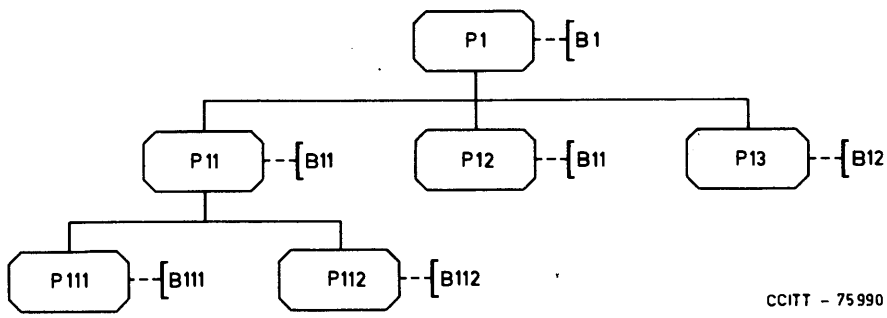
在PR中，功能块树、进程树和功能块交互作用图是混合在一起的，所得到的结构也包含进程语句，也就是它表示了整个系统。

与这些文件的直接对应是通过子功能块表、信道表及信号表给出的，可以把它们放进功能块子结构部分中，而该系统的结构则由功能块嵌套定义给出。

图D—141和图D—142显示GR功能块树和进程树，图D—143的PR例子是参照它们作出的。



图D—141
功能块树



图D—142
进程树

```
SYSTEM J;
- - - -
BLOCK B1;
- - - -
SUBSTRUCTURE B1;
SUBBLOCKS: B11, B12;
- - - -
BLOCK B11;
- - - -
SUBSTRUCTURE B11;
SUBBLOCKS: B111, B112;
- - - -
BLOCK B111;
- - - -
ENDBLOCK B111;
BLOCK B112;
- - - -
ENDBLOCK B112;
ENDSUBSTRUCTURE B11;
ENDBLOCK B11;
BLOCK B12;
- - - -
ENDBLOCK B12;
ENDSUBSTRUCTURE B1;
ENDBLOCK B1;
ENDSYSTEM J;
```

图D—143

功能块嵌套定义举例

在PR中，各进程及其结构都在它们的功能块内表示。有两种表示进程子结构的可能方式：在较高层次的功能块子结构处表示或是在每个较高层次的进程内部表示，图D—144和D—145指明了这两种可能性。

```
SYSTEM s;
- - - -
BLOCK b1;
- - - -
PROCESS p1;
- - - -
ENDPROCESS p1;
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12;
- - - -
SUBSTRUCTURE p1;
  p11 in b11,
  p12 in b12,
ENDSUBSTRUCTURE p1;
BLOCK b11;
- - - -
PROCESS p11;
- - - -
ENDPROCESS p11;
SUBSTRUCTURE b11;
SUBBLOCKS: b111, b112;
- - - -
SUBSTRUCTURE p11;
  p111 in b111,
  p112 in b112,
ENDSUBSTRUCTURE p11;
BLOCK b111;
- - - -
PROCESS p111;
- - - -
ENDPROCESS p111;
ENDBLOCK b111;
BLOCK b112;
- - - -
PROCESS p112;
- - - -
ENDPROCESS p112;
ENDBLOCK b112;
ENDSUBSTRUCTURE b11;
ENDBLOCK b11;
BLOCK b12;
- - - -
PROCESS p12;
- - - -
ENDPROCESS p12;
ENDBLOCK b12;
ENDSUBSTRUCTURE b1;
ENDBLOCK b1;
ENDSYSTEM s;
```

图D—144

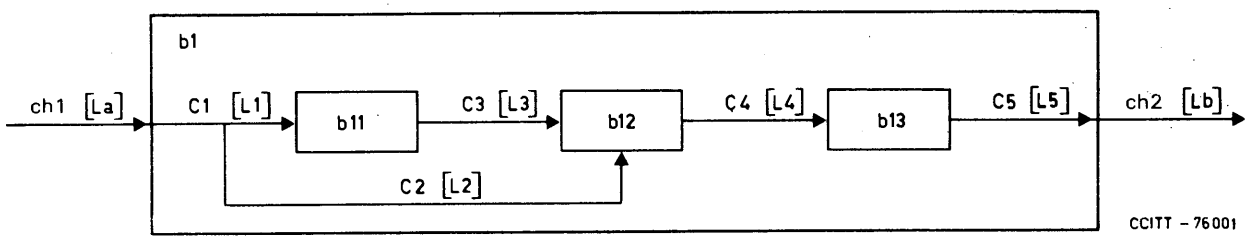
在功能块子结构层次上具有进程子结构的系统表示法

```
SYSTEM s;
- - - -
BLOCK b1;
- - - -
PROCESS p1;
- - - -
SUBSTRUCTURE p1;
  p11 in b11,
  p12 in b12,
ENDSUBSTRUCTURE p1;
ENDPROCESS p1;
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12;
- - - -
BLOCK b11;
- - - -
PROCESS p11;
- - - -
SUBSTRUCTURE p11;
  p111 in b111;
  p112 in b112;
ENDSUBSTRUCTURE p11;
ENDPROCESS p11;
SUBSTRUCTURE b11;
SUBBLOCKS: b111, b112;
- - - -
BLOCK b111;
- - - -
PROCESS p111;
- - - -
ENDPROCESS p111;
ENDBLOCK b111;
BLOCK b112;
- - - -
PROCESS p112;
- - - -
ENDPROCESS p112;
ENDBLOCK b112;
ENDSUBSTRUCTURE b11;
ENDBLOCK b11;
BLOCK b12;
- - - -
PROCESS p12;
- - - -
ENDPROCESS p12;
ENDBLOCK b12;
ENDSUBSTRUCTURE b1;
ENDBLOCK b1
ENDSYSTEM s;
```

图D—145

在进程定义内部具有进程子结构的系统表示法

连接（由划分得到的）各子功能块的各信道在功能块子结构定义中作出其定义。
由于在系统中包含有很多功能块（见图D—146），故在子功能块定义之前先定义信道会更好些。
这些信道的各种信号在信号定义部分作出定义。
图D—147 是参照图D—146的G R表示法作出的。



$$La = \begin{bmatrix} s1, \\ s2 \end{bmatrix} \qquad Lb = \begin{bmatrix} s3, \\ s4 \end{bmatrix}$$

$$L1 = [s1] \qquad L2 = [s2]$$

$$L3 = \begin{bmatrix} s5, \\ s6 \end{bmatrix} \qquad L4 = \begin{bmatrix} s7, \\ s8 \end{bmatrix}$$

$$L5 = \begin{bmatrix} s3, \\ s4 \end{bmatrix}$$

图D—146
一个使用功能块交互作用图的二层次系统表示

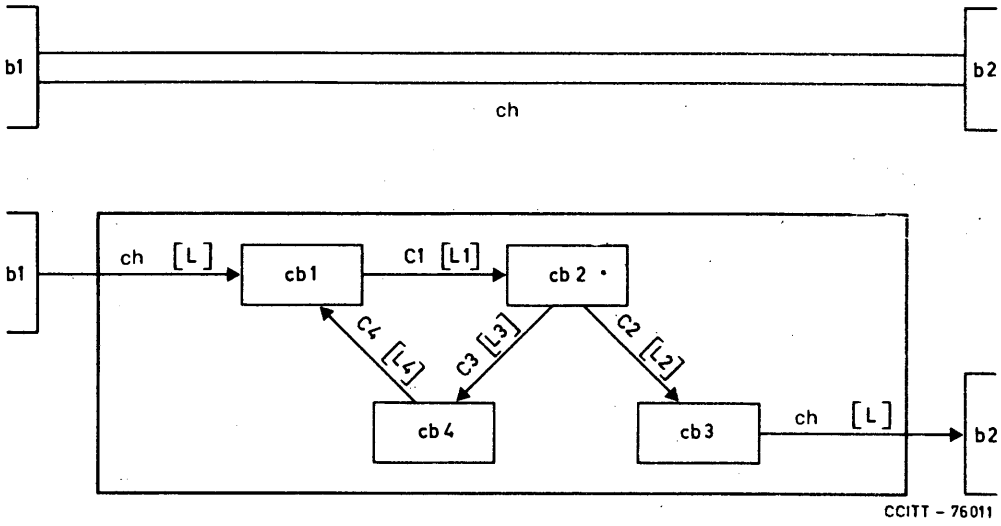
```
SYSTEM s;
CHANNEL ch1
  FROM ENV TO b1
  WITH s1, s2;
CHANNEL ch2
  FROM b1 TO ENV
  WITH s3, s4;
BLOCK b1;
- - - -
SUBSTRUCTURE b1;
SUBBLOCKS: b11, b12, b13;
CHANNELS: C1, C2, C3, C4, C5
  SPLIT ch1 INTO C1, C2;
  SPLIT ch2 INTO C5;
CHANNEL C1
  FROM ENV TO b11
  WITH s1;
CHANNEL C2
  FROM ENV TO b12
  WITH s2;
CHANNEL C3
  FROM b11 TO b12
  WITH s5, s6;
CHANNEL C4
  FROM b12 TO b13
  WITH s7; s8;
CHANNEL C5
  FROM b13 TO ENV
  WITH s3; s4;
    SIGNAL- - - -;
BLOCK b11;
- - - -
ENDBLOCK b11;
BLOCK b12;
- - - -
ENDBLOCK b12;
BLOCK b13;
- - - -
ENDBLOCK b13;
ENDSUBSTRUCTURE b1;
ENDSYSTEM s;
```

图D—147

显示信道的使用的二层次系统表示

信道定义或是在系统层次上（构成系统的各功能块之间信道的定义），或是在功能块子结构定义内（子功能块之间信道的定义）。在GR中，描述分解信道的图是信道子结构图；在PR中，这相当于信道定义内的一个描述，也相当于在较低层次上定义它的功能块定义和信道定义内的描述。

图D—148在两个不同层次上给出了两个功能块间一个信道的例子（第二个层次较低，且更为详细）。



$$L = \begin{bmatrix} s1, \\ s2, \\ s3, \\ s4 \end{bmatrix}$$

$$L1 = [s5]$$
$$L2 = [s6]$$
$$L3 = [s7]$$
$$L4 = [s8]$$

图D—148

用功能块和信道来表示信道

在这个例子的下面的图中，功能块b 1和b 2没有再进一步被划分，但在较高层次上作为一个系统所见到的信道Ch，在较低层次的表示中显示出了它的内部结构。表示信道的分解最好是在系统的各功能块不能再进一步分解时，否则各信道的各功能块之间以及相应的各功能块的各子功能块之间的相互作用必须指明，这就增加了表示该结构的难度。

图D—149指出了图D—148中信道分解的PR表示。

```
SYSTEM s;
- - - -
CHANNEL ch
  FROM b1 TO b2
  WITH s1, s2, s3, s4;
SUBSTRUCTURE ch;
BLOCKS: cb1, cb2, cb3, cb4;
CHANNELS: C1, C2, C3, C4;
- - - -
INCOMING ch TO cb1;
OUTGOING ch FROM cb3;
CHANNEL C1
  FROM cb1 TO cb2
  WITH s5;
CHANNEL C2
  FROM cb2 TO cb3
  WITH s6;
CHANNEL C3
  FROM cb3 to cb4
  WITH s7;
CHANNEL C4
  FROM cb4 TO cb1
  WITH s8;
SIGNAL
- - - -;
BLOCK cb1;
- - - -
ENDBLOCK cb1;
BLOCK cb2;
- - - -
ENDBLOCK cb2;
BLOCK cb3;
- - - -
ENDBLOCK cb3;
BLOCK cb4;
- - - -
ENDBLOCK cb4;
ENDSUBSTRUCTURE cn;
BLOCK b1;
- - - -
ENDBLOCK b1;
BLOCK b2;
- - - -
ENDBLOCK b2;
ENDSYSTEM s;
```

图D—149

划分一个信道的例子

在这里提供的所有例子中，其各功能块、进程和信道如果太长或太复杂的话，可以在另外某处用宏符号来描述。在认为需要此类描述的地方，仅需用一个宏调用就行了。而宏定义就包含了所要的信息。

D．7.3.2　信号定义

信号可以在系统层次上定义，或在功能块层次上定义，或在一个进程定义的内部定义。在系统层次上定义的信号表示与环境相互交换的信号和系统各功能块之间相互交换的信号；在功能块层次上定义的信号表示同一功能块内的各进程之间相互交换的信号。

在图D—150中，信号SIG1、SIG2在连接功能块b1和B2的信道内列出，或在连接其中一个功能块和环境的信道内列出，在功能块b1中声明的信号s1、s2、s3由属于该功能块的各进程所使用；在功能块B2中声明的信号s1、s2、s4、s5与在b1中声明的那些信号不同，它们可以为功能块B2的各进程所使用。

在一结构化系统中，可以有在功能块层次上定义的信号，以表示低层次功能块之间的信号交换（§D．7 3.1）。在信道定义中，可以有低层次信道的信号定义，这些低层次信道是通过划分所考虑的信道而产生的（§D．7.3.1）。各子信道的信号集合必须互不相交。

```
SYSTEM a;
    .
  SIGNAL SIG1(ty1, ty2), SIG2(ty2, ty3);
    .
  BLOCK b1;
      .
      .
      .
    SIGNAL s1(t1,t2,t3),s2,s3(t4,t2);

  ENDBLOCK b1;
  BLOCK B2;


    SIGNAL s1(t1,t2,t3),s2,s4,s5(t4,t2);


  ENDBLOCK B2;
ENDSYSTEM a;
```

图D—150

在各个层次上的信号定义的例子

D．7.3.3　信道定义

信道可以在系统层次上定义；或者，如果系统是结构化的，也可以在功能块层次上定义。最后，信道可以在一信道定义的内部定义。在系统层次上定义的信道表示系统功能块之间及这些功能块和环境之间的各信道。

在图D—151中，信道C1和C2被功能块b1用来与环境进行通信。

如果系统是结构化的，则每个功能块可以包含信道的定义，这些信道使得通过功能块的划分产生的若干功能块之间、及这些功能块和环境之间能够进行通信（§D．7.3.1）。

在信道定义中，可以有因某低层次信道分解得出的各低层次信道的定义（§D．7.3.1）。信道定义含有该信道的信号表，信号表中的各信号必须在信号定义节中定义。

```
SYSTEM a;

    CHANNEL    c1 FROM b1 TO ENV
               WITH s1,s2,s3
               REFINEMENT CHANNELS: c1.1,c1.2;
    CHANNEL    c2 FROM ENV TO b1
               WITH s4,s5
               REFINEMENT CHANNELS: c2.1,c2.2;


ENDSYSTEM a;
```

图D—151

信道定义举例

D．7.3.4 数据定义

根据建议Z．101，SDL具有每个系统都可以利用的预定义数据类型，它们是整型、实型、字符型、字符串、布尔型、进程实例标识符、时刻、和持续时间。不需要声明就可以在变量定义中使用它们的预先规定的名字（也就是INTEGER、REAL、CHARACTER、STRING、BOOLEAN、PID、TIME、DURATION）。

变量定义包含在一进程定义或一过程定义中，任何必须被出口、被进口、被透露或被视见的变量，一定要声明其性质为EXPORTED、IMPORTED、REVEALED或VIEWED。

每个数据都被一个进程实例所拥有。数据在进程定义中被定义（这样一来，一个进程的所有进程实例都有在进程层次上给出的数据定义的副本）。

要是在声明（DCI）节内要声明透露，只须在变量类型名字之前加上关键字REVEALED。

VIEWED声明写在DCL节之外，语法上先写关键字VIEWED，后面跟有变量名字、该变量的类型，最后是透露它的进程的标识符。

图D—152中，在进程P1和P2中分别有一个VIEWED和REVEALED变量“digit”声明的例子。它们都属于功能块b1。在本例中，进程p1是变量“digit”的拥有者，而进程p2能视见它。

```
SYSTEM a;
      .
      .
  BLOCK b1;
      .
      .
  PROCESS p1;
      .
      .
  DCL
    REVEALED digit INT,
    counter, alarm number INT,
    a,b,c, BOOL;



  ENDPROCESS p1;
  PROCESS p2;
  (   )
  DCL d,e,f, BOOL;
    VIEWED digit p1,



    ENDPROCESS p2;
  ENDBLOCK b1;
ENDSYSTEM a;
```

图D—152

同一功能块的各进程之间数据的可见性举例

在图D—153中，进程P1的所有实例都视见到变量“digit”。因为已将它声明为REVEALED和VIEWED。

```
SYSTEM a;
      .
      .
      .
BLOCK b1;
      .
      .
PROCESS p1;
      .
      .
      .
DCL
      REVEALED digit INT;
      .
      .
VIEWED digit INT p1;
      .
      .
      .
END PROCESS p1;
      .
      .
      .
      .
END BLOCK b1;
      .
      .
END SYSTEM a;
```

图D—153

同一进程定义的各进程实例之间的可见性举例

在必须指明需要视见操作时，可使用关键字VIEW，后面跟上要视见的变量名字和拥有该变量的进程实名字。变量名字用一逗号与进程实例名字隔开，两者都用圆括号括起。若略去关键字VIEW，会使进程实例其局部数据中去查找具有该名字的变量（图D—154）。

```
SYSTEM s;
.......
  BLOCK b1;
    .......
    PROCESS p1;
    .......
    DCL
      REVEAL digit INT,
      .......
    ENDPROCESS p1;
    PROCESS p2;
    .......
    DCL
    t INT,
    VIEWED digit INT p1,
    .......
    TASK t:=5*VIEW(digit,p1);
      .......
    ENDPROCESS p2,
  ENDBLOCK b1;
ENDSYSTEM s;
```

图D—154

使用一视见变量的例子

可被出口的变量在它的定义中（在出口进程中定义），必须有属性EXPORTED。

进口进程在进口节内要用关键字IMPORTED来声明打算进口的变量,后面跟要进口的各变量的名字、它们的类型以及出口进程的标识符。

一个变量可以同时被声明为IMPORTED和EXPORTED，这就允许一个进程实例把一个变量出口到同一进程的另一实例，並从其它实例进口该变量。在这种情况下，一定要注意避免名字冲突：赋值语句。

V：=IMPORT (V, PID)

将导致用PID实例的“v”代替自己的“v”。

由图D—155的例子中可以看出，每次进口操作要用关键字IMPORT,后面跟变量名字和进程实例标识符,两者用逗号隔开並且用圆括号括起来。

如果一个变量既被出口又被透露，则在声明中EXPORTED属性上应再加属性REVEALED（见图D—155）。请记住仅对同一功能块的各进程实例透露才是有效的。

在图D—155的例子中，功能块b 2中的进程P 2所拥有的变量“Couner”是既被出口又被透露的，它向进程P 1出口，並透露给同一功能块b 2中的进程P 3。

```
SYSTEM a;
        .
        .
  BLOCK b1;
        .
        .
    PROCESS p1;
      DCL
    EXPORTED digit INT,
    IMPORTED counter INT BLOCK b2,p1;
        .
        .
    ENDPROCESS p1;
  ENDBLOCK b1;
  BLOCK b2;
        .
        .
    PROCESS p2;
      DCL
        EXPORTED,REVEALED counter INT,
        .
        .
        .
    ENDPROCESS p2;
    PROCESS p3;
        .
        .
      DCL
        VIEWED counter PROCESS p2,
        .
        .
        .
    ENDPROCESS p3;
  ENDBLOCK b2;
ENDSYSTEM a;
```

图D--155

属于不同功能块的各进程之间数据的可见性举例

D．7.3.5　宏定义

宏结构是处理重复的一种手段。在PR程序中可以有宏，以分离出一串语句，而用一宏调用来取代它们(§D．7.3.7.8和图D—156 b））。

宏定义必须在系统、功能块、进程、或者过程定义的开头给出（图D—156 a）），这取决于是否能从所有进程调用它，还是从某一功能块中的各进程调用它，还是只从某一进程调用它或只从某一过程调用它。注意在PR中，宏总有一个入口处和一个出口处，因此在用PR来表示分别具有两个以上入口处或出口处的GR图时，必须要使用标号和连接（JOIN）。

图D—156中的例子，表示具有两个出口处a和b的宏（图D—156a）），这意味着在主程序中（图D—156 b））有两个相应的标号a和b。

```
MACRO EXPANSION a;
  INPUT register_alarm;

  JOIN a;
  INPUT tone;


  JOIN b;
ENDMACRO a;
```

a）宏定义

```
.....
.....
MACRO a;
a: .....
   .....
b: .....
   .....
```

b）宏调用

图D—156

使用宏的例子

D．7.3.6　过程定义

过程可以在分层的SDL结构的各个层次上定义，也就是在系统、功能块、进程或过程的层次上定义。

根据一个过程领定义的位置，此过程如果在系统层次上定义，则对系统中所有的进程和过程是可见的；若在功能块层次上定义，则对此功能块的所有进程和过程是可见的；或者此过程只是对定义它的进程（过程）是可见的。

形参定义以关键字FPAR开始。如果相应的实参是一信号，那么作为形参所列出的变量可以有属性IN、IN/OUT或SIGNAL。

具有关键字IN的参数是按值传递的，具有关键字IN/OUT的参数是按地址传递的（见§D．7.3.7.9）。一个过程仅有一个出口处，但可以有一个或多个RETURN语句。在解释关键字RETURN之后，下一个语句将是主程序中接在该过程调用后面的语句（图D—157）。

过程的可见性模式很接近于数据类型定义的模式，仅有的不同是没有预定义过程。因此任何过程都应这样来定义，以使调用者能够看见它。图D—157给出了一个过程定义的例子。在例子的第一部分形参是正确的。在第二个过程定义语句中有一个错误，因为各形参的类型不是按过程调用中各实参对应的正确次序来排列的。

```
....
....
SIGNAL sig1(int,bool,int),
....
DCL number INTEGER, connected BOOLEAN;
....
PROCEDURE proc1
FPAR
      SIGNAL sig,
      in num INTEGFR,
      IN/OUT conn BOOLEAN;
....
....
....
ENDPROCEDURE;
```

按照过程调用这个过程定义语句是正确的，但

```
PROCEDURE proc1
FPAR
    IN num INTEGER,
    SIGNAL sig,
    IN/OUT conn BOOLEAN;
```

是错的!!!

```
....
....
CALL proc1(sig1,number,conn);
....
....
```

图D—157

使用过程的例子

D．7.3.7 进程定义

如前所述，进程定义是嵌在PROCESS和ENDPROCESS语句之间的。进程的名字必须接在关键字PROCESS后面。该名字（隐式地限定了进程）和所在的功能块名字及系统名字一起形成进程的标识符。

如果有任何形参 它们必须在进程语句之后声明，如图D—158所示，关键字FPAR位于各参数之前。

实参写在创建（CREATE）语句中（§见D．7.3.7.1）。

```
PROCESS process name:
FPAR
variable name type;
```

图D—158

形参定义

实例的数目必须包括在括号中，它是任选的。

实例数目是一对用逗号隔开的整数，第一个整数表示在系统初始化时存在多少进程，而第二个整数表示在系统生命期间可能存在的最大进程实例数。如果第一个数被省略，那么在系统开始时只有一个实例，如果两个数都被省略，那么贯穿整个系统生命期只有一个进程实例（图D—159）。

```
a) PROCESS p1(5,18);          初始化时有5个实例.
                              在系统生命期内最多
                              时有18个实例
b) PROCESS p2(,5);                等效于 PROCESS p2(1,5);
c) PROCESS p3;                    等效于 PROCESS p3(1,1);
d) Process mistake (5,3); 错!!!!
```

图D—159

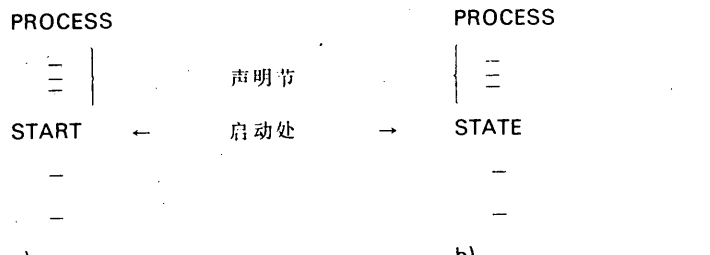
进程举例

很明显，最大的实例数（第二个数）应等于或大于系统初始化时所具有的实例数（图D—159d）。

在PROCESS语句之后，必须定义进程所拥有的数据，然后是局部于该进程的过程和宏（见§D．7.3.5和§D．7.3.6）。

在这些定义之后开始表示进程的行为，这或者通过后面跟一跃迁串的START 语句来指明，或者通过一STATE 语句来指明。

如果没有START语句，每个进程实例在第一个STATE语句处启动它的实体（图D—160 b））。



图D—160

进程的启动

D．7.3.7.1　进程的创建

一进程实例发出一创建动作，能够创建同一进程的实例或同一功能块的不同进程的其它实例。

CREATE语句可以包含实参表，括在圆括号内，如图D—161所示。

```
SYSTEM s;
  BLOCK b;
    PROCESS p;
    .......
    .......
    DCL             a,c   INTEGER,
                    b     BOOLEAN.
          .......
          .......
            CREATE p1(a,b,c);
          .......
          ENDPROCESS p;
          .......
          PROCESS p1;
          FPAR
          .......
            digit INTEGER, connected BOOLEAN, number INTEGER;
          START
          .......
          ENDPROCESS p1;
        ENDBLOCK b;
      ENDSYSTEM s;
```

图D—161

创建一个进程实例的例子

D．7.3.7.2 状态及重复出现

在PR中，一状态是用关键字STATE，后跟状态名字来表示的。

一状态在下一状态语句（下一STATE语句）或在进程末尾（ENDPROCESS或STOP语句）处结束。

此外在PR中常常有明确的语句来指明进入一个状态（NEXTSTATE）或离开一个状态（STATE）。

没有一个通用的语句指明上述两种情况，象在GR中那样。

状态的重复出现和在§D.6（GR）中所给出的语义模型有相同的说明和关系。

在§D．6.3.6.5 中给出过一些GR例子，下面是这些例子的PR形式。

```
b:    NEXTSTATE state_1;
   STATE State_1;
     INPUT A;
     NEXTSTATE state_2;
     INPUT C;
     JOIN a;
   STATE State_2;
     INPUT C;
     JOIN b;
     INPUT B;
a: NEXTSTATE State_3;

   STATE State_3;
     INPUT D;
     JOIN b;
```

a) 完整的图

```
     -
     -
   STATE State_1;
     INPUT A;
     NEXTSTATE STATE_2;
     INPUT C;
     NEXTSTATE State_3;
   STATE State_2;
     INPUT B;
     NEXTSTATE State_2;
     INPUT C;
     NEXTSTATE State_1;
   STATE State_3;
     INPUT D;
     NEXTSTATE State_1;
```

b) 具有多个主状态的图a)，其下一状态用作为主状态的连接符

图D—162

某个状态的重复出现举例（与图D—93等效）

D.7.3.7.3　PR语句和数据的使用

D.7.3.7.3.1　输入语句

INPUT语句包含一信号表，信号中所包含的数据项都是用变量标识符命名的，变量标识符必须具有在信号定义中指明的类型，因此它们的位置是很重要的。这些变量标识符括在圆括号内，并用逗号隔开（见图D—164）。如果丢弃一个或多个信号数据项，那么相应的变量就应弃去，这时用连续两个逗号来表示（图D—163）。

```
INPUT a(var1,var2,,var4);
```

注-在这个语句中，丢弃信号a的第三个数据项

图　D—163

输入信号a仅定义了四个数据项中的三个

```
SIGNAL sig1 (INTEGER,BOOLEAN,INTEGER);
DCL a INTEGER,b BOOLEAN,c INTEGER,
```

a）声明

```
INPUT sig1(a,b,c);
```

b）一个正确的输入

```
INPUT sig1(a,c,b);
```

c）一个不正确的输入

图D—164
INPUT　语句

D.7.3.7.3.2　保存语句

SAVE语句可以有信号名字表，或者有一个星号。如果在INPUT语句中所有未指名的进入信号都要保存的话就用星号。使用SAVE的一个简单例子在图D—165 中给出。

```
—
—
—
SAVE State_31;
  SAVE S;                          /*S '到达，进入队列并保留在队列中，R到达
                                   并立即消耗掉，触发到状态32的跃迁。
  INPUT R;
  NEXTSTATE State_32;
STATE State_32                     一到达状态32，'S'立即被消耗，
                                   并触发到下一状态的跃迁。*/
  INPUT S;
—
—
—
```

图 D—165

使用保存的例子

D.7.3.7.3.3 输出语句

输出（OUTPUT）语句包含发送到其它进程去的信号表。对在信号中包含的每个数据项，可以有一相应的实在表达式或值，如果一个或多个数据项的值没有规定，那就被略去，而空白位置用两个连续的逗号来表示（图D—166）。

```
OUTPUT sig1(a,,c);
```

图 D—166

输出信号Sig1的三个数据项中只定义了两个

实在值表置于圆括号内，在该信号中所涉及的每个数据项的值必须具有所规定的类型。参照图D—164a）输入语句中的声明，图D—167示出了一个正确的输出和一个不正确的输出。

```
OUTPUT sig1(2,true,10)
```

a）一个正确的输出

```
OUTPUT sig1(false,true,10)
```

b）一个不正确的输出

图 D—167

输出语句

D.7.3.7.3.4 任务语句

任务语句可以包含一个或多个赋值语句、一组正文语句和(或)非形式正文。非形式正文由一个名字和（或）一用单引号括起来的短语组成。各语句成正文彼此用逗号隔开（图D—168）。

```
TASK a:=b;
TASK 'connect the subscriber';
TASK RESET(TIME);
TASK main_assignment, c:=d+e
TASK var1:=var2*var3,
     var4:=var5 MOD var6;
```

图 D—168

任 务 语 句

D.7.3.7.3.5 判定语句

SDL/PR的判定用关键字DECISION来表示，后面跟判定名字和(或）表达式或正文串。后者是包含在引号''之间的正文（形式的或非形式的）。

在引出路径的集合的前面要放一条，DECISION语句，在集合结束处要放一条ENDDECISION语句（见图D—169 ）。

```
DECISION ....;
(result): ....;
(result): ....;
(result): ....;
  ENDDECISION;
```

图 D—169

判定的界限

判定的结果在圆括号内指明，它们可用由引号定界的一个或多个正文串来表示，或者用包含在判定语句内的表达式的求值所获得的值来表示。括号内的各种不同结果用逗号隔开，各个值用常量表达式表示，或者用常量表达式的上下界范围来表示。结果值必须具有在判定语句中包含的表达式的类型，如图D—170 、D—171和D—172 各例所示。

可以显式地指明一些结果，而用关键字ELSE来组合所有其它可能的结果，举例见图D—170 。

```
DECISION 'x';
(2):          .  | 路径 1
              .  |
(6):          .  | 路径 2
              .  |
(9,10):       .  | 路径 3
              .  |
ELSE:         .  | 路径 4
              .  |
ENDDECISION;
```

注－x是 1 和10之间的一个整数。

x= 2 时选择路径 1，

x= 6 时选择路径 2，

x= 9 或x= 10时选择路径 3，

x= 1 、 = 3 、 = 4 、 = 5 、 = 7 或= 8 时选择路径 4 。

图 D—170

具有ELSE 的判定

```
                    DECISION 'Subscriber category';
('international', 'national'):   .  | 路径 1
                                  .  |
('local')                         .  | 路径 2
                                  .  |
```

图 D—171

DECISON举例

例子

```
DCL x INT,
  DECISION x;
(2+5)          :.........;
(6+8,10+12)  :.........;
ELSE           :.........;
  ENDDECISION;
  DECISION a;
(true) :.........;
(false):.........;
  ENDDECISION;
  DECISION x;
    (10) :.........;
ELSE    :.........;
  ENDDECISION;
```

图 D—172

判定结果举例

ENDDECISION语句在结构化程序设计中提供构造能力，如图D—173所示。

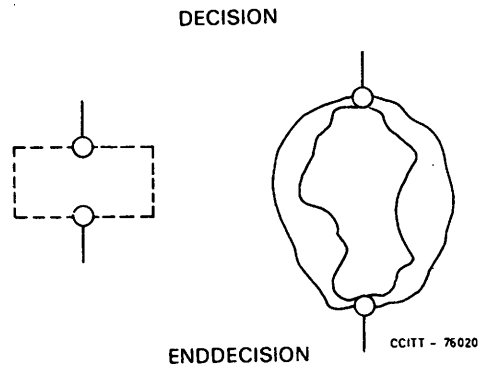


图 D—173

DECISION（判定）的结构能力

全部路径在ENDDECISION语句处结束。那些原先未结束的路径在ENDDECISION语句后继续，在图D—174 和图D—175 中显示了等效的例子。

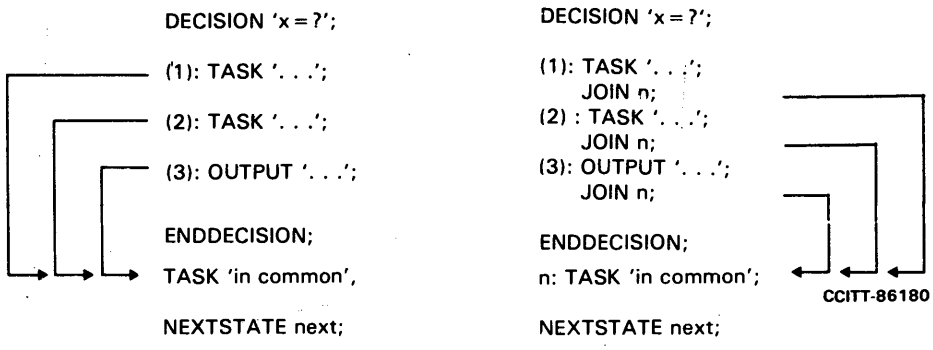


图 D—174

根据判定（DECISION）的转移

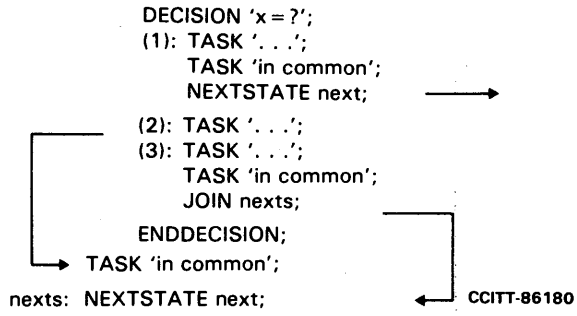


图 D—175

图 D—174 所示例子的等效转移

判定语句可以用来模拟IF—THEN结构、DO—WHILE及LOOP—UNTIL结构。

图D—176 表示IF—THEN结构，图D—177 表示IF—THENELSE结构，图D—178 表示DO-WHILE结构，而图D—179 表示LOOP—UNTIL结构。

```
BLOCK a;
  PROCESS b;
    STATE s1;
      INPUT i1;
      DECISION 'i1 = ?'
(0): TASK t1;
ELSE:;
      ENDDECISION;
  ENDPROCESS b;
ENDBLOCK a;
```

图 D—176

IF—THEN结构

```
BLOCK a;
  PROCESS b;
    STATE s1;
      INPUT i1;
      DECISION 'i1=?'
(0): TASK t1;
ELSE: TASK t2;
      ENDDECISION;
  ENDPROCESS b;
ENDBLOCK a;
```

图 D—177

IF THEN ELSE 结构

```
BLOCK b1;
  PROCESS p1;
    STATE s1;
      INPUT i1;
1:    DECISION 'b=?';
      (false): JOIN 11;
      ELSE : ;
      ENDDECISION;
      TASK t1;
      JOIN 1;
11:-
      -
      -
  ENDPROCESS p1;
ENDBLOCK b1;
```

图 D—178

DO WHILE 结构

```
BLOCK b1;
  PROCESS p1;
    STATE s1;
      INPUT i1;
1   DECISION 'b=?';
      (false): JOIN 1;
      ELSE:;
      TASK t2;
      ENDDECISION;
  ENDPROCESS p1;
ENDBLOCK b1;
```

图 D—179

LOOP…UNTIL…结构

D.7.3.7.3.6　替换语句

在确定一个系统的规格时，会有若干行为中任一行为都能满足我们要求的情况，于是我们要指明若干容许的行为，以便让实现者去选择其中的一种。

各种可供选择的方案用关键字ALTERNATIVE和一相关联的表达式或非形式正文来指明。一组可选择的行为以语句ENDALTERNATIVE作结尾；每个可选择的行为用一名字来标识。

其整体结构和DECISION的结构相同，其差别是在语义上。对DECISION来说，是按照进程生存期中变化着的某一数据项的值来选择某二个分支的，而对ALTERNATIVE，在进程实现中只存在一个单一的分支，因此它是在实现之前就选择好的，图D—180给出了一个例子。

```
ALTERNATIVE 'Alarm response'
  ('first choice'): OUTPUT ring_bell;
  ('second choice'): OUTPUT light_alarm;
ENDALTERNATIVE;
```

在实现时我们可以有

或

```
OUTPUT ring_bell;
```

或

```
OUTPUT light_alarm;
```

图　D—180

一个ALTERNATIVE语句的例子

ALTERNATIVE语句的结果具有和判定相类似的语法，图D—181表示了这样的一个例子。

```
DCL a INTEGER 1:10;
ALTERNATIVE a
(1,4) :.....;
(5,10):.....;
ENDALTERNATIVE;
```

图　D—181

替换语句举例

D.7.3.7.4　状态图形

在PR中没有对应的表示。

D.7.3.7.5　时间

时间概念用于SET和RESET语句。它们在PRTASK语句中可以是形式正文（见§D.7.3.7.3.4）。

D.7.3.7.6　允许条件

在PR中，允许条件用关键字PROVIDED后面跟待求值的条件来表示。这个关键字附加到INPUT语句中，附加到INPUT的条件是一布尔表达式。如果条件为真，就允许产生跃迁；如果条件为假，信号就被保存。因为在每个状态上，一个输入只能出现一次，所以不允许从同一状态使一输入出现两次，每次具有一个不同的允许条件。

允许条件所附表达式中使用的变量可以是局部变量或进口的变量，它们不能够被视见到。

使用允许条件的一个例子如图D—182所示。

```
SYSTEM s;
  BLOCK b1;
  .........
    PROCESS p;
    .........
    DCL
      connected BOOLEAN,
    IMPORTED alarm BLOCK b2,p2;
    .........
    STATE s1;
      INPUT i1 PROVIDED connected;
      .........
      INPUT i2 PROVIDED IMPORT alarm,p2;
      .........
    ENDPROCESS p;
  ENDBLOCK b1;
  BLOCK b2;
  ........
    PROCESS p2;
    ........
    DCL
      EXPORTED alarm BOOLEAN;
      ........
    ENDPROCESS p2;
  ........
  ENDBLOCK b2;
ENDSYSTEM s;
```

图　D—182

允许条件举例

D.7.3.7.7　连续信号

在PR中，连续信号用关键字PROVIDED后跟待求值的条件来表示。连续信号语法上与允许条件不同之处在于没有一个相关联的输入语句。所跟条件是利用该进程可见数据的一个布尔表达式，如在求值时条件为真，则激发跃迁。如果有若干个连续符号，则必须指明求值的次序。这是通过把一个优先级与各个条件联系起来而实现的，为此要使用关键字　PRIORITY　后跟一个整数值。规定连续信号的优先权比通常输入信号的优先权低。要求值的条件可以只包含局部的或进口的变量，不能包含任何可被视见的变量（图D—183　和D—184　）。

```
SYSTEM s;
  .....
  BLOCK b1;
    .....
    PROCESS p1;
      .....
      DCL
        x INT;
      .....
      STATE s1;
      PROVIDED x=2 PRIORITY 0
      .....
      PROVIDED x=5 PRIORITY 1
      .....
      INPUT i1;
      .....
    ENDPROCESS p1;
  ENDBLOCK b1;
ENDSYSTEM s;
```

图 D—183

局部变量的连续信 号举 例

```
SYSTEM s;
  .....
  BLOCK b1;
    .....
    PROCESS p1;
    .....
    DCL
      connected BOOL,
    IMPORTED alarm BLOCK b2,p2;
    .....
    STATE s1;
    PROVIDED connected PRIORITY 0;
    .....
    INPUT i1;
    .....
    PROVIDED IMPORT (alarm,p2) PRIORITY 1;
    .....
    ENDPROCESS p1;
  ENDBLOCK b1;
  BLOCK b2;
    .....
    PROCESS p2;
    .....
    DCL
      EXPORTED alarm BOOL;
    .....
    ENDPROCESS p2;
    .....
    ENDBLOCK b2;
    .....
ENDSYSTEM s;
```

图 D—184

进口变量的连续信 号举 例

D.7.3.7.8　宏调用

MACRO语句是：MACRO　名字；

与关键字MACRO相关联的名字指明一具有该名字的MACRO定义（参考§D.7.3.5）。MACRO语句可以插在一PR表示的任何位置，但必须先要作出相应的宏定义。

为了解释该PR表示，应当用MACRO定义来替换MACRO调用语句的每次出现，如图D—185的例子所示。

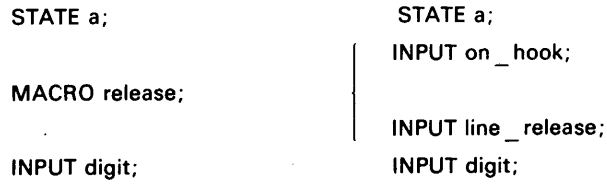


图　D—185

MACRO调用语句的解释

D.7.3.7.9　过程调用

过程调用包含该过程的实在参数表，它们由该过程可见到的输入和输出信号组成。请注意：调用者可见到的、但未在过程调用中声明为实在参数的全部信号，在过程生存期内均自动地予以保存。此外，各实在参数还包含对调用者是可见的数据，并且也是对过程可见的数据。

也要注意：在过程定义中对IN或IN/OUT进行了声明，所以就不许在调用语句中重复声明。另外，还要注意在过程定义中具有IN/OUT属性的参数应与调用者所直接或间接（当调用者是一过程时）拥有的变量相对应。在实在参数中提到的数据是与形式参数按次序相关联的。如果一个参数没有给出，就必须用两个连续的逗号来指明。在这种情况下对应的形式参数具有“不确定的”值。图D—186 示出了一个例子。

```
CALL proc1(sig1,,conn);
```

（见§ D.7.3.6的例子、即图D-157）

图　D—186

对一过程调用（CALL）的解释

D.7.3.7.10　标号（连接符）

标号用作与语句相关联的入口点，这使控制能借助一JOIN语句而转移（图D—187 ）。

图　D—187

标号

不允许把控制转移到（因而也不允许把标号联到）如图D—188所示的各种类型的语句。

SYSTEM,	ENDSYSTEM,
BLOCK,	ENDBLOCK,
PROCESS,	ENDPROCESS,
STATE,	ENDDECISION,
INPUT,	SAVE,
ENDMACRO,	ENDPROCEDURE,
ENDALTERNATIVE	

图 D—188

不允许用标号的地方

一个标号只能与一个语句相关联。标号总是局部于一个进程的，不允许借助标号把控制从一进程转移到另一进程。

D.7.3.7.11 语句的可达到性

除非指明了控制的转移，语句是一个接一个地顺序进行解释的。

转移控制的语句如图D—189至D—193所示。

S T A T E：在某一信号到来时，控制被转移到包含该信号名字的I N P U T或S A V E语句。

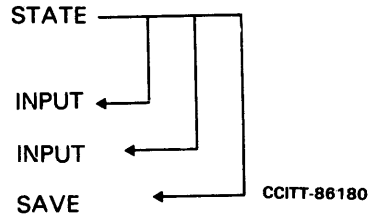


图 D—189

S T A T E中控制的转移

J O I N：控制被转移到一个语句，这个语句的标号具有在J O I N中指出的名字。应当有一个且仅有一个语句具有此标号。

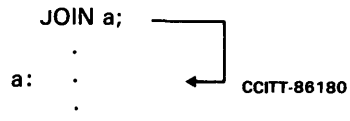


图 D—190

J O I N中控制的转移

NEXTSTATE：控制被转移到一个状态，该状态具有在NETSTATE语句中语句中指明的名字。

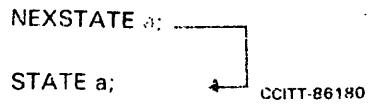


图 D—191
在NEXTSTATE中控制的转移

STOP：解释结束，没有控制转移。

图 D—192
没有控制转移

隐式转移：与原先在判定转移中说明的相同。

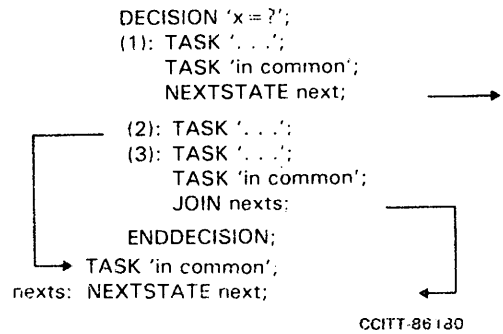


图 D—193
控制的隐式转移举例

D.7.3.7.12 发散与汇聚的使用

在PR中，发散与汇聚也用§D.7·3.7.10所定义的标号来实现。

D.7.3.7.13 注释

注释借助于关键字COMMENT来插入。

此关键字能象一个语句那样，插入到一TASK语句能插入的任何地方。此外，此关键字可以在任何其他的语句结尾处插入。

STATE a COMMENT '……';

END语句除外，这里不能连接注释。

凡是在空白可以出现的地方都可以加CHILL注释（/*……*/），使用这类注释的一些例子在图D—194和D—195中给出。

```
DCL
a INTEGER; /*变量a的使用说明*/
b BOOLEAN; /*变量b的使用说明*/
```

图 D—194

在变量声明中使用CHILL注释

```
PROCEDURE ABC (IN a INTEGER, /*这个形式参数的意义*/,
IN/OUT b BOOLEAN; /*这个形式参数的意义*/);
```

图 D—195

在过程定义中使用CHILL注释

D.7.3.7.14 简化符号

*（星号）可以插在 STATE/INPUT/SAVE语句中，用来表示该语句所能用的全部名字。举例：

```
a) STATE *;                    b) STATE A,B,C,D,G;
```

注-若A、B、C、D、G是在该进程中定义的一组状态，则a)与b)等效。

图 D—196

ALL（全部）状态符号

```
a) INPUT*;                     b) INPUT I1, I3, I6;
```

注-a)与b)是等效的，这里I1、I3和I6是进入该进程的进入信号，而未声明作为该状态的INPUT或SAVE。最多只能有一个带*的INPUT或SAVE可以附加到一特定状态。

图 D—197

ALL（全部）进入信号符号

a) SAVE *　　b) SAVE I1, I3, I6;

注-a)与b)是等效的，这里I1、I3和I6是该进程的进入信号没有声明作为该状态的INPUT，没有别的SAVE能附加到同一状态。

图　D—198

ALL（全部）保存符号

对于STATE*，我们能附一个除外表，来排除表中列出的各状态，被排除的各状态名字要用〔〕（方括号）括起。

a) STATE * [A,C,D];　　b) STATE B, E, F, G;

注-若A、B、C、D、E、F、G是进程的状态，则a)和b)等效.

图 D—199

除某些状态以外的全部符号

一个短划（－）能插在一条NEXTSTATE语句中，用以指明跃迁的始发状态与跃迁结束处的状态相同（图D—200）。

a)
```
STATE a;
INPUT i;

NEXTSTATE -;
```

b)
```
STATE a;
INPUT i;

NEXTSTATE a;
```

注-a)和b)等效

图　D—200

始发状态与结束状态相同

该符号在下述场合特别有用：即希望在若干状态中执行相同的一组动作，然后又回到原来的状态，如图D—201所示。

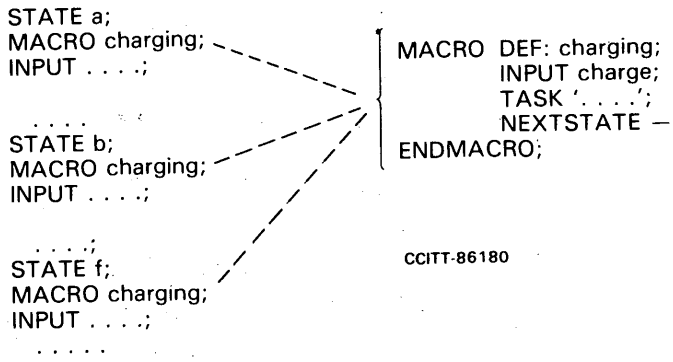


图 D—201
宏定义

D.8 映射

这一章描述SDL和CHILL之间的映射（§ D.8.1）及SDL/GR和SDL/PR之间的映射（§ D.8.2）的一些情况。

D.8.1 SDL和CHILL之间的映射

本节说明把SDL映射到CHILL的某些可能的方法。办法是通过一个例子，我们并不想举出全部方法，也不打算建议其中的哪一种应该在实际中采用。

事实上映射不能够仅仅考虑可利用的CHILL编译程序和目标机，因为一般说来映射是一个很复杂的智力活动，只有通过经验，设计者、程序员才能选定一特定的CHILL程序结构，用来实现一特定的SDL表示。这种情况也适用于用SDL来表示由CHILL程序实现的功能。一对一的映射（如果能实现的话）未必是用SDL来表示由CHILL实现的各功能的最好途径。

用此种方法，一个完全的SDL图和一段（不完全的）CHILL程序之间的映射总结构构示于图D-202。

这两种语言结构间映射的若干例子在图D—203至D—206中给出，它们涉及以下的SDL结构：

* 状态和信号的接收与保存，选择下一状态，
* 输出，
* 连接，
* 判定。

声明模块既包含了被变换的SDL图中所用的全部信号定义和声明，又包含了与这些信号有关的所有变量。所有这些变量对表示SDL图功能块的该模块都是移出的。

```
Declaring: MODULE
        /*CHILL模块含有在SDL图中使用的各信号和有关变量*/                    */
            GRANT
        /*信号和变量的移出*/                                              */
            SIGNALS
        /*信号定义*/                                                      */
            SYNMODE (OR NEWMODE)
        /*类型定义*/                                                      */
END Declaring;
Functional_Block: MODULE
        /*含有SDL图的过程部分的模块*/
            SEIZE
        /*移入可以被该功能块接收的和发送自（到）该功能块的所有信号和变量*/
        /*数据定义和声明，这种数据对所有属于该模块的进程是全局性的数据*/
Process name:PROCESS ( );
        /*局部数据定义和声明*/
            nextstate:=....;
            join:=none;
            DO FOR EVER;
              state_loop:CASE nextstate OF
        /*根据变量nextstate的循环，该变量指示SDL状态*/
                                (state_label1):RECEIVE CASE
                                (signal name1):


                                (signal namen);
            ESAC state_loop;
            DO WHILE join/=none;
                CASE join OF
                    (join_lab1): join: = none;


                    (join_labm): join: = none;


                ESAC;
            OD;
        OD;
        END process_name;
END Functional_Block;
```

图 D—202

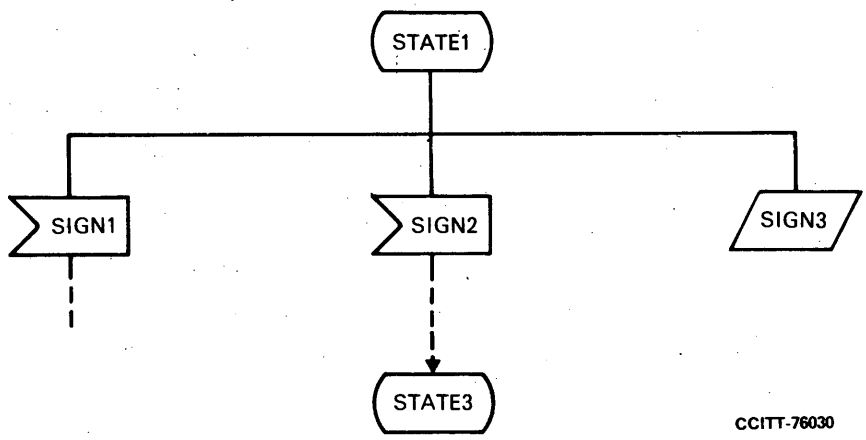
从SDL到CHILL一对一映射的例子

此功能块摸块表示ＳＤＬ各进程的行为（过程部分）。

在这个变换模式中，每个ＳＤＬ进程表示为一个无限循环，一个名为“nextstate ”的变量指明待检查的状态，而一个名为“join ”的变量指明各可能的连接点，以确定各组公用语句。

利用ＣＨＩＬＬ的case结构，可以选择nextstate的值。case的每个入口标志一个ＳＤＬ状态，在每个入口处要在可能输入的各信号之间进行选择，每个输入信号确定一组待完成的动作（“跃迁路径”）。

在每条迁迁路径的终点或者对变量“nextstate ”进行赋值，以直接确定下一状态，或者对变量“join ”进行赋值。根据变量“joi n ”的当前值，继发一个选择循环，就ＳＤＬ意义来说，每个跃迁结束，並在结束时把一个值赋给变量“nextstate ”。



a) SDL

```
STATE1:
   RECEIVE CASE
      (SIGNAL1): .............
                 .............
      (SIGNAL2): .............
                    NEXTSTATE: = STATE3;
      ELSE GETOUT (LIST1)
ESAC STATE1;
```

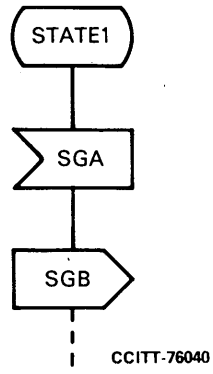
b) CHILL

图　D—203

映射ＳＴＡＴＥ/ＩＮＰＵＴ/ＳＡＶＥ/ＮＥＸＴＳＴＡＴＥ的例子

在建立ＳＤＬ和ＣＨＩＬＬ的联系中，主要问题之一是信号接收的语义彼此不同。事实上ＣＨＩＬＬ不消耗（因此不破坏）任何信号，除非把它们接收下来（持续信号）；而ＳＤＬ进程却消耗（因此破坏）全部接收到的信号，直到收到的信号与该状态输入信号表中的一个输入信号相符合。这种语义的不一致已经通过引入内部例行程序ＧＥＴＯＵＴ解决了，它在ＣＨＩＬＬ　ＲＥＣＥＩＶＥ　ＣＡＳＥ结构中作为一个选择（ＥＬＳＥ路径），　　如图D—203所示。ＣＨＩＬＬ内部例行程序ＧＥＴＯＵＴ知道（通过参数）输入和保存的信号表，当调用它时，就破坏进程可利用的其它信号。

在执行ＣＥＴＯＵＴ例行程序之后，就安排状态选择器以重复那个状态的循环，直至选择到一个有效的输入信号（或直至到达一个有效的输入信号，如果当前存在有效信号的话）为止。



a) SDL

```
STATE1:
   RECEIVE CASE
      (SGA): pi:=get_instance_value();
             send SGB to pi;
             nextstate:=.....;
      ELSE   nextstate:=state1;
ESAC STATE1;
```

b) CHILL

图 D—204

映射OUTPUT的例子

例如，图D—204中，输入信号SGA一经被识别，就选择信号SGB的合适的目的进程实例，并且发送信号SGB。

在发送信号SGB之前，可能需要装填某些要被信号传送的信息域。这或者可紧接在信号发送之前来进行或者可在以前事先装好。

当在图中（见图D—205)遇到一个连接点时，就给变量“JOIN”赋适当的值。如图D—202所示，接着执行根据变量“join ”的值所定的循环，以便确定下一个状态。从编程语言的观点看，一个连接点可以看作是一“goto ”结构。把所有的连接点集中起来便于检查，並使整个骨架程序不使用goto 就能编制，这样就使程序更易读了。

一个SDL判定可直接翻译成CHILL的case结构，如图D—206所示。

D.8.2 GR和PR之间的映射

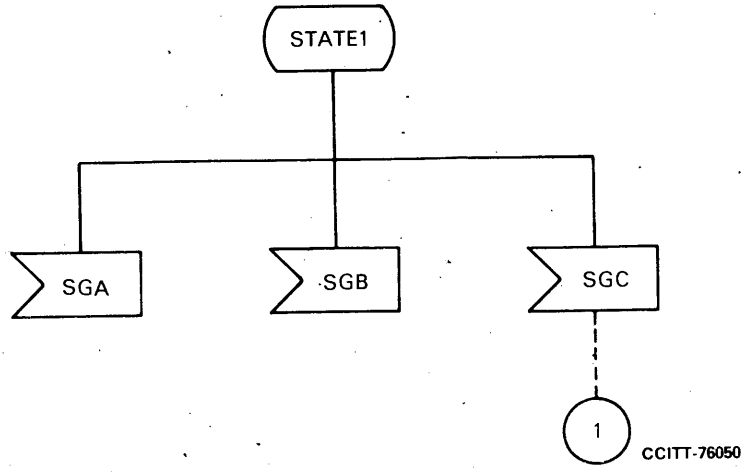
一个系统的某些部分只能用PR来描述，如数据定义和信号定义。因此，使用GR表示的系统可能需要用某些PR语法来补充，这就意味着用GR表示的系统总是被映射到PR，但是PR表示法却不见得能用GR完全地加以表示。

图D—207示出一功能块交互作用图中功能块B的内部结构，以及其相应的PR表示。

图D—208示出进程PRI的一个简单进程定义的例子，它用GR和PR两种形式给出，该图只示出了有图形表示的部分。

对于功能块交互作用图，GR结构与PR结构的对应关系，如图D—209所示。

图D—210表示了进程中GR结构与PR结构的对应关系。



a) SDL

```
STATE1:
   RECEIVE CASE
      (S1 in m): case m.id of
                    (SGA): . . . . . ;
                    (SGB): . . . . . ;
                    (SGC): . . . . . ; JOIN:=1;
                    ELSE nextstate:=state1;
                 esac;
ESAC STATE1;
```

b) CHILL

图 D—205

映射JOIN的例子

D.9 SDL应用举例

D.9.1 引言

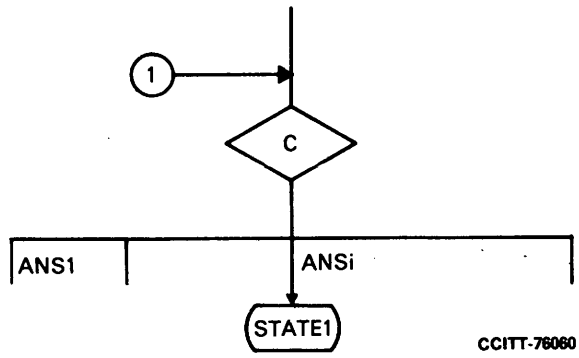
D.9章含有三个使用SDL的例子。这些例子取自电讯应用领域，并利用了各种不同的SDL子集，我们尽量采用实际例子，并介绍尽可能多的SDL概念。

在每个例子中，均在一高的抽象层次上表示一个系统，而且只有该系统的一部分才详加描述，其它部分声明为“未定义的”。

对系统定义的一个基本要求是，其所有各个部分应明确地结合起来成为一完整的整体（见§D.5）。因为这些建议没有提供满足这个要求的语言结构，必须在这些例子范围内准备这样的结构，这些引用结构是以SDL语法规则为基础的。

因为有些用户可能喜欢不同种类的引用结构，故在例子中用了两种主要方法，一种方法是基于SDL/PR语法，另一种方法是在图形中使用注释。

这些例子只用来表明SDL的应用，而不是国际规范。



a) SDL

```
1: CASE C OF
     (ANS1): ..... ;
     (ANS2): ..... ;
       .
       .
       .
     (ANSi): ..... ; nextstate := state 1 ;
   ESAC 1 ;
```

b) CHILL

图 D—206

映射DECISION的例子

D.9.2 电话交换系统

在本例中只使用基本的SDL（建议Z．101）。

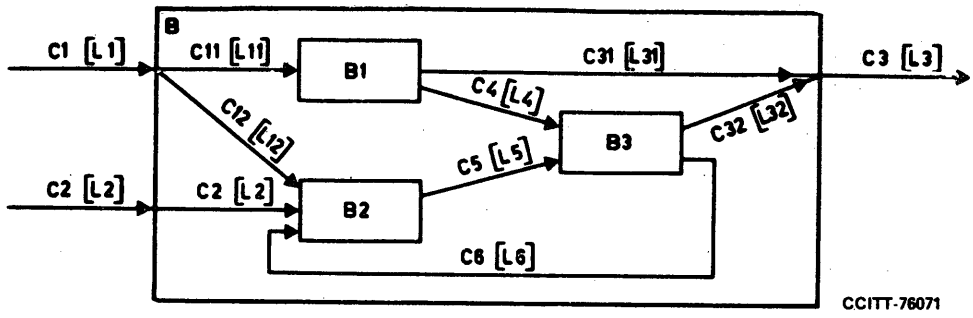
为便于互相对比，系统定义用SDL的所有具体语法形式表示。

本例涉及一电话交换系统中的呼叫处理进程，并认为读者已有了电话学的基本知识。第一阶段（呼叫建立）在主叫用户（A）和被叫用户（B）之间建立连接，两者都属于同一交换局；在第二阶段（通话），进程，待通话结束的信号；最后阶段（通话结束）用户断开，进程恢复到它的空闲状态。

D.9.2.1 在两种表达方式中的SDL/GR语法形式

通常使用的图形语法形式（面向跃迁的表达方式）不利用状态图形，而另一种形式（面向状态的表达方式）却利用状态图形。功能块交互作用图对两种形式都一样。

在面向状态表达方式的例子中不包含变量定义，且计时器的设置和复位是隐式表示的。还要注意，下一状态符号用的是一缩小的状态符号，符号内仅含有足以作为标识的状态序号。



a) GR 语法

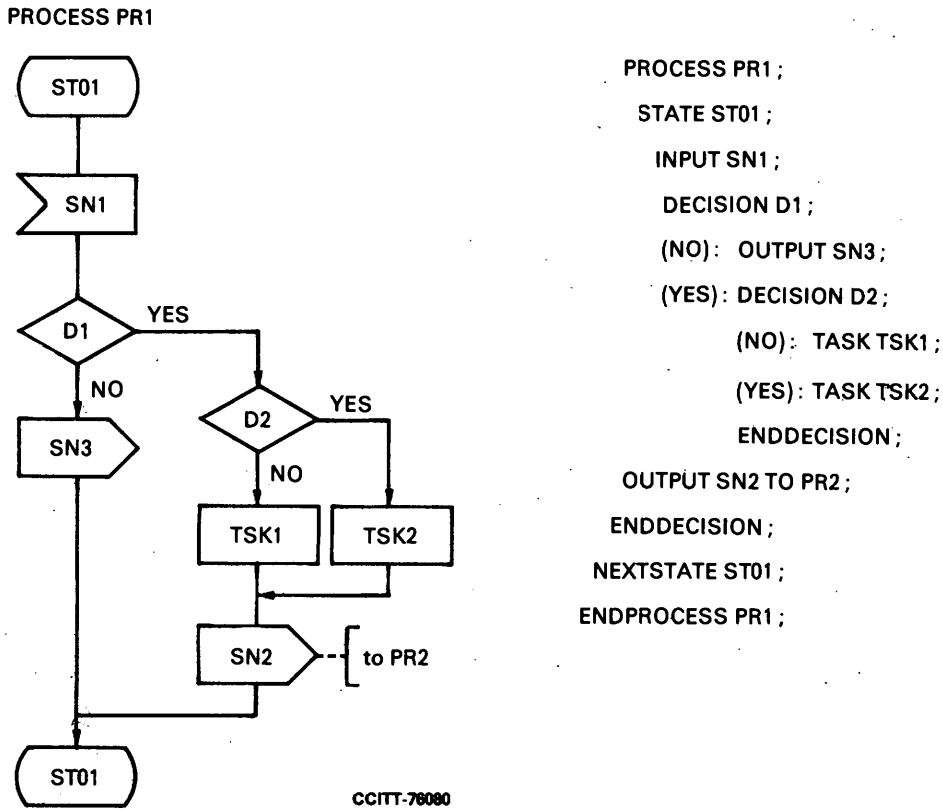
```
BLOCK B;
   SUBSTRUCTURE;
   SUBBLOCKS: B1, B2, B3;
   SPLIT C1 INTO C11, C12;
   SPLIT C2 INTO C2;
   SPLIT C3 INTO C31, C32;
   CHANNEL C11 FROM ENV TO B1 WITH /* L11 */;
   CHANNEL C12 FROM ENV TO B2 WITH /* L12 */;
   CHANNEL C2  FROM ENV TO B2 WITH /* L2 */;
   CHANNEL C31 FROM B1 TO ENV WITH /* L31 */;
   CHANNEL C32 FROM B3 TO ENV WITH /* L32 */;
   /*新的信道*/
   CHANNEL C4 FROM B1 TO B3 WITH /* L4 */;
   CHANNEL C5 FROM B2 TO B3 WITH /* L5 */;
   CHANNEL C6 FROM B3 TO B2 WITH /* L6 */;
   ENDSUBSTRUCTURE;
ENDBLOCK B;
```

b) PR 语法

注－根据语法规则，PR 程序是不完全的。

图 D—207

功能块交互作用图的GR与PR之间的对应关系



注-D 1和D 2是形式地定义的数据，PR 2在PR 例子中是PID类型的数据，但在GR 例子中可以是进程名字。

图 D—208

进程图的GR与PR之间的对应关系

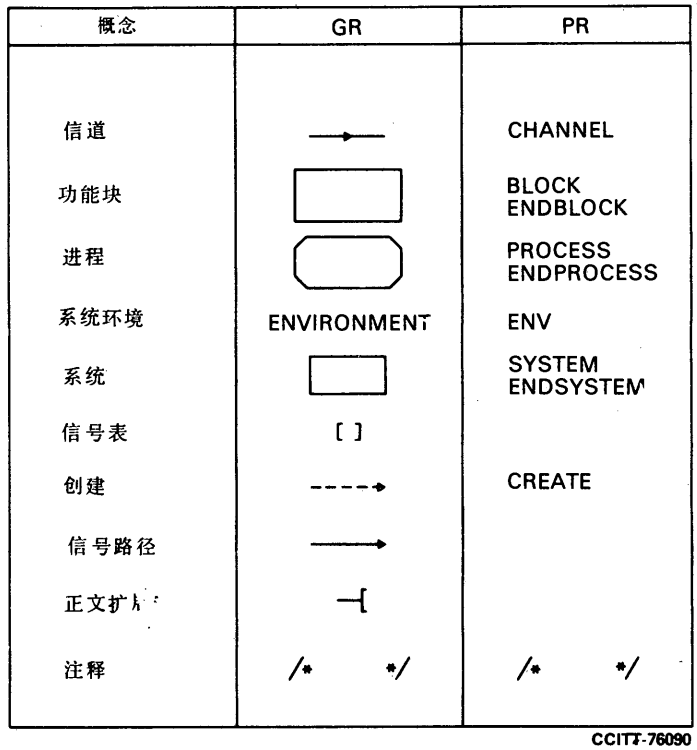
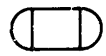
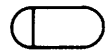
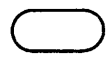


图 D—209

功能块交互作用图中所用符号的GR和PR结构的对应关系

概念	GR	PR
状态		STATE
下一状态		NEXTSTATE
输入		INPUT
输出		OUTPUT
任务		TASK
判定		DECISION ENDDECISION
入连接符		x:
出连接符		JOIN x
正文扩展		
注释		COMMENT or /*.....*/
合并跃迁		JOIN x x:
全部	*	*
全部（除了…）	* [.....]	* [.....]
保存		SAVE
过程调用		CALL
过程启动		PROCEDURE
过程停止		RETURN
宏调用		MACRO
宏入口处		MACRO EXPANSION
宏出口处		ENDMACRO
启动		START
停止		STOP
创建请求		CREATE
连续信号	< >	PROVIDED
允许条件		INPUT... PROVIDED
任选		ALTERNATIVE ENDALTERNATIVE

CCITT-76101

图 D—210

进程图内GR和PR结构的对应关系

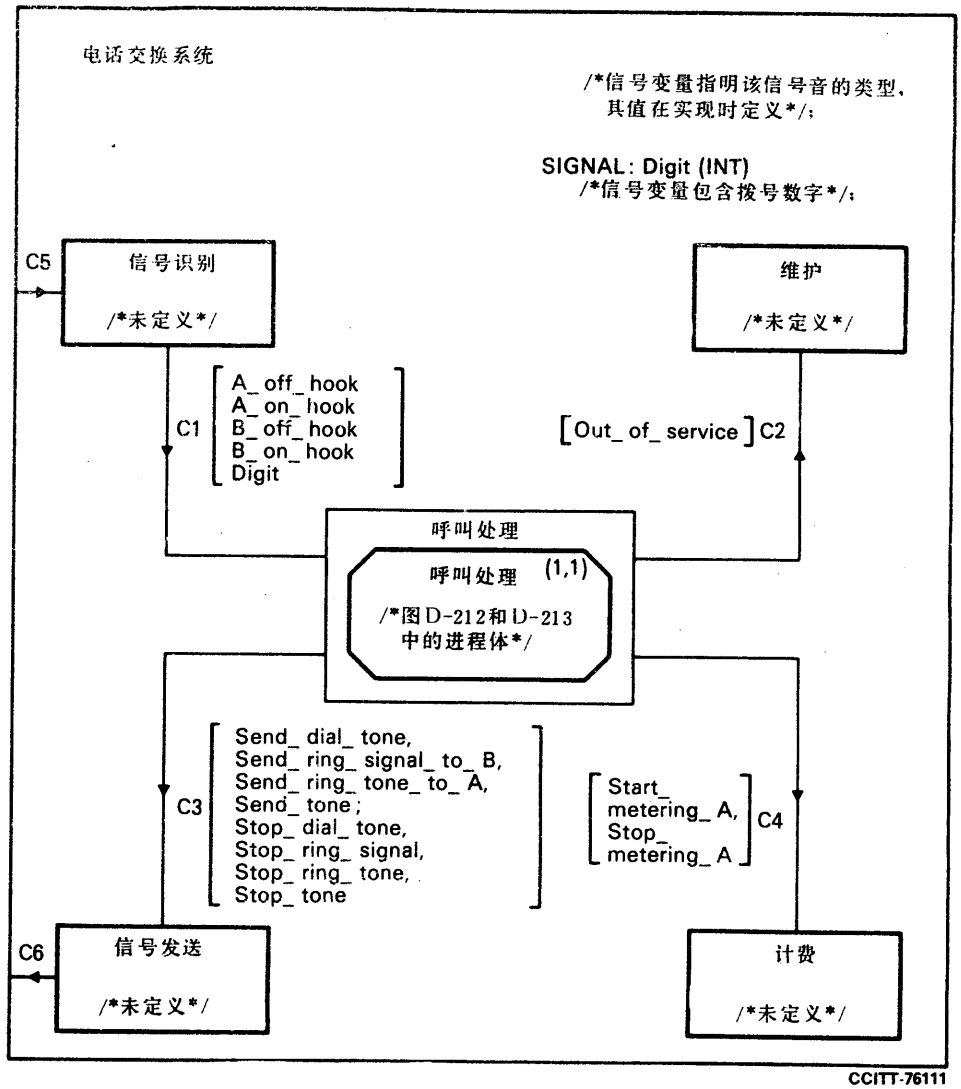


图 D—211
电话交换系统功能块交互作用图

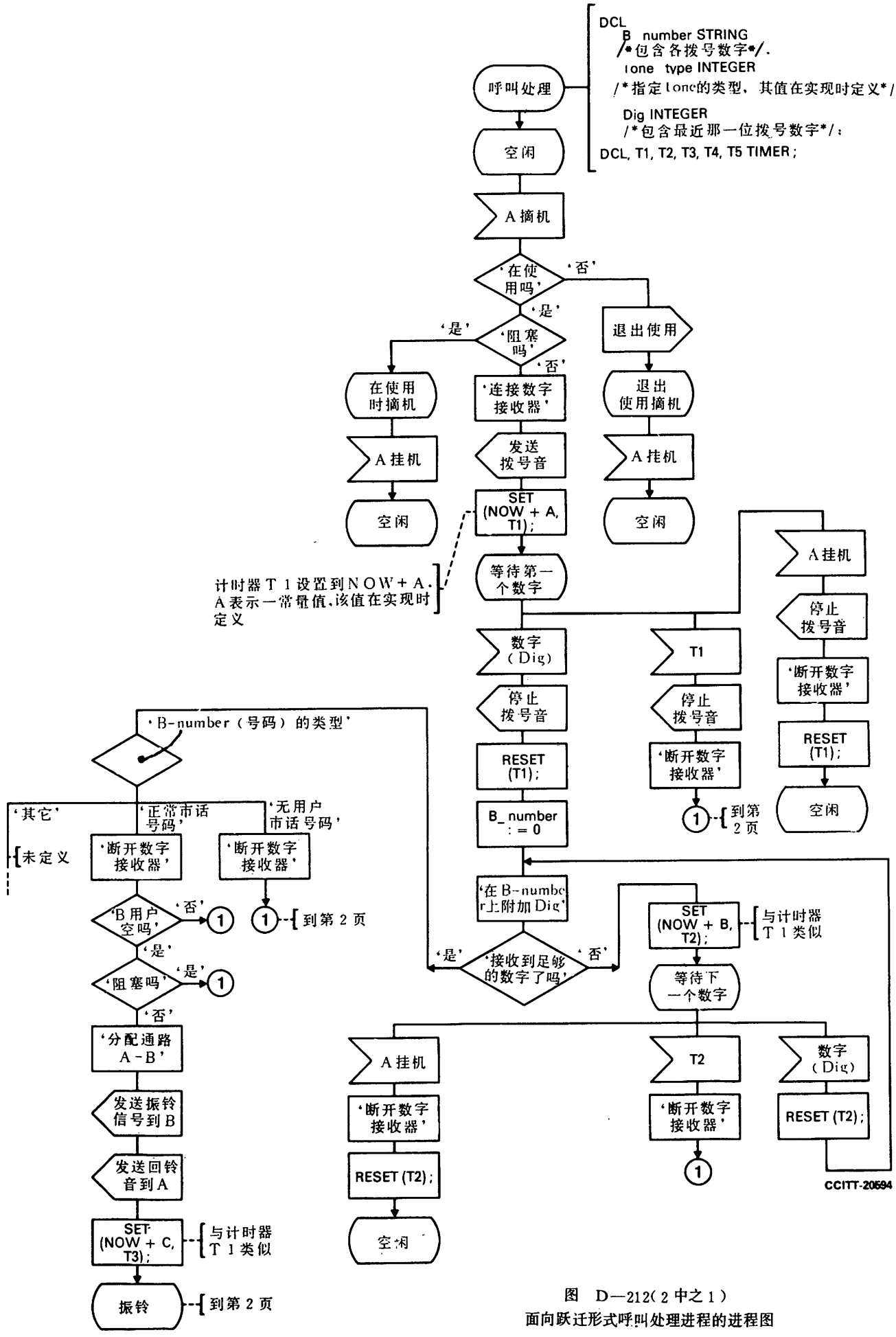


图 D—212（2 中之 1）
面向跃迁形式呼叫处理进程的进程图

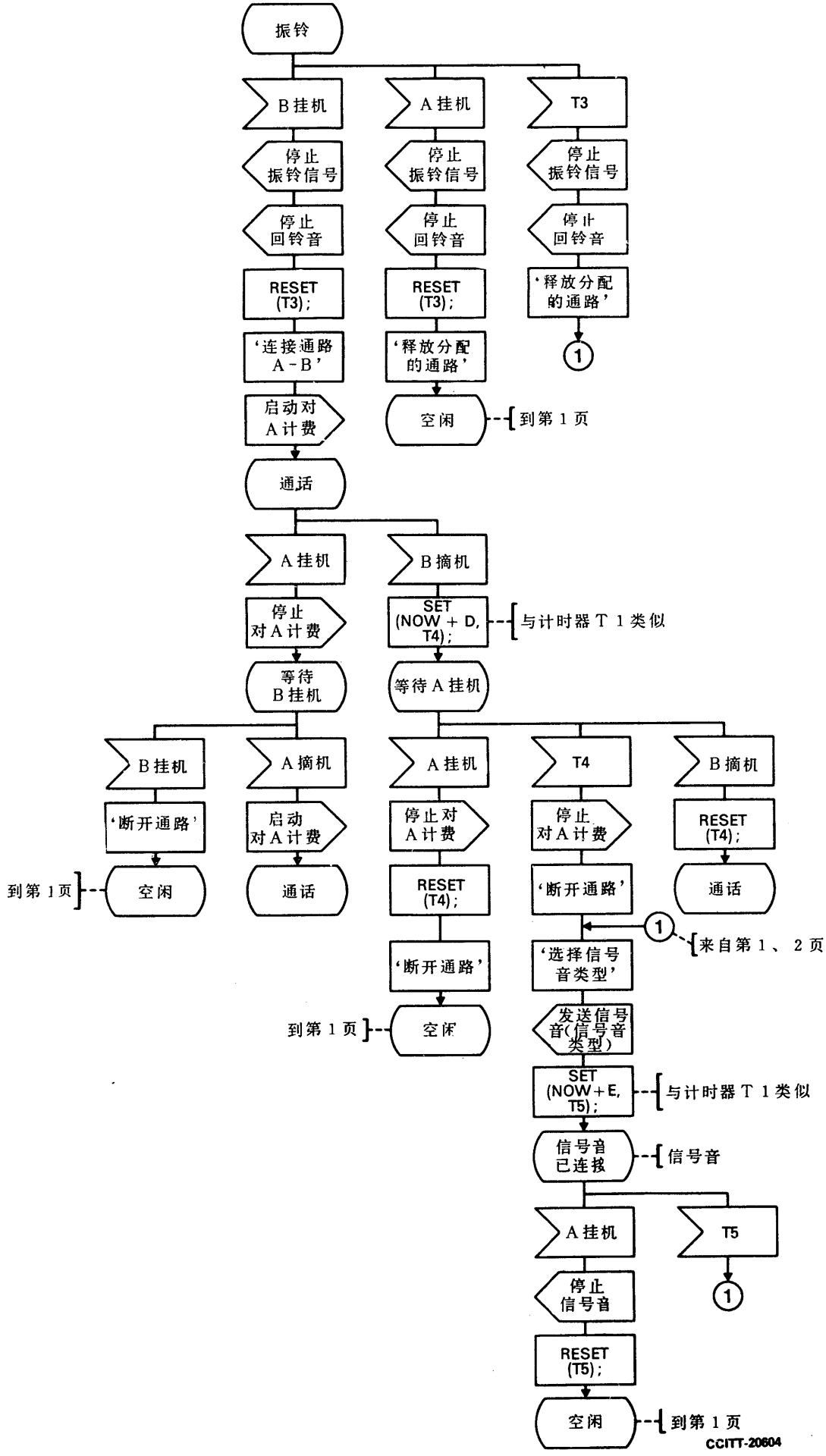
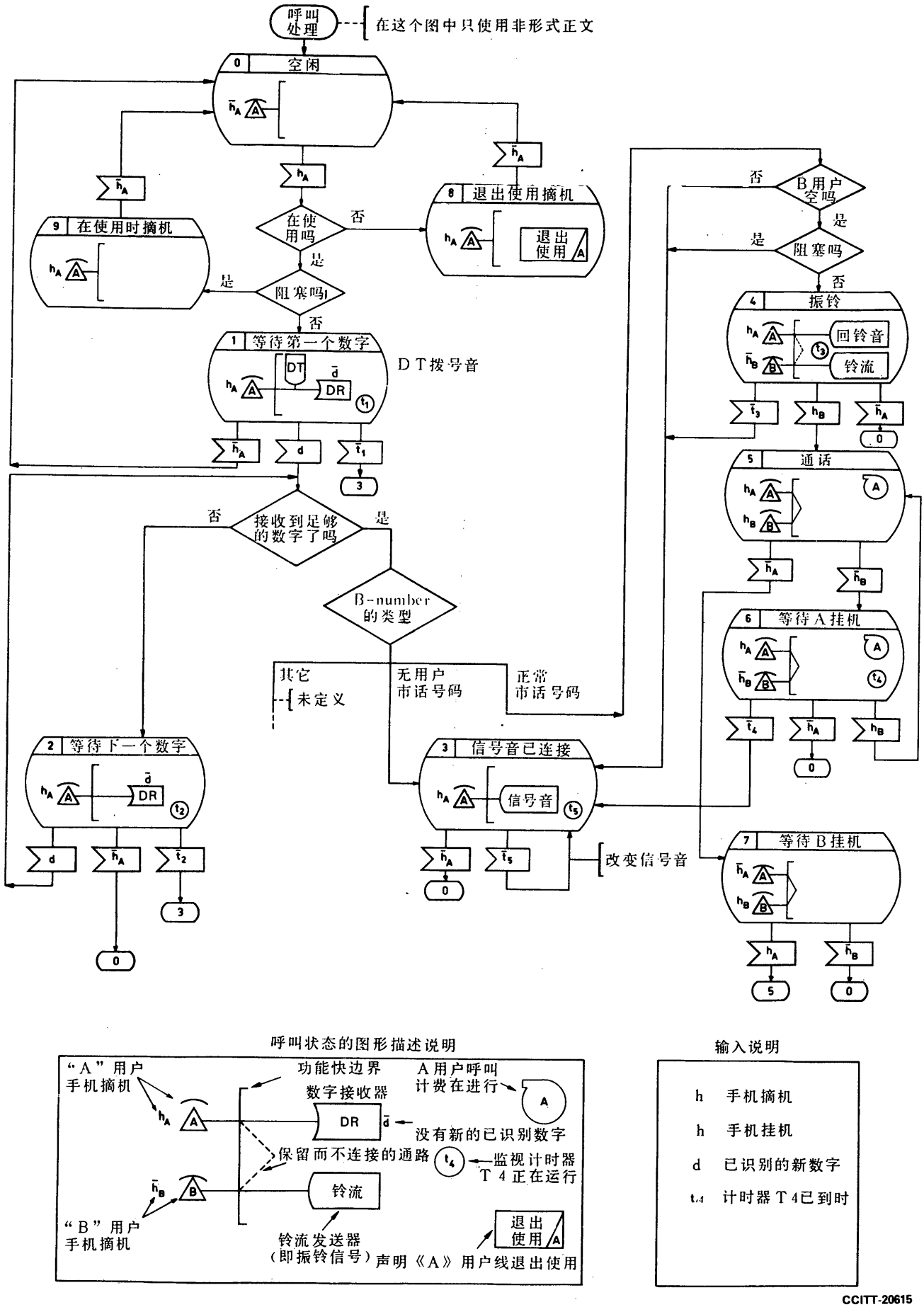


图 D—212(2中之2)

面向跃迁形式呼叫处理进程的进程图



呼叫状态的图形描述说明

输入说明

h	手机摘机
h	手机挂机
d	已识别的新数字
t_4	计时器T4已到时

CCITT-20615

图 D—213

面向状态形式，呼叫处理进程的进程图

D.9.2.2 SDL/PR语法形式

```
SYSTEM Telephone_switching_system;
        CHANNEL C1 FROM Signal_recognition TO Call_handling
                WITH    A_off_hook,
                        A_on_hook,
                        B_off_hook,
                        B_on_hook,
                        Digit;
        CHANNEL C2 FROM Call_handling TO Maintenance
                WITH    Out_of_service;
        CHANNEL C3 FROM Call_handling TO Signal_ sending
                WITH    Send_dial_tone,
                        Send_ring_signal_to_B,
                        Send_ring_tone_to_A,
                        Send_tone,
                        Stop_dial_tone,
                        Stop_ring_signal,
                        Stop_ring_tone,
                        Stop_tone;
        CHANNEL C4 FROM Call_handling TO Metering
                WITH    Start_metering_A,
                        Stop_metering_A;
        CHANNEL C5 FROM ENV TO Signal_recognition
                WITH     /*未定义*/;
        CHANNEL C6 FROM Signal  sending to ENV
                WITH     /*未定义*/;
        SIGNAL Send_tone (INT)
                /*信号变量指明tone（信号音）的类型，
                  其值在实现时定义*/;
        SIGNAL Digit (INT)
                /*信号变量包含拨号数字*/;


BLOCK Signal_recognition /*未定义*/;
ENDBLOCK;
BLOCK Signal_sending /*未定义*/;
ENDBLOCK;
BLOCK Maintenance /*未定义*/;
ENDBLOCK;
BLOCK Metering /*未定义*/;
ENDBLOCK;
```

图 D—214（4 中之 1）

使用SDL/PR 语法的系统定义

```
BLOCK Call_handling;
    PROCESS Call_handling (1,1);
        DCL B_number STRING
            /*包含拨号数字*/;
        DCL Tone_type INTEGER
            /*指明tone(信号音)的类型，其值在实现时定义*/;
        DCL Dig INTEGER
            /*包含最近那一位拨号数字*/;
        START; NEXTSTATE Idle;
        STATE Idle;
            INPUT A_off_hook;
                DECISION 'In service';
                    ('No'):   OUTPUT Out_of_service;
                              NEXTSTATE Off_hook_out_of_service;
                    ('Yes'):  DECISION 'Blocking';
                                  ('Yes'):  NEXTSTATE Off_hook_in_service;
                                  ('No'):   TASK 'Connect digit receiver';
                                            OUTPUT Send_dial_tone;
                                            TASK SET (NOW +A, T1);
                                            /*计时器T1设置到NOW+A，A表示一个常量值，
                                              其值在实现时定义*/;
                                            NEXTSTATE Await_first_digit;
                              ENDDECISION;
                ENDDECISION;
        STATE Off_hook_in_service;
            INPUT A_on_hook;
                NEXTSTATE Idle;
        STATE Off_hook_out_of_service;
            INPUT A_on_hook;
                NEXTSTATE Idle;
        STATE Await_first_digit;
            INPUT Digit (Dig);
                OUTPUT Stop_dial_tone;
                TASK RESET (T1),
                    B number:=0;
            10: TASK 'Append Dig to B_number';
                DECISION 'Sufficient number of digits received';
                    ('No'):   TASK SET (NOW +B,T2);
                              NEXTSTATE Await_next_digit;
                    ('Yes'):  DECISION 'Type of B_number';
                                  ('Local normal'): TASK 'Disconnect digit receiver';
                                      DECISION 'B party free';
                                          ('No'):   JOIN 1;
                                          ('Yes'):  DECISION 'Blocking';
                                                        ('Yes'):  JOIN 1;
                                                        ('No'):   TASK 'Allocate path A-B';
                                                                  OUTPUT Send_ring_signal_to_B;
                                                                  OUTPUT Send_ring_tone_to_A;
                                                                  TASK SET (NOW +C, T3);
                                                                  NEXTSTATE Ringing;
                                                    ENDDECISION;
                                      ENDDECISION;
                                  ('Local vacant'):   TASK 'Disconnect digit receiver';
                                                      JOIN 1;
                                  ('Other'):          /*未定义*/;
                              ENDDECISION;
                ENDDECISION;
```

图 D—214（4 中之 2）

使用SDL/PR语法的系统定义

```
    INPUT T1;
        OUTPUT Stop_dial_tone;
        TASK 'Disconnect digit receiver';
        JOIN 1;
    INPUT A_on_hook;
        OUTPUT Stop_dial_tone;
        TASK 'Disconnect digit receiver';
        TASK RESET (T1);
        NEXTSTATE Idle;

STATE Await_next_digit;
    INPUT Digit (Dig);
        TASK RESET (T2);
        JOIN 10;
    INPUT T2;
        TASK 'Disconnect digit receiver';
        JOIN 1;
    INPUT A_on_hook;
        TASK 'Disconnect digit receiver';
        TASK RESET (T2);
        NEXTSTATE Idle;

STATE Ringing;
    INPUT B_off_hook;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK RESET (T3);
        TASK 'Connect path A-B';
        OUTPUT Start_metering_A;
        NEXTSTATE Conversation;
    INPUT A_on_hook;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK RESET (T3);
        TASK 'Release allocated path';
        NEXTSTATE Idle;
    INPUT T3;
        OUTPUT  Stop_ring_signal,
                Stop_ring_tone;
        TASK 'Release allocated path';
        JOIN 1;

STATE Conversation;
    INPUT A_on_hook;
        OUTPUT Stop_metering_A
        NEXTSTATE Await_B_on_hook;
    INPUT B_on_hook;
        TASK SET (NOW +D, T4);
        NEXTSTATE Await_A_on_hook;
```

图 D—214（4 中之 3）

使用SDL/PR语法的系统定义

```
        STATE Await_B_on_hook;
            INPUT A_off_hook;
                OUTPUT Start_metering_A;
                NEXTSTATE Conversation;
            INPUT B_on_hook;
                TASK,'Disconnect path';
                NEXTSTATE Idle;
        STATE Await_A_on_hook;
            INPUT A_on_hook
                OUTPUT Stop_metering_A;
                TASK RESET (T4);
                TASK 'Disconnect path';
                NEXTSTATE Idle;
            INPUT B_off_hook;
                TASK RESET (T4);
                NEXTSTATE Conversation;
            INPUT T4;
                OUTPUT Stop_metering_A;
                TASK 'Disconnect path';
            1:  TASK 'Select tone type';
                OUTPUT Send_tone (Tone_type);
                TASK SET (NOW +E, T5);
                NEXTSTATE Tone_connected;


        STATE Tone_connected;
            INPUT A_on_hook;
                OUTPUT Stop_tone;
                TASK RESET (T5);
                NEXTSTATE Idle;
            INPUT T5;
                JOIN 1;
ENDPROCESS Call_handling;
ENDBLOCK Call_handling;
ENDSYSTEM;
```

图 D—214（4 中之 4）

使用SDL / PR 语法的系统定义

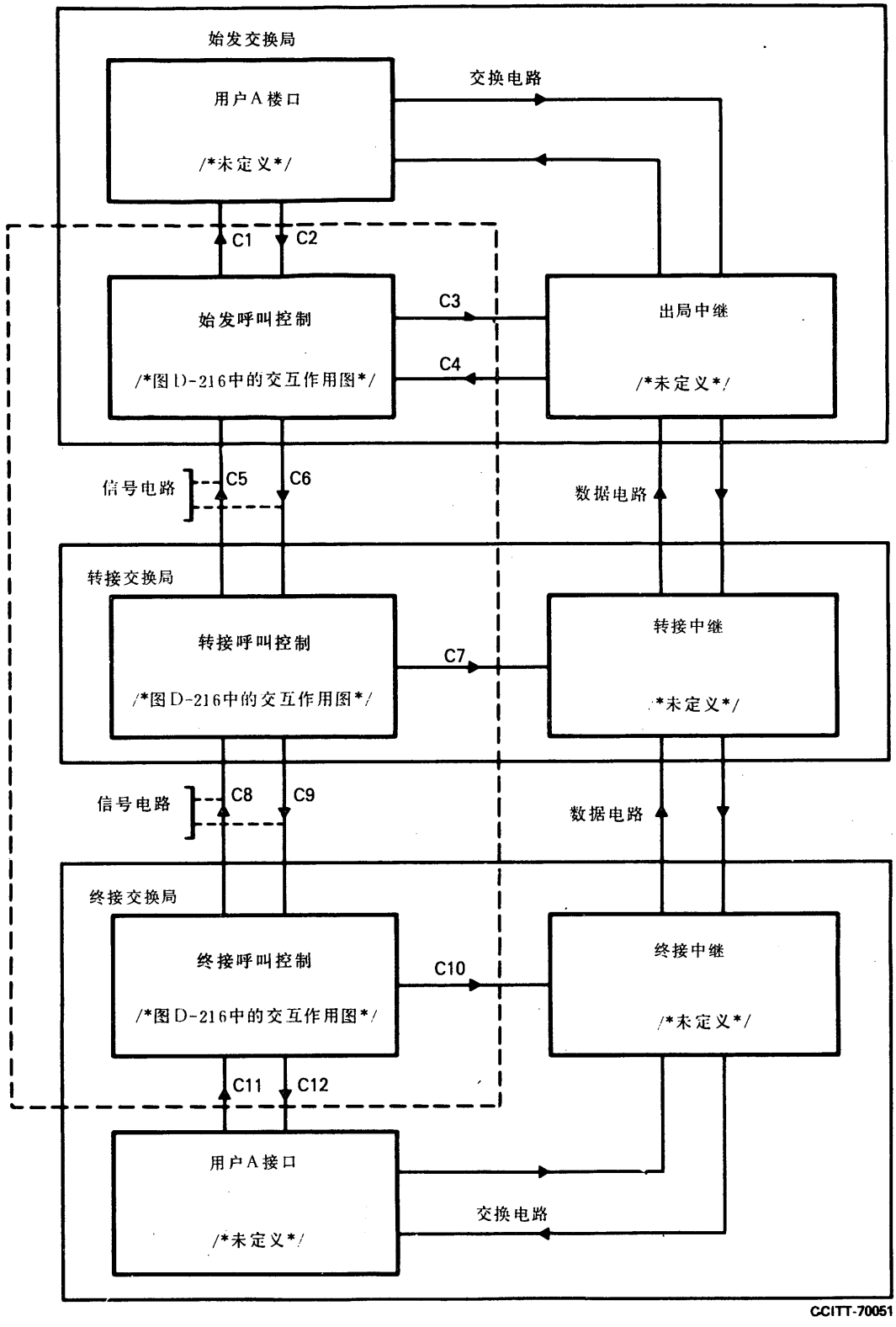
D.9.3 公共信道信号系统的数据用户部分

下面的例子是以建议X.61为基础的，它不一定和X.61相同，并且不打算用来作为国际标准。这个例子介绍建议Z.103中定义的过程的用法及任选概念的用法。

在使用公共信道信号的电路交换网络中，对一群互相交换的中继线路来说，信号消息是用一集中的信号网络来传递的。图D—215是具有三类交换局的电路交换数据网络的概要图，公共信道信号系统的数据用户部分用始发局、转接局和终接局中的呼叫控制功能块来表示。概要图不是规格说明的正式组成部分，但是包括在规格中，以便阐明该信号系统和数据网络其它组成部分之间的关系。该信号系统的功能块交互作用图如图D—216所示。

始发、转接和终接进程（图D—217、D—218和D—219）表示两个用户接口间建立、保持和结束一数据连接所必需的功能。在呼叫建立阶段，终接交换局对该入局呼叫可实现两种响应中的一种，这些供选择的方案是用任选符号来规定的。在终接交换局连接完成后，被叫用户接受呼叫时，就把一个“准备接受数据”信号发送回始发进程。

当通话结束时，始发、转接和终接各进程都执行类似的功能，以释放中继线路。这些功能是由图D—220和D—221中的全局过程所规定的。



注-公共信道信号系统的数据用户部分用虚线边界指明。

图 D—215

由三个交换局组成的电路交换数据网络概要图

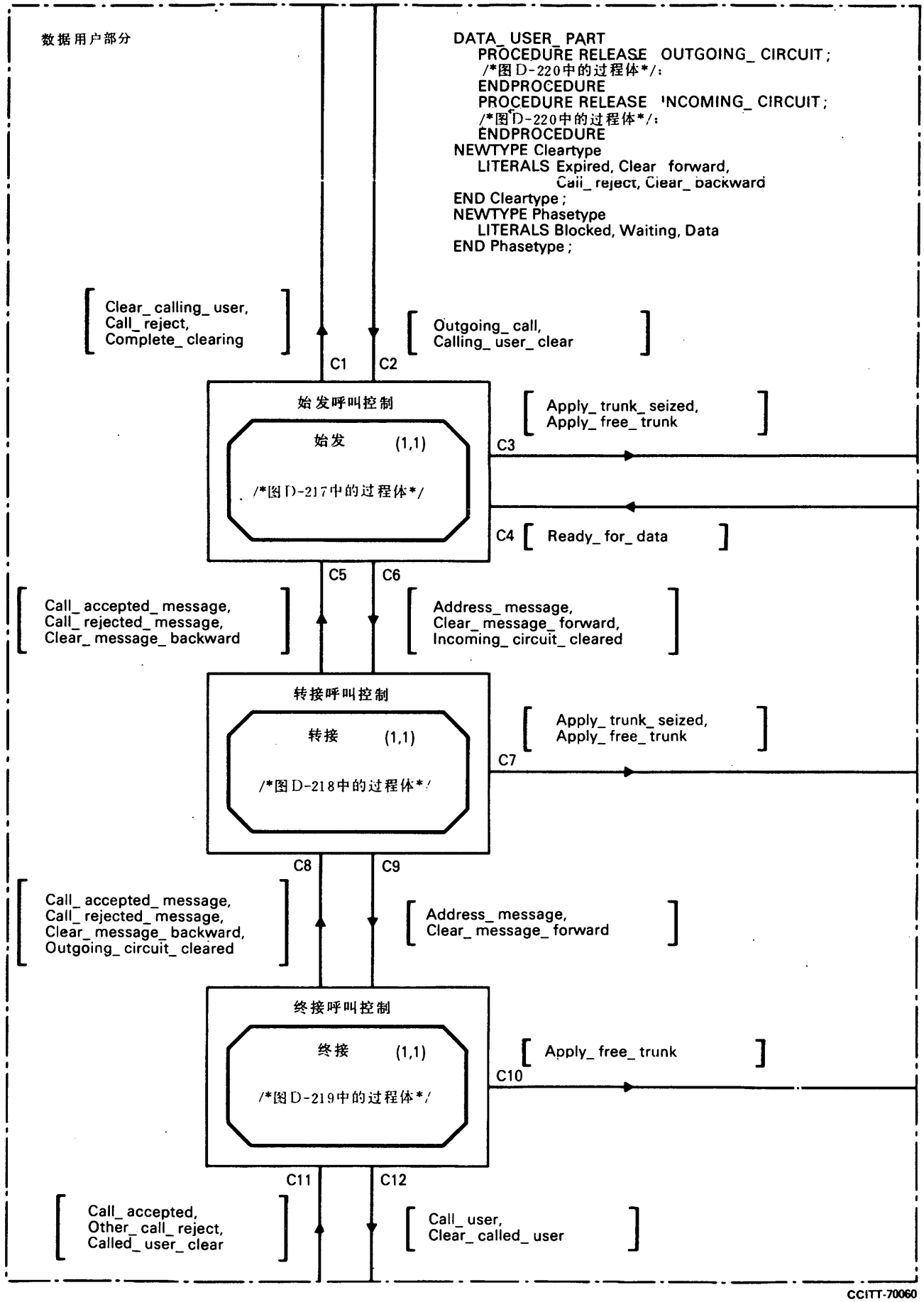


图 D—216

系统DATA—USER—PART的功能块交互作用图

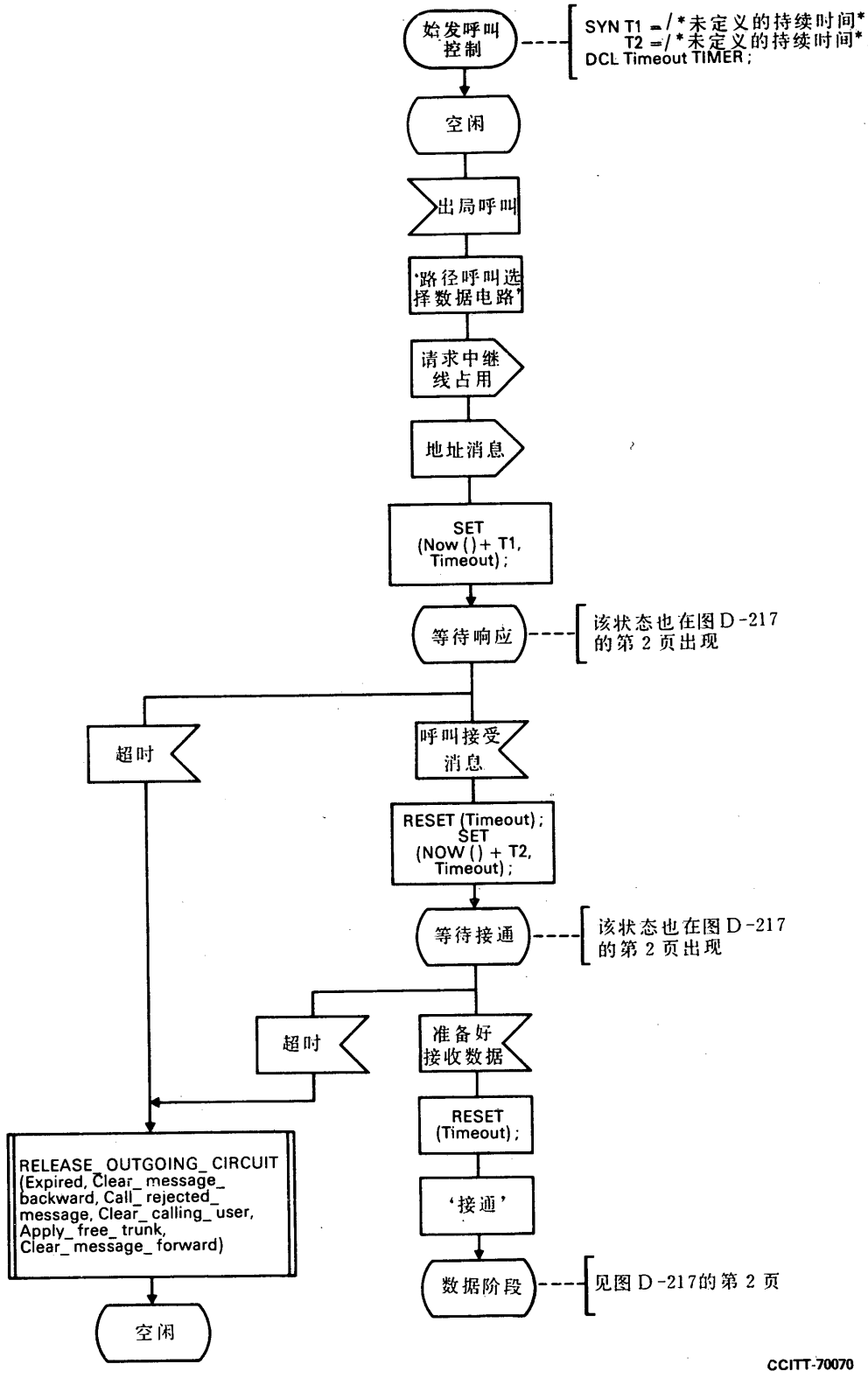


图 D—217（2 中之 1）

始发呼叫控制进程图

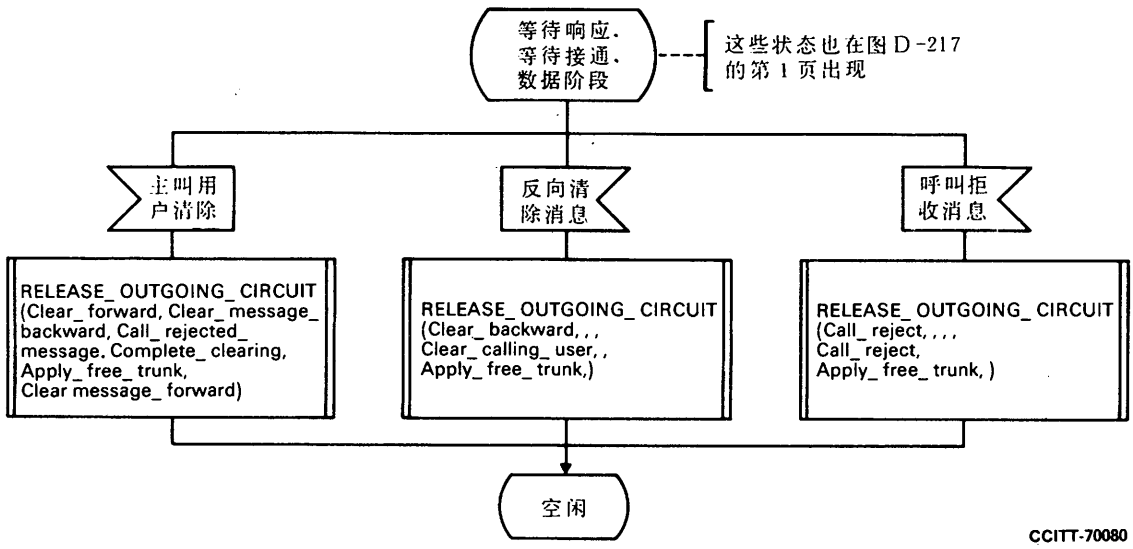


图 D—217（2 中之 2）

始发呼叫控制进程图

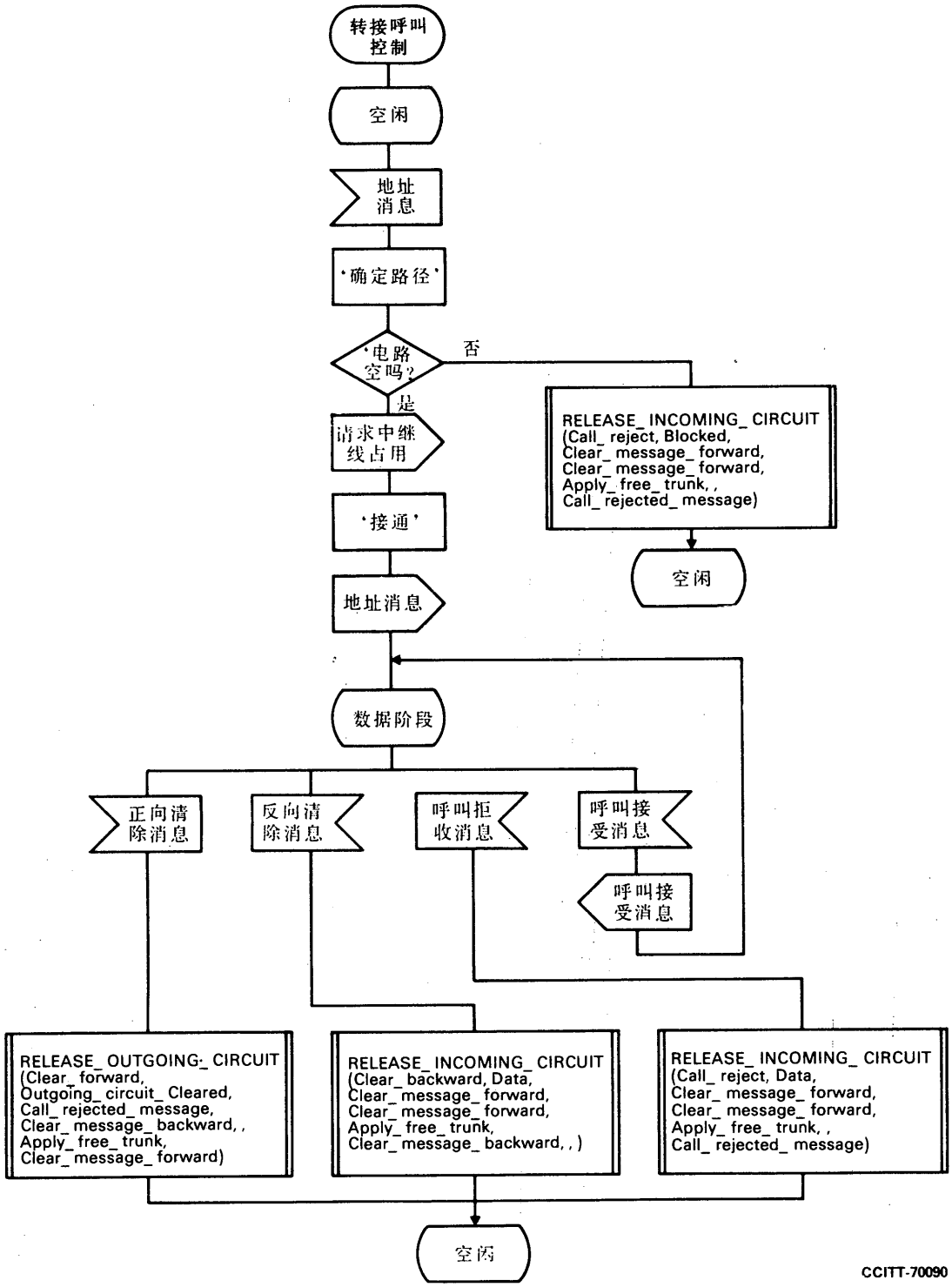


图 D—218

转接呼叫控制进程图

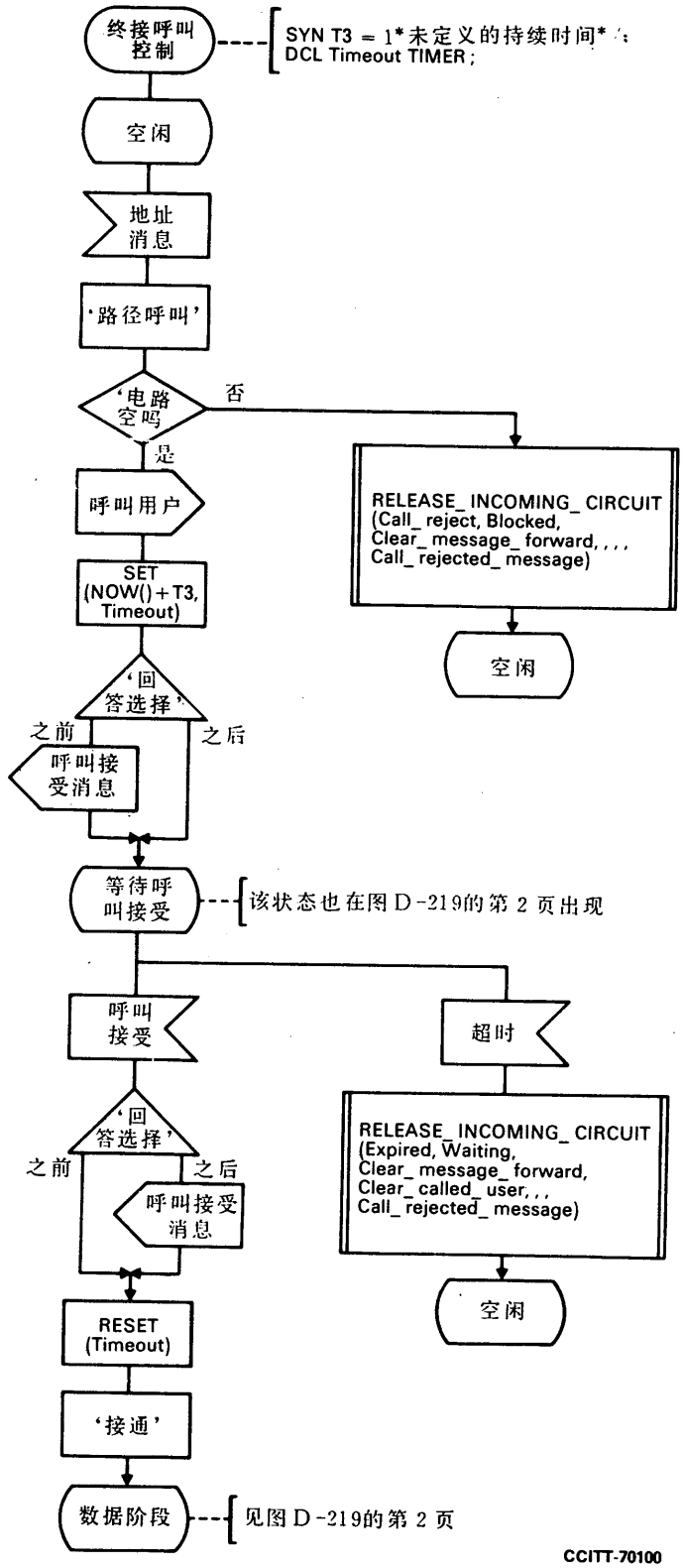


图 D—219 （2中之1）

终接呼叫控制进程图

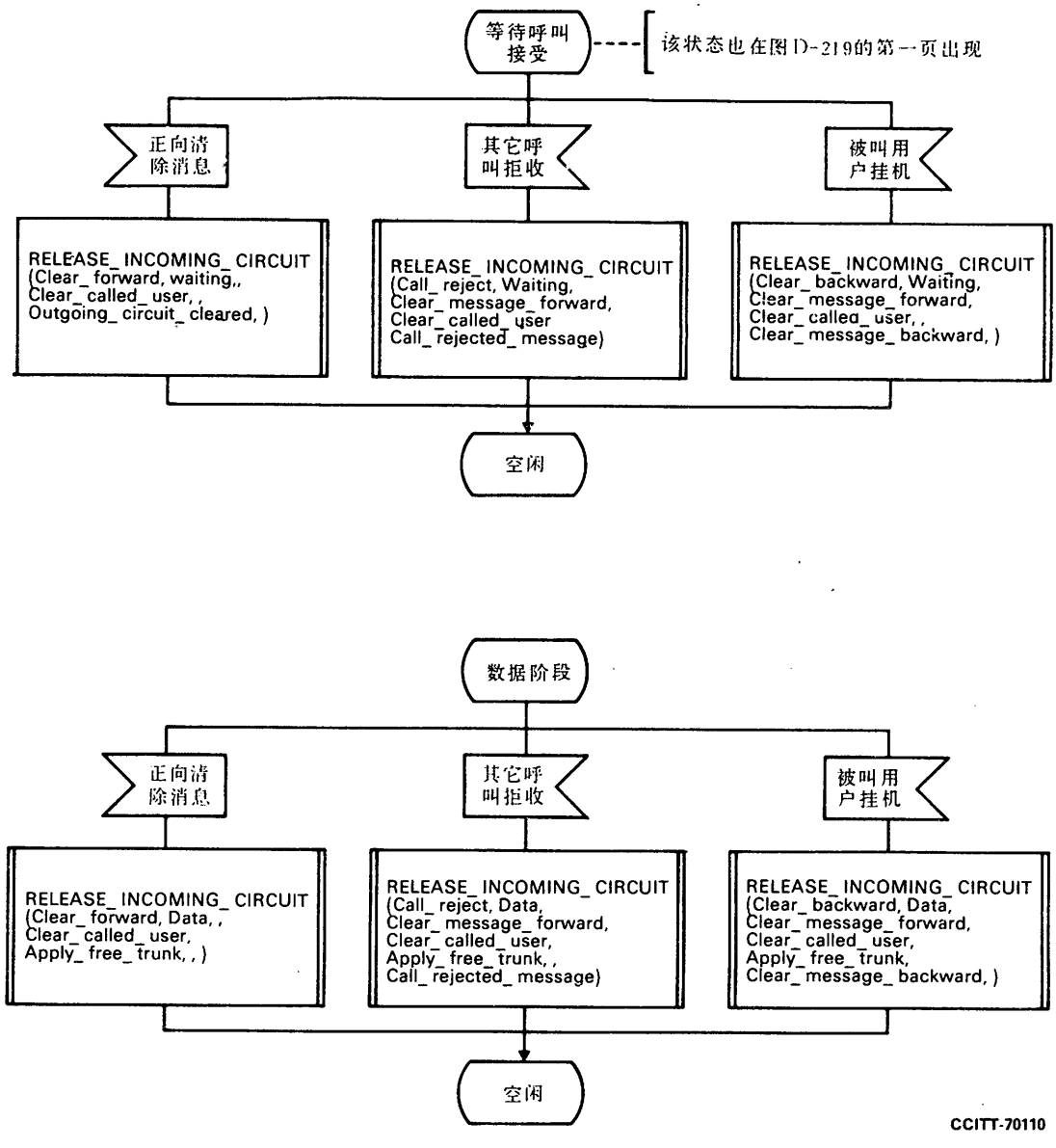


图 D—219（2 中之 2）

终接呼叫控制进程图

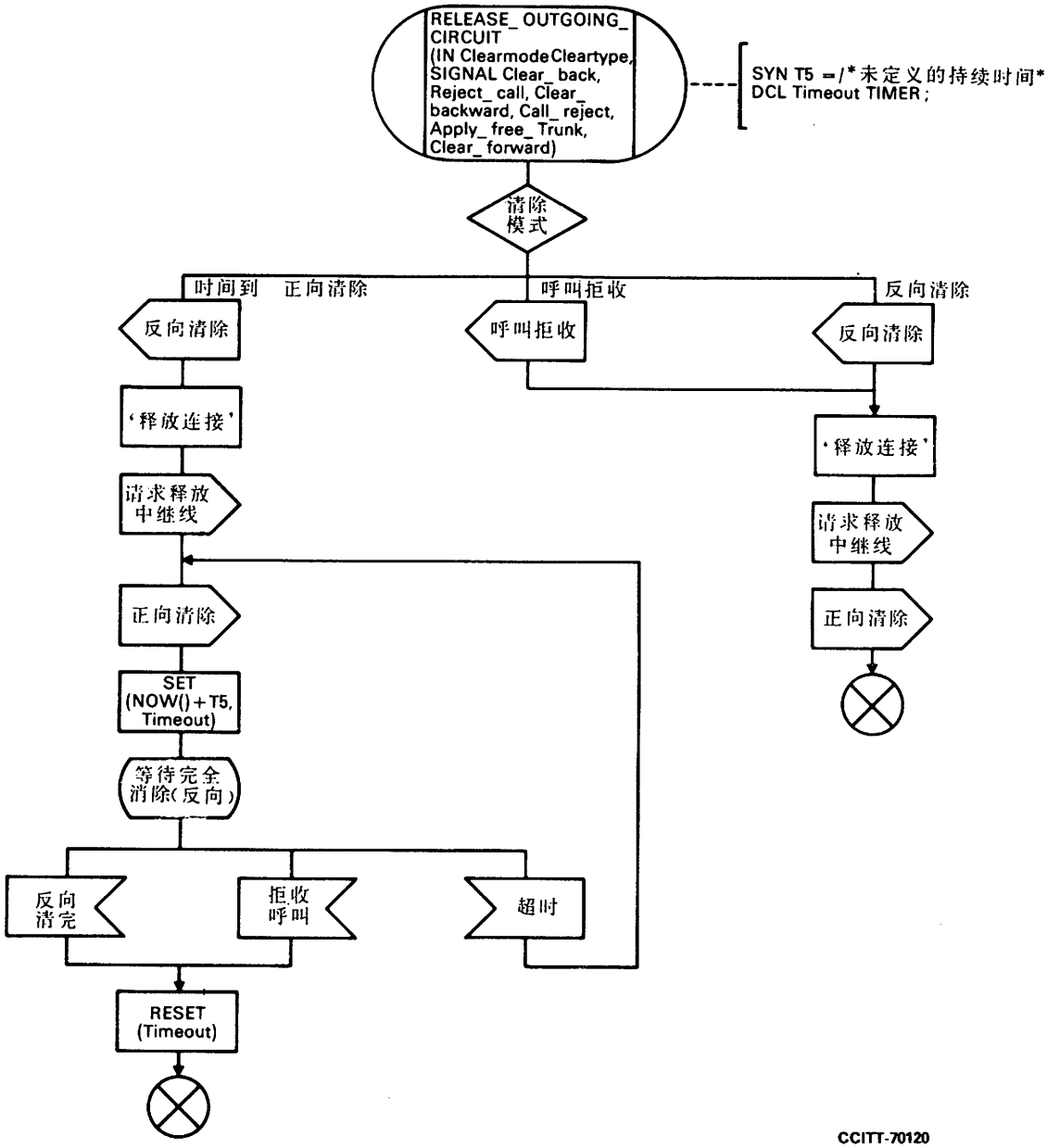


图 D—220

释放出局电路过程的过程图

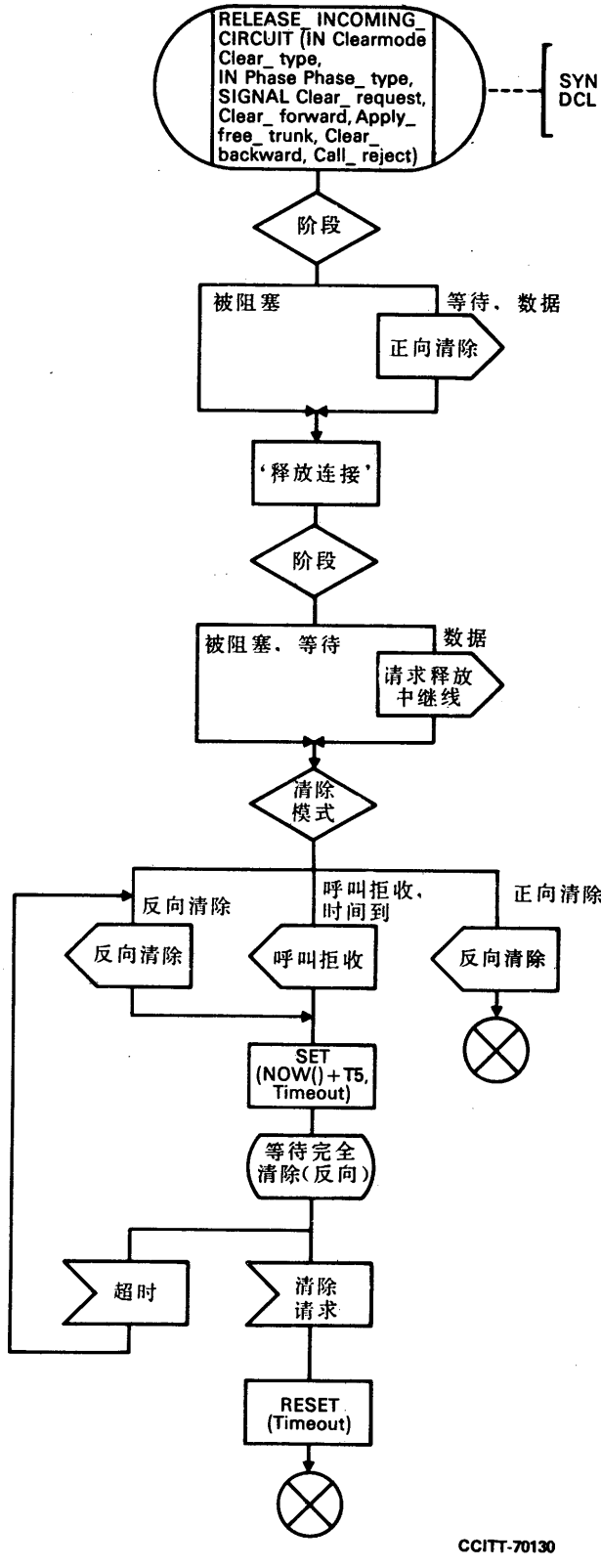


图 D—221

释放入局电路过程的过程图

D.9.4 0类传输协议

下面的例子以开放系统互连的传输协议建议为依据。该例子是传输协议的一个子集，未必与建议等效，且不打算用来作为国际标准。

遵循建议Z.101和Z.103的SDL已经在使用。虽然用了数据，但多少处于一种非形式方法，没有使用建议Z.104。已经模拟了Z.104的枚举类型，用的是一些INTEGER型变量，对未定义的各常量给出了名字。数据的运算（操作）还没有被形式地定义。

D.9.4.1 概述

OSI参考模型的传输层在各对用户之间设置通信信道（称为传输连接），为其各用户提供服务，在每个用户连接点，传输层用一传输实体表示，各传输实体使用传输协议和网络层的服务彼此进行通信。图D—223是一概览图，该例子描述了仅实现0类传输协议的一个传输实体。

在两个传输实体之间建立了一传输连接。规格说明所依据的该模型独立地描述一个传输实体的行为，并可应用于该连接的两端。图D—224的交互作用图指明了表示传输实体的单个功能块，它通过称为传输服务访问点（TSAP）和网络服务访问点（NSAP）的各信道与环境连接，传输服务用户和网络层被看作是环境。

传输实体包含两个进程，第一个是DIRECTOR，它具有一个总是存在的实例（用进程符号右上角的数字指明）；第二个进程是ENDPOINT，它只具有一些动态的创建的实例，其中之一是在建立传输连接的每个尝试的生存期间创建的。如果DIRECTOR收到一个信号‘T—CONNECT请求’或‘N—CONNECT指示’，它就立即创建一新的ENDPOINT实例，用以处理与该连接尝试有关的所有后继信号。每个ENDPOINT实例有一个‘种类’特性，具有值INITIATOR或值RESPONDER，用来确定该实例的初始行为；当连接被其中一个传输服务用户所结束，或在出现一个错误之后，该实例本身就结束了。

DIRECTOR的行为用图D—225中的进程图给出，ENDPOINT的行为用图D—226～图D—230给出。ENDPOINT的规格利用各SDL过程来说明，包括建立连接、传递数据及结束各阶段。

D.9.4.2 系统的SDL规格

系统规格是用下面的SDL—PR正文来表示的。该正文与各建议的区别在于其某些部分用SDL/GR形式给出，而以注释形式在SDL—PR中引用这些部分。

```
SYSTEM TRANSPORT_ENTITY;
    /*在两用户间的传输连接中，TRANSPORT ENTITY(传输实体)的作用示于图D-223的概览图。*/
    /*信道和有关信号示于图D-224的交互作用图*/

  /*输入表*/
    T_CONN_REQ          (PID,STRING)    ,
    T_CONN_RSP          (PID,STRING)    ,
    T_CONN_RPLY                         ,
    T_DATA_REQ          (STRING)        ,
    T_DISC_REQ                          ,
    N_CONN_IND          (PID,STRING)    ,
    N_CONN_CFM          (PID,STRING)    ,
    N_CONN_RPLY                         ,
    N_DATA_IND          (STRING)        ,
    N_DISC_IND                          ,
    N_RESET_IND                         ;
      MACRO EXPANSION signals;
        T_CONN_REQ, T_CONN_RSP, T_CONN_RPLY, T_DATA_REQ, T_DISC_REQ, N_CONN_
    IND, N_CONN_CFM, N_CONN_RPLY, N_DATA_IND, N_DISC_IND,
        N_RESET_IND,
      ENDMACRO signals;
  /*输出表*/
    T_CONN_IND          (PID,STRING)    ,
    T_CONN_CFM          (PID,STRING)    ,
    T_DATA_IND          (STRING)        ,
    T_DISC_IND                          ,
    N_CONN_REQ          (PID,STRING)    ,
    N_CONN_RSP          (PID,STRING)    ,
    N_DATA_REQ          (STRING)        ,
    N_DISC_REQ                          ;
```

图 D—222 （3 中之 1）

使用SDL/PR的系统规格

```
BLOCK TRANSPORT_ENTITY;
    PROCESS DIRECTOR (1,1);
        /*此进程的唯一实例控制各ENDPOINT的赋值*/
        DCL conn_params STRING,
            conn_id PID ;
        /*图D-225进程图给出进程体*/
    ENDPROCESS DIRECTOR;
        PROCESS ENDPOINT (0, ),
            /*DIRECTOR为每个所请求的连接创建此进程的一个实例*/
            FPAR
                kind            INT /*值INITIATOR,RESPONDER*/,
                conn_params     STRING ,
                conn_id         PID;
            DCL
                ref             STRING ,
                tc_id           PID ,
                nc_id           PID ,
                tpdu_size       INT   /*值*/
                cstat           INT   /*值DATA TRANSFER, DISCONNECTED*/;
                    IMPORTED /*来自tc-id通过信道TSAP-in*/
                t_ind_flo       BOOL ,
                    IMPORTED /*来自nc-id通过信道NSAP-in*/
                n_req_flo       BOOL ,
                    EXPORTED /*到tc-id通过信道TSAP-out*/
                t_req_flo       BOOL ,
                    EXPORTED /*到nc-id通过信道NSAP-out*/
                n_ind_flo       BOOL ;

    PROCEDURE INITIATE_CONNECT;
        SIGNAL      MACRO signals;
        IN          conn_params     STRING ,
        IN          tc_id           PID ,
        IN/OUT      nc_id           PID ,
        IN/OUT      tpdu_size       INT ,
        IN/OUT      cstat           INT /*值DATA-TRANSFER, DISCONNECTED*/
      DCL
        n_conn_params STRING ,
        prop_tpdu_size INT ,
        tpdu_flag       INT /*值CC, DR, ER, CR, DT, ISV*/,
        tpdu_data       STRING ,
        nbin            STRING ,
        nbout           STRING ,
        tbout           STRING ,
        taskr           INT /*值CLOSE, NC, TC, NC&TC, NONE*/;
      /*图D-227过程图给出过程体*/
    ENDPROCEDURE INITIATE_CONNECT;
    PROCEDURE RESPOND_CONNECT
        SIGNAL      MACRO signals;
        IN          conn_params     STRING ,
        IN/OUT      tc_id           PID ,
        IN          nc_id           PID ,
        IN/OUT      tpdu_size       INT ,
        IN/OUT      cstat INT /*值DATA-TRANSFER, DISCONNECTED*/;
      DCL
        n_conn_params STRING ,
        prop_tpdu_size INT ,
        tpdu_flag       INT /*值CC, DR, ER, CR, DT, ISV*/,
        tpdu_data       STRING ,
        nbin            STRING ,
        nbout           STRING ,
        tbin            STRING ,
        tbout           STRING ,
        taskr           INT /*值CLOSE, NC, TC, NC&TC, NONE*/
      /*图D-228过程图给出过程体*/
    ENDPROCEDURE RESPOND_CONNECT;
```

图 D—222(3 中之 2)

使用SDL/PR的系统规格

```
    PROCEDURE DATA_PHASE;
        SIGNAL      MACRO signals;
        IN          ref             STRING ,
        IN          tc_id           PID ,
        IN          nc_id           PID ,
        IN          tpdu_size       INT ,
        IN/OUT      cstat           INT /*值DATA-TRANSFER, DISCONNECTED*/,
        IN/OUT IMPORTED /*来自tc-i 通过信道TS AP-in*/
        t_ind_flo                   BOOL ,
        IN/OUT IMPORTED /*来自nc-id通过信道NS AP-in*/
        n_req_flo                   BOOL ,
        IN/OUT EXPORTED /*到tc-id通过信道TS AP-out*/
        t_req_flo                   BOOL ,
        IN/OUT EXPORTED /*到nc-id通过信道NS AP-out*/
        n_ind_flo                   BOOL ;
      DCL
        input_present               BOOL ,
        output_present              BOOL ,
        tbin                        STRING ,
        nbin                        STRING ,
        tbout                       STRING ,
        nbout                       STRING ,
        output_segment              STRING ,
        input_tsdu                  STRING ,
        tpdu_flag                   INT /*值CC, DR, ER, CR, DT, INV*/,
        taskr                       INT /*值CLOSE, NC, TC, NC&TC, NONE*/;
      /*图D-229过程图给出过程体*/
    ENDPROCEDURE DATA_PHASE;
    PROCEDURE RELEASE_CONNECT
        SIGNAL      MACRO signals;
        IN          taskr           INT /*值CLOSE, NC, TC, NC&TC, NONE*/,
        IN          tc_id           PID ,
        IN          nc_id           PID ,
        IN/OUT      cstat           INT /*值DATA-TRANSFER, DISCONNECTED*/,
        IN/OUT      tpdu_flag       INT /*值CC, DR, ER, CR, DT, INV*/,
        IN/OUT      last_tpdu       STRING;
      DCL
        nbin                        STRING ,
        nbout                       STRING ,
        tbout                       STRING ;
      /*图D-230过程图给出过程体*/
    ENDPROCEDURE RELEASE_CONNECT;
  /*图D-226进程图给出过程体*/
  ENDPROCESS ENDPOINT;
ENDBLOCK TRANSPORT_ENTITY;
ENDSYSTEM TRANSPORT_ENTITY;
```

图 D—222(3中之3)
使用SDL/PR的系统规格

D.9.4.3 对数据项的运算

对数据项的运算是非形式地表示的，它们是：

alloc-ref：给出一个唯一的传输协议连接参考点

form-t-conn-ind：
form-t-conn-ctm：
form-t-data-ind：
form-t-disc-ind：
form-n-conn-req：
form-n-data-req：
form-n-conn-rsp：
form-n-disc-req：

每一个运算形成一组数据域，用于相应的服务原语，这要根据所提供的参数以及传输实体的背景知识（例如在n-conn-req情况下，传输和网络地址之间的映射）

conn-feasible：主叫TSAP处的传输实体知识指明连接可以实现（有本地资源可利用，被叫TASP可以通达，服务质量可达到）—布尔

conn-acceptable：在被叫NSAP处的传输实体具有资源来接受网络连接—布尔

conn-providable：被叫端的传输实体知识指明传输连接可以实现（有被叫TSAP可利用，服务质量可达到）—布尔

tpdu-type：确定n-data-ind的内容（值：CR、CC、DT、DR、ER或INV—最后一项指明某TPDU无效，不管是哪种原因）

prop-tpdu-size：对tpdu-size值，启动者的建议

agree-tpdu-size：对tpdu-size，应答者的同意值

extract-tpdu-size：确定在CC TPDU中包含的参数值

take-segment：从TSDU取下一个DT TPDU

ap end-segment：把当前的DT TPDU加到部分完成的TSDU中

last-tpdu：当前DT TPDU已置TSDU的结束标记运算的输入值列在运算名字之后的括号内。

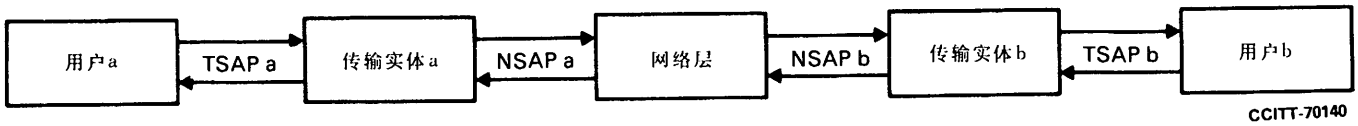


图 D—223

在一传输连接中，表示TRANSPORT-ENTITY的作用的概览图

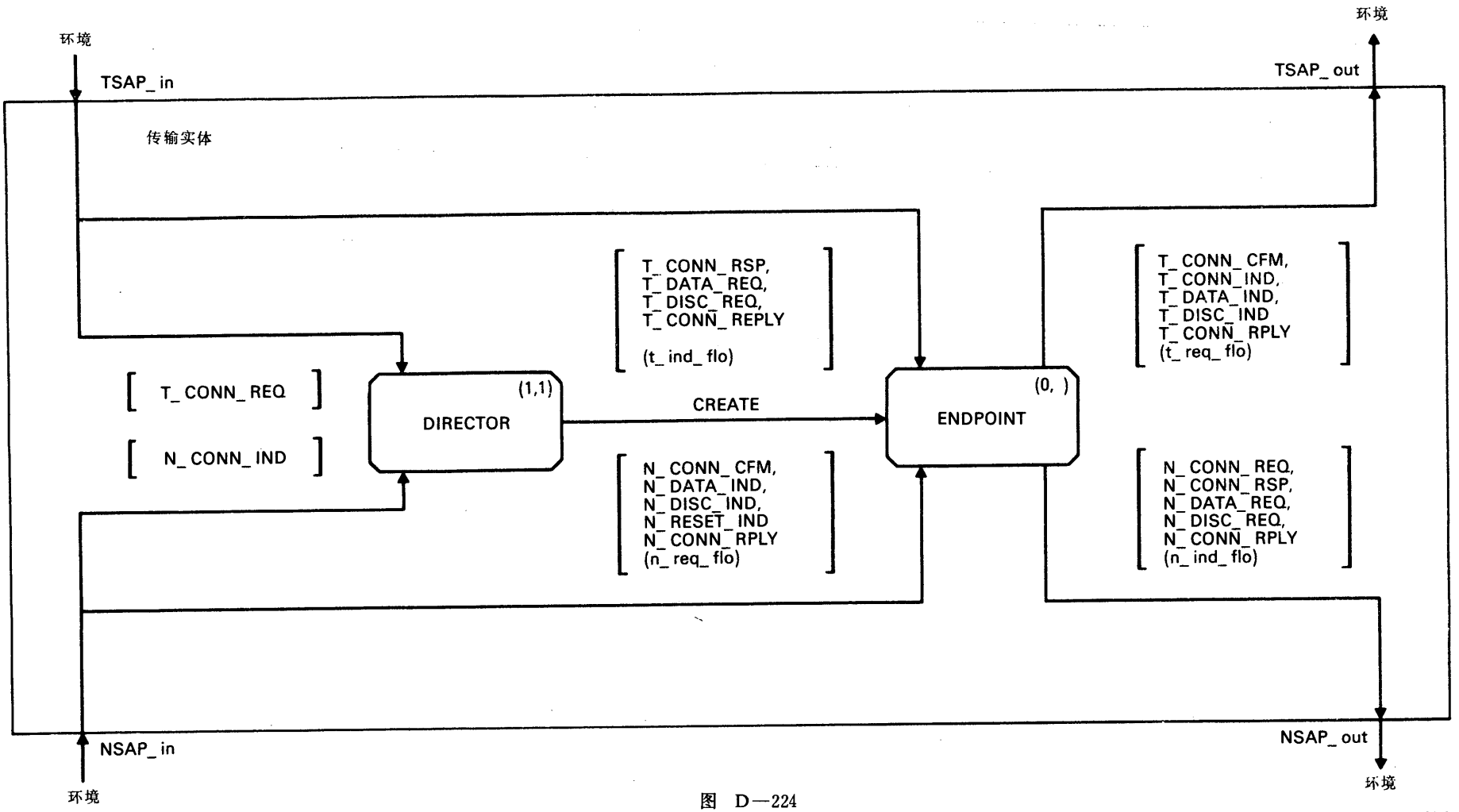


图 D—224

TRANSPORT－ENTITY块的交互作用图

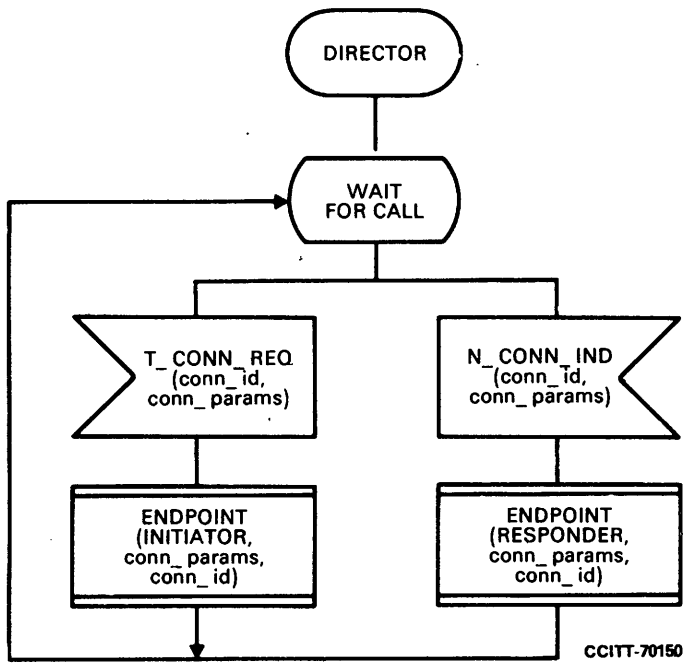


图 D—225

D I R E C T O R 进程图

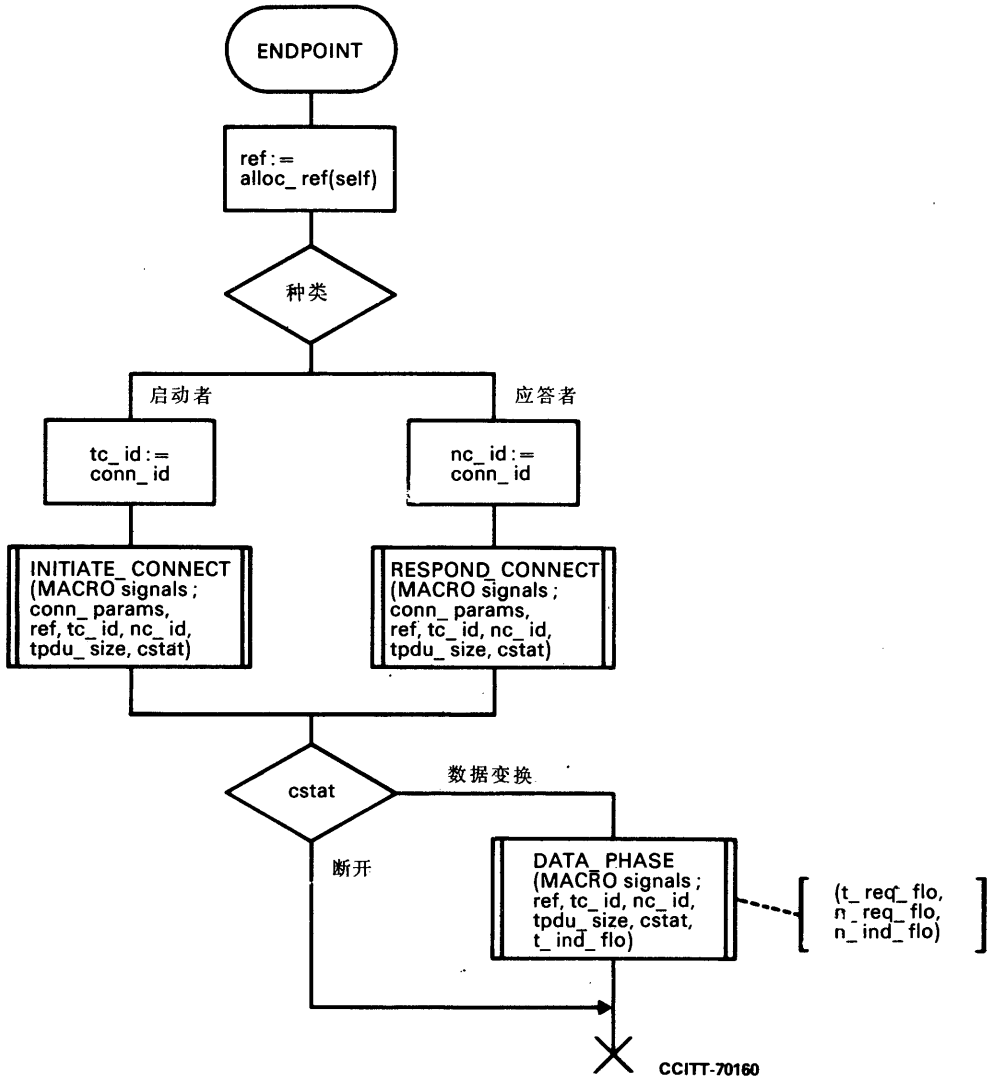
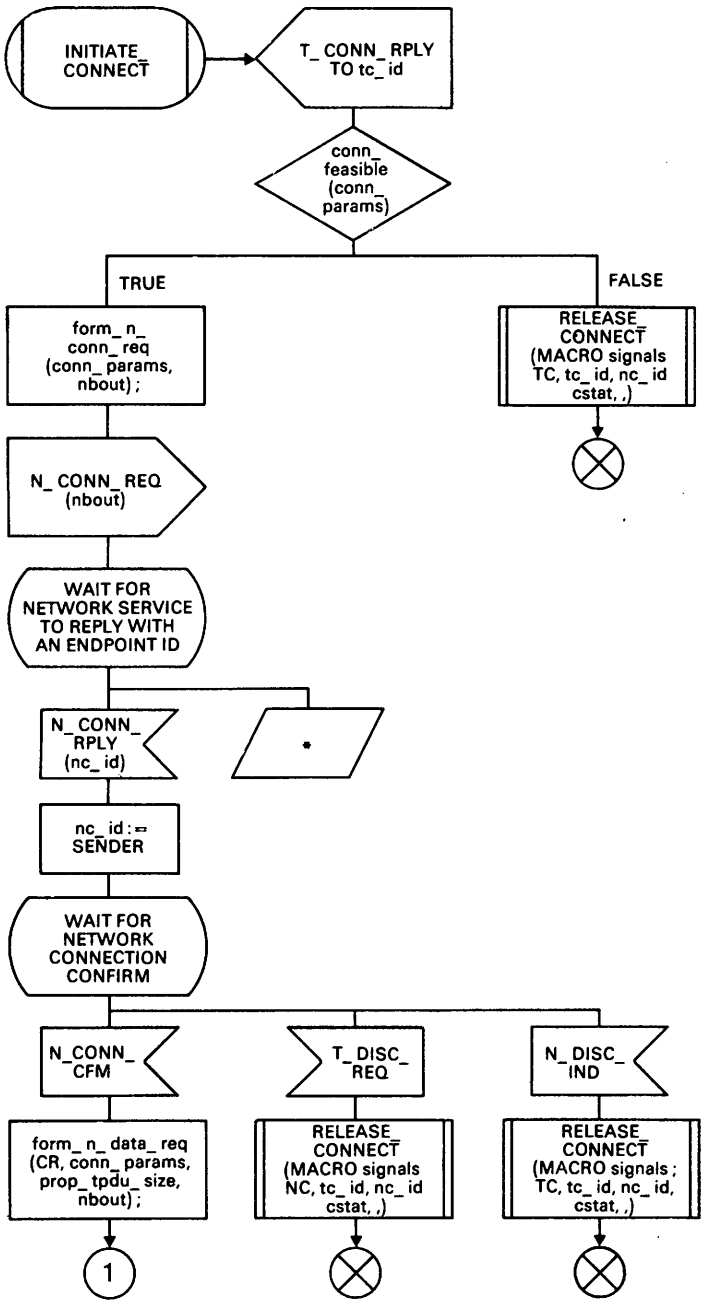


图 D—226

ENDPOINT进程图



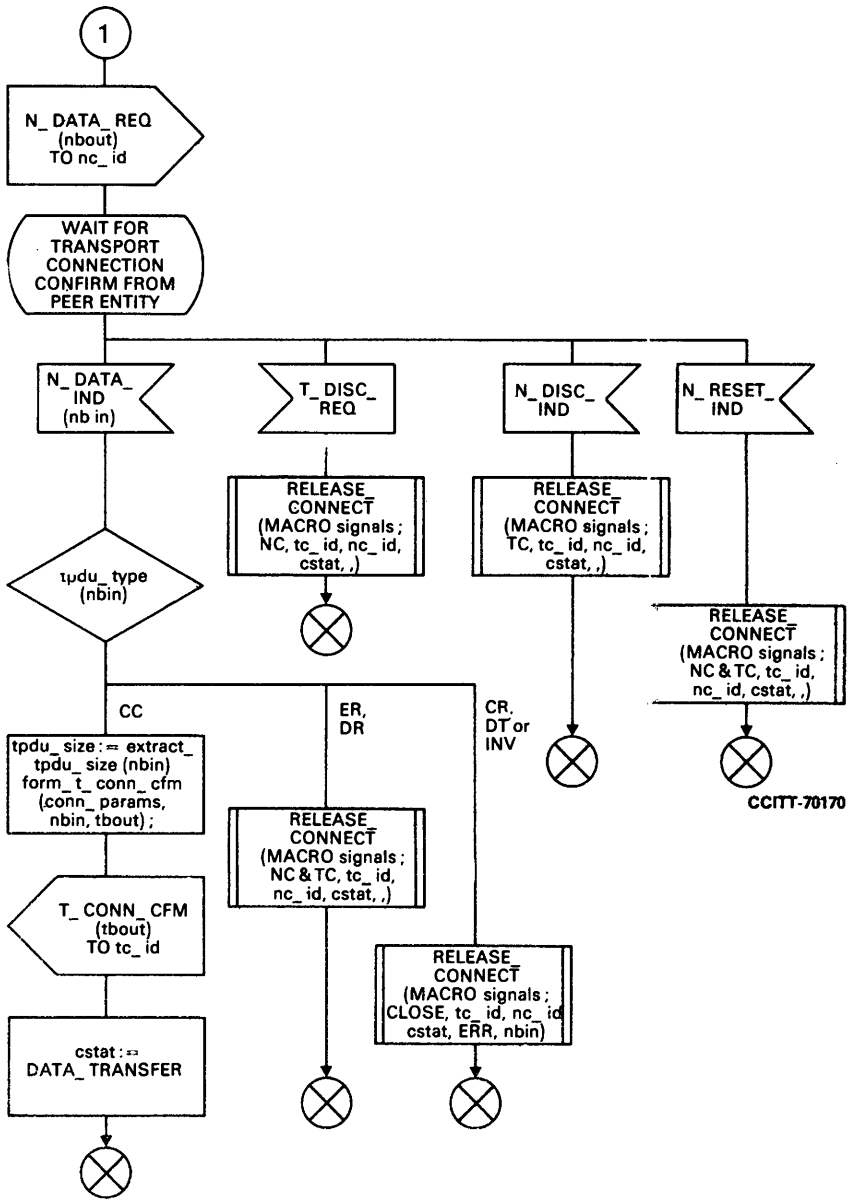


图 D—227

INITIATE-CONNECT过程图

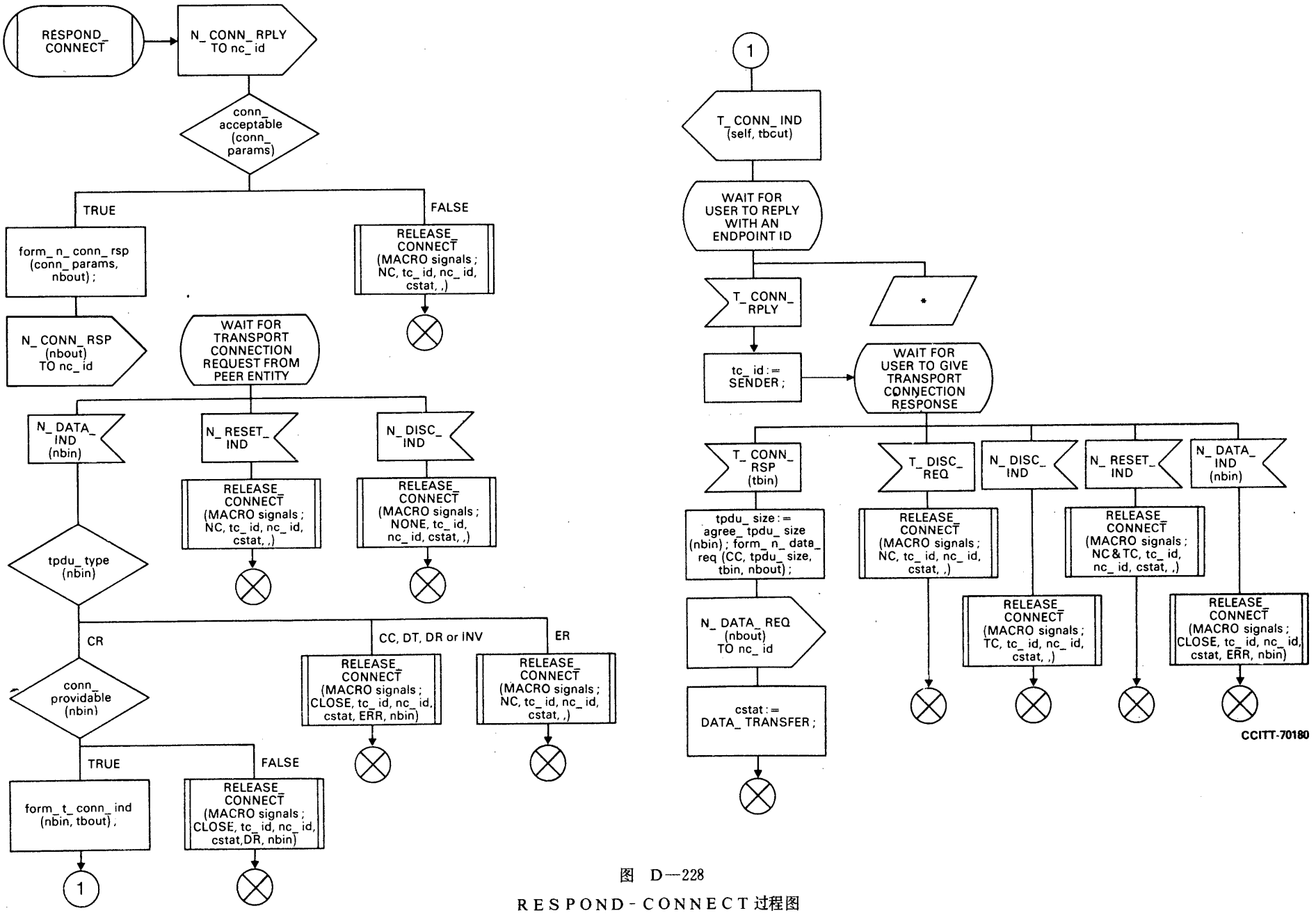


图 D—228

RESPOND - CONNECT 过程图

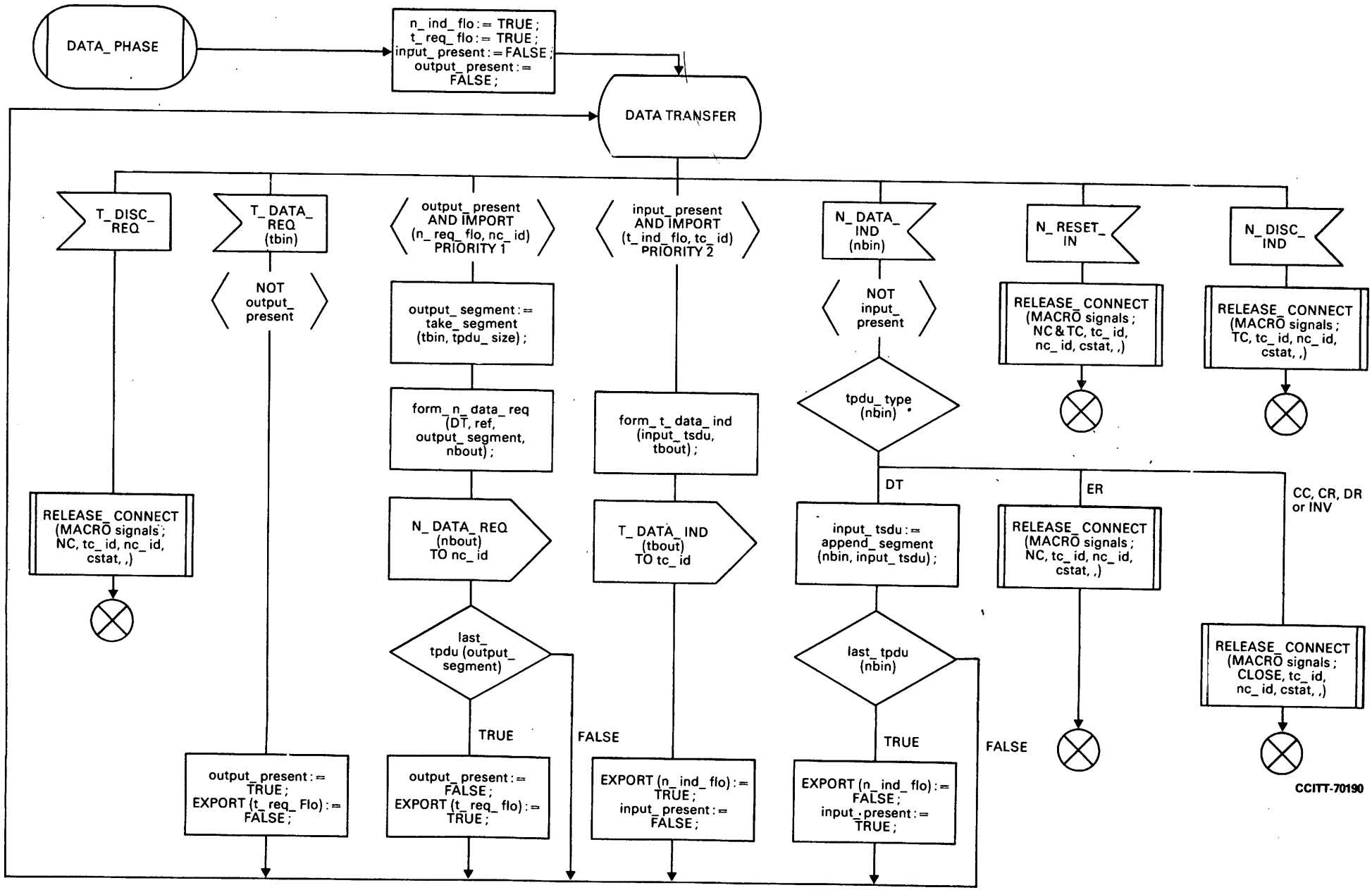


图 D—229

D A T A - P H A S E 过程图

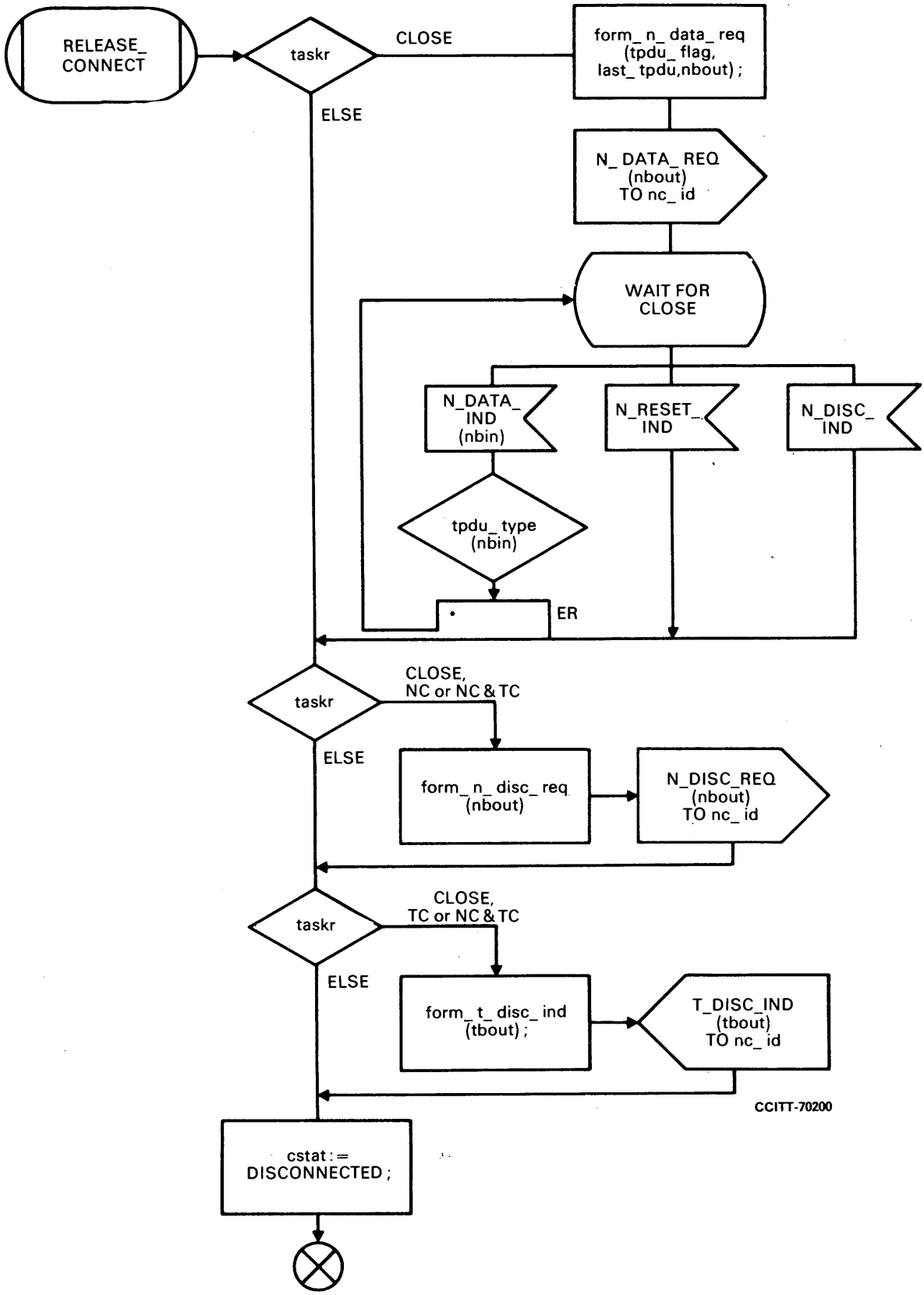


图 D—230

RELEASE-CONNECT 过程图

D.10 SDL工具

D.10.1 引 言

本章介绍一组可用来支持SDL的工具，这些工具可以支持生成各种文件、SDL图（GR形式）或清单（PR形式），和（或）验证SDL表示的正确性。

在本指南中没有把所有可能的工具列出一个完全的清单，所需的工具取决于用户选择的方法。

原则上，在没有任何工具的情况下也能使用SDL。然而，由于现代系统的内在复杂性，使得SDL表示往往很复杂，因此需要有一些自动工具来帮助对许多系统的规格、设计和文件进行管理。例如，一个交换局的图形文件编制用人工制图及人工更新是复杂和昂贵的，如使用适当的辅助手段将大大降低复杂性和成本。

由于以上的考虑，已将SDL设计成能够有效地使用一些工具来支持它的使用。

D.10.2 工具的分类

SDL工具可以按SDL文件产生期间所进行的工作来分类，例如：

* 输入工具

按照语法的形式，我们可以有适用于SDL流图、正文短语及图形等形式的输入辅助手段。

* 语法检查工具

包括三种语法中每一种语法的分析程序。

* 文件生成工具

SDL文件一旦被存贮在计算机内，这些工具就能对它们进行访问和复制，可能要用到若干外部设备。所使用的语法形式可以与输入文件所用的语法不同。此外有些工具也可以根据原先进入的文件类型推导出新的文件类型。

* 系统模拟和分析工具

表示一个系统的SDL文件可用来抽象出系统的模型，在该模型上可以进行检验，通过它们能够寻找死锁，能够在同一系统的各种模型之间（如规格和描述之间）作比较，或者进行系统行为的模拟等。

* 支持代码生成的工具

一些很详细的SDL表示可用来帮助软件编程。这些工具可以这样来开发，使之能实现半自动的代码序列生成。

一类特殊而有用的工具是：

* SDL训练工具

这些工具可以单独使用，或与其它工具结合在一起来使用，后者允许使用其它功能的用户在需要时能够得到训练工具的帮助。

由于SDL被用于一个系统生命期的许多不同阶段，在一个综合的工程环境中，人们会容易地遇到所有工具类型的用途。

D.10.3 文件输入

从输入的观点看，由于PR语法是与任何字符串输入等效的，所以输入SDL的正文短语形式（PR）不需要特别的要求。这样它就具有可以利用同一工具（行编辑程序）的优点。然而其它两种语法需有图形处理功能。

显然，尽管对PR输入的支持可能是有利的，但如果我们期望用GR和PE语法作为输入媒介，那么对GR和PE输入的支持就是重要的了。

GR/PE文件的输入可以放到一起来考虑，因为这两种语法都使用图形。所需要的是一图形编辑程序和一以图形形式输出图的设备。因为有可能在一预先规定的座标网格上输入一个图，故不要求有一图形输入设备。每个符号可以与一给定的字符串相关联。例如，有可能输入一个状态，办法是给出该状态在一座标网格上的位置（一对座标、方格数等）和用一字符串给出该状态的类型。

图形输入设备可以给用户以立即的反馈，然而，“网格方法”实现起来更快更容易。

往往要求一个图形编辑程序具有诸如连接两个符号，把一组符号移到该页的另一区域或移到其它的页上，以及连续删除（删除一个符号包含删除与该符号的连接）等功能。至于PR工具，GR/PE输入工具应当在SDL语义、语法上进行模拟。因此应当阻止无效的连接，并提醒用户去填入所有不完全的部分等等。

这些工具面临着一些问题，这些问题是由图形设备的物理限制例如“分辨能力”产生的。要有足够数量的能看清楚的字符，而同时又在屏幕上显示相当数量的符号，几乎是不可能的。

必须考虑诸如可变焦距的窗口或上滚下滚等解决办法，但它们不能完全令人满意。如果用户抄写图形，那么高分辨率就不是必须的，但若各图由用户直接产生，则高分辨率就是很需要的了。由于同样的原因（既要求概貌又要求一定数量的细节的图）在图的显示中希望要有高的分辨率。

辅助PR输入的工具是很有用的：它们可以将所期望的PR关键字提示给用户，指出缺少了某些结束的关键字（如ENDDECISION，ENDSTATE等）。

它们能按照接收到的关键字立即安排PR的格式，自动地插入定界符，并向用户提供面向PR的功能键等。

这类工具可以用现有的字符串编辑程序为基础来实现。这些编辑程序可以扩展，以包括上面提到的一些特性。

D.10.4 文件校验

文件一旦存入机器，下一步就是校验它们。首先应对它们分别进行校验，然后应把有关联的图组合在一起再校验，一直到整个系统被校验完为止。

如果是采用了面向SDL的工具进行输入的，则对每个单独文件的很多校验工作就随之完成了。

由“不合理的”操作（例如不是跟在状态后面的输入或保存）产生的错误，在输入阶段应全部被检测出来并进行校正。然而有些错误只有在一个文件的输入阶段完成时才能检测出来，当然在各文件相互矛盾的情况下，也只有在输入完成时才能查出。

若干SDL规则可以自动地受检验，例如要求所有的输出应当有一对应的输入。

在多层次表示的情况下，各层次之间的一致性在一定程度上可以受检验。

可以用形式的SDL模型推导出一批校验过程。

D.10.5 文件的复制

存贮在计算机内的SDL文件必须能被检索、显示和复制，需要有执行所有这些功能的工具。能够只检索文件的一部分或只检索多个文件的一些子集，会是很有用的。检索可以是面向SDL的，例如：“寻找所有发送一给定信号的进程”，或“在哪些状态”执行某一动作等等。在信息要使用图形语法显示时，显示信息的工具具有特殊的意义。有关文件输入的讲述在GR/PE语法中也同样适用。文件的复制取决于待复制文件的类型，取决于该文件如何存贮以及取决于外部输出设备的特性。可能也取决于如何输入这些文件。用户用一种语法输入文件却可能希望得到一个使用另一种语法的输出文件。

文件的复制受外部输出设备的限制，例如一个图可能太大，放不进一纸页给定的面积中，因而必须

把图分割成若干块。必须加上连接符，并应插入交叉引用。人们希望把工具所加的“附加”内容与原来输入的特性区别开来。另外一些物理限制会妨碍全部有用信息的输出，例如，给定的符号尺寸太小，包含不了全部有关的正文，这时可有若干策略供用户选择，包括展宽符号、削减正文、削减正文但加上完整的正文作为附注、把正文安排在符号旁边等等。人们希望输出格式灵活的各种工具，其功能包括符号的各种不同尺寸、各种输出格式、垂直显示或水平显示等。

一个文件总是应该能够复制得与它输入时完全相同。

D.10.6 文件生成

从用户输入的并存贮在计算机内的SDL文件出发，可以自动地生成若干其它文件，其中包括：

— 按每个进程，每个功能块或每个系统组织的信号表；
— 进程交互作用图，指明通信进程中各动作的相互作用及后继序列；
— 状态概览图，以一组用弧连接的状态来表示进程流图，弧代表跃迁；
— 按每个进程，每个功能块或每个系统组织的交叉引用表；
— 划分图，示出各功能块和层次的结构；
— 系统行为，作为对一系列环境动作的响应；
— 索引：文件一旦生成必将复制，以上提供的相同考虑也是适用的。

以GR形式输入的SDL文件可以自动地转换为等效的PR形式，反过来也是一样。

为了产生一正确的PR形式，表示一功能块各进程的所有GR图必须和有关的GR功能块交互作用图一起加以考虑。

下面几点应予以考虑：

— GR形式含有不能翻译成PR形式的直观信息（在PR中不存在），例如，符号的座标在PR形式中是无意义的。
— 连接不同页上的各流线的连接符可以删除。

然而，从PR到GR倒过来翻译则要复杂得多，并且不太可能满足所有可能的读者。除了行首退格之外，对PR形式的优雅没有主观的标准，而对GR形式却有着各种各样的主观标准。

由于GR形式的二维表示，可以删去为适应PR的顺序结构而插入的某些标号，因为用一连线就足够了。从一PR形式可以产生两种不同的GR表示，即功能块交互作用图和进程图（当然，如果PR是表示单独进程的话，就只能产生进程图）。

这种翻译通常产生GR图的一个模型，该模型包含一工具所必需的全部信息，用以在某一图形设备上编辑图形格式和复制该图。

请注意，两种不同的工具把某些PR翻译成GR时，可能得到布局不同的两种GR表示。这样得到的两种GR表示都是正确的，只要它们保持原表示中的语义。

D.10.7 系统模拟和分析

SDL文件不论它们是规定一个系统还是描述一个系统，基本上都是该系统的一个模型。

该模型主要用来把信息从一个人传递给另一个人，但也能用工具进行解释，检验其一致性、完备性（在仅用来规定一个系统的某些部分的情况下，不能满足完备性）和正确性、以及是否与SDL规则相符合（如文件检验一节所述）。

此外，可以研制一些工具，以便用模型来模拟各种系统功能的行为。模拟器可以与环境相互作用，并能依据模型得出用户期望的结论。

如果加上辅助信息，以指明在执行每个动作时所耗费的时间，以及计量可利用的资源（队列、实例等），则这种模拟也能研究系统的能力。

可以研制一些工具，从系统模型出发，来建立一个环境模型，并建立有意义的顺序，来检验实际系统。路径分析可以检测模型中的死锁。

系统模型也能用于联机的文件编制。如果实际系统和文件存储器之间存在适当的连接，那么就可能研制一种工具，在模型上跟踪系统的实时事件。

为了实现这一点，应当提供从系统方面看到的物理事件与SDL文件编制所涉及的逻辑事件之间的对应关系。如果把文件组织成若干抽象层次，则用户可以选择要跟踪的层次。这一点是很有用的，因为这能使具有不同教育和训练的用户能观察到系统的活动。

解释SDL模型的工具也可以用来挑选出同一系统的不同模型间的行为差异，它还可以用来比较各种不同的系统描述（由不同公司生产的系统），或者将系统规格与系统描述相比较，这样就有可能了解一个系统描述是否符合最初的规格。

D.10.8 代码生成

规定一形式地定义的语法和一SDL的形式数学定义，有可能实现一些工具，用来把SDL表示的语义映射到编程语言的语义，这类工具也许不能提供完整的实现程序，但至少在为实际程序提供一个框架方面，它们会是很有用的。

本用户指南的D．8.1节举了一个例子，概略地介绍了如何实现SDL和CHILL之间的映射。

D.10.9 训练

一套完整的关于SDL的训练教程已经编好，它由约二百页正文和一批幻灯片（约200张）组成。该教程包括了此语言的各个方面，同时还提供了一些例子以及关于使用SDL的几点建议。

SDL教程可以从国际电信联盟总秘书处—销售科得到。

通信地址：International Telecommunication Union, General Secretariat Sales Section, Place des Nations, CH－1211 Geneve 20 (SwitzerLand)。

中国印刷- ISBN 92—61—02245—6
ISBN7115—03472—9/Z