国 际 电 信 联 盟

CCITT

国 际 电 报 电 话 咨 询 委 员 会

蓝 皮 书

卷 X.1

功能规格和描述语言(SDL)使用形式描述方法(FDT)的标准

建议 Z.100和附件A、B、C和E，
建议 Z.110

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

国 际 电 信 联 盟

CCITT

国 际 电 报 电 话 咨 询 委 员 会

蓝 皮 书

卷 X.1

功能规格和描述语言(SDL)
使用形式描述方法(FDT)的标准

建议 Z.100和附件A、B、C和E，
建议 Z.110

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

ISBN 92-61-03755-0

中国印刷

CCITT 图书目录
第九次全体会议(1988 年)

蓝 皮 书

卷 I

卷 I.1 — 全会会议记录和报告
研究组及研究课题一览表

卷 I.2 — 意见和决议
关于 CCITT 的组织和工作程序的建议(A 系列)

卷 I.3 — 术语和定义 缩略语和首字母缩写词 关于措词含义的建议(B 系列)和综合电信统计的建议(C 系列)

卷 I.4 — 蓝皮书索引

卷 II

卷 II.1 — 一般资费原则 — 国际电信业务的资费和帐务 D 系列建议(第 III 研究组)

卷 II.2 — 电话网和 ISDN — 运营、编号、选路和移动业务 建议 E.100-E.333(第 II 研究组)

卷 II.3 — 电话网和 ISDN — 服务质量、网络管理和话务工程 建议 E.401-E.880(第 II 研究组)

卷 II.4 — 电报业务和移动业务 — 运营和服务质量 建议 F.1-F.140(第 I 研究组)

卷 II.5 — 远程信息处理业务、数据传输业务和会议电信业务 — 运营和服务质量 建议F.160-F.353、F.600、F.601、F.710-F.730(第 I 研究组)

卷 II.6 — 报文处理和查号业务 — 运营和服务的限定 建议 F.400-F.422、F.500(第 I 研究组)

卷 III

卷 III.1 — 国际电话接续和电路的一般特性 建议 G.100-G.181(第 XII 和 XV 研究组)

卷Ⅲ.2 — 国际模拟载波系统　建议 G.211-G.544(第 XV 研究组)

卷Ⅲ.3 — 传输媒质 — 特性　建议 G.601-G.654(第 XV 研究组)

卷Ⅲ.4 — 数字传输系统的概况;终端设备　建议 G.700-G.795(第 XV 和第 XⅧ研究组)

卷Ⅲ.5 — 数字网、数字段和数字线路系统　建议 G.801-G.961(第 XV 和第 XⅧ研究组)

卷Ⅲ.6 — 非话信号的线路传输　声音节目和电视信号的传输　H 和 J 系列建议(第 XV 研究组)

卷Ⅲ.7 — 综合业务数字网(ISDN) — 一般结构和服务能力　建议 I.110-I.257(第 XⅧ研究组)

卷Ⅲ.8 — 综合业务数字网(ISDN) — 全网概貌和功能、ISDN 用户-网络接口　建议 I.310-I.470(第 XⅧ研究组)

卷Ⅲ.9 — 综合业务数字网(ISDN) — 网间接口和维护原则　建议 I.500-I.605(第 XⅧ研究组)

卷Ⅳ

卷Ⅳ.1 — 一般维护原则:国际传输系统和电话电路的维护　建议 M.10-M.782(第Ⅳ研究组)

卷Ⅳ.2 — 国际电报、相片传真和租用电路的维护　国际公用电话网的维护　海事卫星和数据传输系统的维护　建议 M.800-M.1375(第Ⅳ研究组)

卷Ⅳ.3 — 国际声音节目和电视传输电路的维护　N 系列建议(第Ⅳ研究组)

卷Ⅳ.4 — 测量设备技术规程　O 系列建议(第Ⅳ研究组)

卷 V — 电话传输质量　P 系列建议(第 XⅡ 研究组)

卷Ⅵ

卷Ⅵ.1 — 电话交换和信令的一般建议　ISDN 中服务的功能和信息流　增补　建议 Q.1-Q.118(乙)(第Ⅺ研究组)

卷Ⅵ.2 — 四号和五号信令系统技术规程　建议 Q.120-Q.180(第Ⅺ研究组)

卷Ⅵ.3 — 六号信令系统技术规程　建议 Q.251-Q.300(第Ⅺ研究组)

卷Ⅵ.4 — R1和 R2信令系统技术规程　建议 Q.310-Q.490(第Ⅺ研究组)

卷Ⅵ.5 — 综合数字网和模拟—数字混合网中的数字本地、转接、组合交换机和国际交换机　增补　建议 Q.500-Q.554(第Ⅺ研究组)

卷Ⅵ.6 — 各信令系统之间的配合　建议 Q.601-Q.699(第Ⅺ研究组)

卷Ⅵ.7 — 七号信令系统技术规程　建议 Q.700-Q.716(第Ⅺ研究组)

卷Ⅵ.8 — 七号信令系统技术规程　建议 Q.721-Q.766(第Ⅺ研究组)

卷Ⅵ.9 — 七号信令系统技术规程　建议 Q.771-Q.795(第Ⅺ研究组)

卷Ⅵ.10 — 一号数字用户信令系统(DSS 1)　数据链路层　建议 Q.920-Q.921(第Ⅺ研究组)

卷Ⅵ.11 — 一号数字用户信令系统(DSS 1)　网络层、用户—网络管理　建议 Q.930-Q.940(第Ⅺ研究组)

卷Ⅵ.12 — 公用陆地移动网 与ISDN和PSTN的互通 建议Q.1000-Q.1032(第Ⅺ研究组)

卷Ⅵ.13 — 公用陆地移动网 移动应用部分和接口 建议Q.1051-Q.1063(第Ⅺ研究组)

卷Ⅵ.14 — 其它系统与卫星移动通信系统的互通 建议Q.1100-Q.1152(第Ⅺ研究组)

卷Ⅶ

卷Ⅶ.1 — 电报传输 R系列建议 电报业务终端设备 S系列建议 (第Ⅸ研究组)

卷Ⅶ.2 — 电报交换 U系列建议(第Ⅸ研究组)

卷Ⅶ.3 — 远程信息处理业务的终端设备和协议 建议T.0-T.63(第Ⅷ研究组)

卷Ⅶ.4 — 智能用户电报各建议中的一致性测试规程 建议T.64(第Ⅷ研究组)

卷Ⅶ.5 — 远程信息处理业务的终端设备和协议 建议T.65-T.101,T.150-T.390(第Ⅷ研究组)

卷Ⅶ.6 — 远程信息处理业务的终端设备和协议 建议T.400-T.418(第Ⅷ研究组)

卷Ⅶ.7 — 远程信息处理业务的终端设备和协议 建议T.431-T.564(第Ⅷ研究组)

卷Ⅷ

卷Ⅷ.1 — 电话网上的数据通信 V系列建议(第ⅩⅦ研究组)

卷Ⅷ.2 — 数据通信网:业务和设施,接口 建议X.1-X.32(第Ⅶ研究组)

卷Ⅷ.3 — 数据通信网:传输,信令和交换,网络概貌,维护和管理安排 建议X.40-X.181(第Ⅶ研究组)

卷Ⅷ.4 — 数据通信网:开放系统互连(OSI) — 模型和记法表示,服务限定 建议X.200-X.219(第Ⅶ研究组)

卷Ⅷ.5 — 数据通信网:开放系统互连(OSI) — 协议技术规程,一致性测试 建议X.220-X.290(第Ⅶ研究组)

卷Ⅷ.6 — 数据通信网:网间互通,移动数据传输系统,网际管理 建议X.300-X.370(第Ⅶ研究组)

卷Ⅷ.7 — 数据通信网:报文处理系统 建议X.400-X.420(第Ⅶ研究组)

卷Ⅷ.8 — 数据通信网:查号 建议X.500-X.521(第Ⅶ研究组)

卷Ⅸ — 干扰的防护 K系列建议(第Ⅴ研究组) 电缆及外线设备的其它部件的结构、安装和防护 L系列建议(第Ⅵ研究组)

卷Ⅹ

卷Ⅹ.1 — 功能规格和描述语言(SDL) 使用形式描述方法(FDT)的标准 建议Z.100和附件A、B、C和E,建议Z.110(第Ⅹ研究组)

卷Ⅹ.2 — 建议Z.100的附件D:SDL用户指南(第Ⅹ研究组)

卷Ⅹ.3 — 建议Z.100的附件F.1:SDL形式定义 介绍(第Ⅹ研究组)

卷X.4 — 建议Z.100的附件F.2:SDL形式定义 静态语义学(第X研究组)

卷X.5 — 建议Z.100的附件F.3:SDL形式定义 动态语义学(第X研究组)

卷X.6 — CCITT高级语言(CHILL) 建议Z.200(第X研究组)

卷X.7 — 人机语言(MML) 建议Z.301-Z.341(第X研究组)

蓝皮书卷 X.1 目录

建议 Z.100 和附件 A、B、C 和 E
建议 Z.110

功能规格和描述语言(SDL)
使用形式描述方法(FDT)的标准

卷 首 说 明

1 在 1989—1992 研究期内委托给各研究组的研究课题可查阅给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。

卷 X.1

建议 Z.100 和附件 A、B、C 和 E
建议 Z.110

功能规格和描述语言(SDL)
使用形式描述方法(FDT)的标准

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

Z.100 目录

建议 Z.100

功能规格和描述语言(SDL)

卷 首 说 明

本建议取代了 CCITT 红皮书的建议 Z.100 至 Z.104 和建议 X.250。

1 SDL 概述

1.1 引言

推荐 SDL(功能规格和描述语言)的目的,是为了提供一种能确切地定义电信系统的功能规格及对其性能进行描述的语言。用 SDL 制订规格和描述的目的是要做到形式化,也就是说,在对电信系统的功能规格及性能进行分析与解释时,可以做到其含意无二义。

规格和描述这两个术语具有下述意义:

a) 系统的规格是要求系统具有的行为的描述;

b) 系统的描述是对系统实际行为的描述。

注 — 由于把 SDL 用于制订规格和把 SDL 用于描述的两种用法没有什么区别,因此,在本文的后继部分中,术语"规格"既用于所要求的行为,也用于实际的行为。

广义地说,系统规格既包括系统的行为规格,也包括一组一般参数。然而,SDL 的目的仅在于定义系统的行为方面;对于容量、重量等特性的一般参数必须采用不同的方法来描述。

1.1.1 目标

当定义 SDL 时,总的目标是要提供这样一种语言,该语言:

a) 易于学习、使用和解释;

b) 为订购及投标提供明确的功能规格;

c) 可以扩展,以适用于新的发展;

d) 能够支持若干种系统功能规格和设计的方法,而不需设定其中任何一种。

1.1.2 应用

SDL 主要用来制订实时系统行为方面的规格。其应用包括:

a) 交换系统中的呼叫处理过程(例如呼叫处理、电话信号、计费);

b) 一般电信系统中的维护和故障处理(例如告警、自动故障排除、例行测试);

c) 系统控制(例如过载控制、更改或扩充过程);

d) 操作和维护功能,网络管理;

e) 数据通信协议。

当然,任何实物,只要其行为可以用离散模式来规定,也就是说,此实物可用离散信息与其环境进行通信,就可以用 SDL 来规定其行为的功能规格。

SDL 是一种丰富的语言,它不但可以用于高层的非形式(或形式化不完全)的规格描述而且还可用于细节的规格描述。使用者须根据所要描述的通信的层次和根据应用该语言的环境

来选择 SDL 的适当部分。根据采用某功能规格的环境，在功能规格的源与目标之间，有很多方面可能有待于共同的理解。

SDL 可以用来产生：

a) 设备的要求；

b) 系统规格；

c) CCITT 建议；

d) 系统设计规格；

e) 细节描述；

f) 系统设计(高层的和详细的)；

g) 系统测试。

并且，使用机构可以选择适当的 SDL 应用的抽象层次。

1.1.3 系统规格

假设系统的激励与响应均是离散的并携带信息，则可用 SDL 规格以激励/响应方式来定义系统的行为。在特殊情况下，系统规格可以看作是系统对任意给定激励序列的响应序列。

通信扩展有限态自动机的概念是系统规格模型的基础。

SDL 也提供了结构化概念，这减轻了制定庞大和(或)复杂系统的规格的困难。这些结构化构件允许把系统的功能规格分解成易于处理的单元。人们可以分别处理和理解这些单元。分解的过程可以由若干步来完成，这将导致由许多单元组成的分层结构，这些单元在不同层次定义了该系统。

1.2 SDL 文法

当描述一个系统的时候，SDL 给出了两种不同的语法形式供使用者选择：一种为图形表示法(SDL/GR)；另一种为正文短语表示法(SDL/PR)。两者均为同一 SDL 语义的具体表示，所以，它们是等价的。特别是对于对应的概念，它们都等同于同一个抽象文法。

SDL/GR 与 SDL/PR 共有的子集，称为公共正文文法。

图 1.1 给出了 SDL/PR、SDL/GR、具体文法及抽象文法之间的关系。

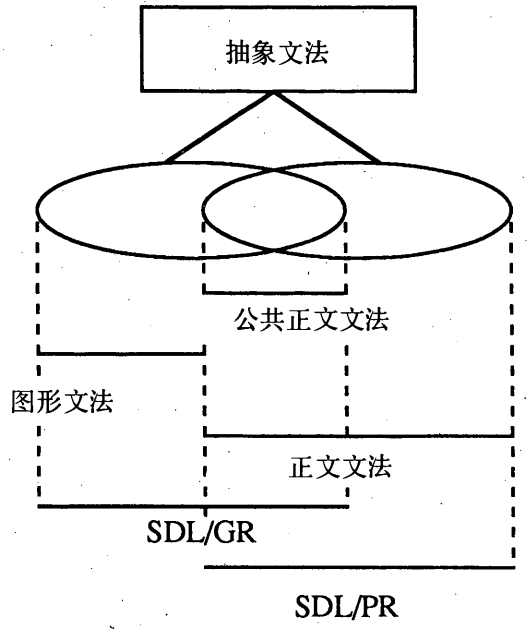


图 1.1

SDL 文法

每一个具体文法都有其自己的语法及与抽象文法的关系(例如如何变换成为抽象语法)。采用这一方法，则 SDL 的语义仅有一种定义，每一具体文法将通过其与抽象语法的关系来继承语义。这一方法也保证了 SDL/PR 与 SDL/GR 的等效性。

建议中还给出了 SDL 的形式定义，它规定了如何将一个系统规格转换成抽象语法，并规定了如何解释由抽象语法给出的某种规格。

1.3 基本定义

本建议中使用了一些一般的概念和约定，现给出它们的定义如下：

1.3.1 类型、定义和实例

在本建议中，类型、类型实例的概念及它们的相互关系是基本。所采用的方案和术语定义如下，并示于图 1.2 中。

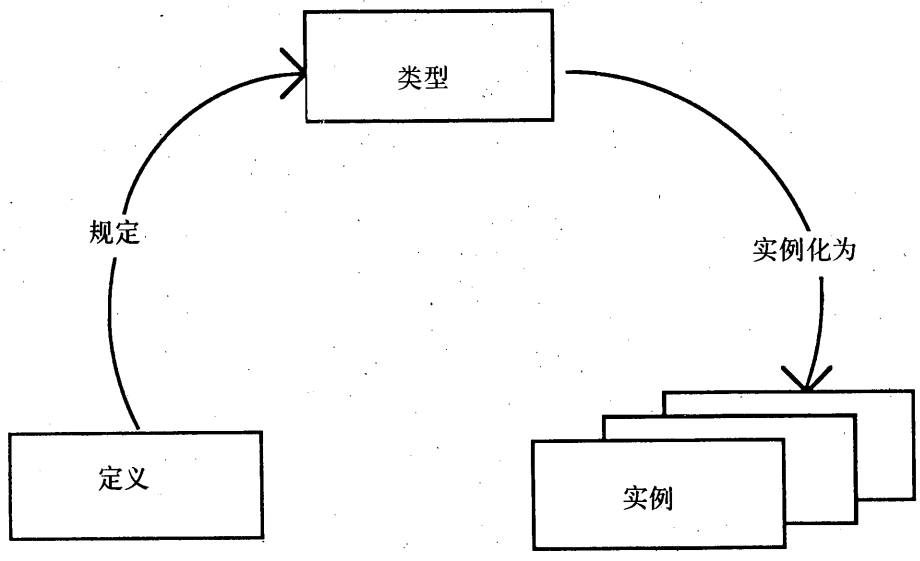


图 1.2

类 型 概 念

类型由定义来规定。一个类型的定义规定了与该类型有关的所有特性。一个类型可以有任意多个实例，一特定类型的任一实例具有定义给该类型的所有特性。

这一方案适用于若干 SDL 概念，例如，系统定义和系统实例，进程定义和进程实例。

数据类型是一类特殊的类型(参见 § 2.3 和 § 5)。

注 — 为避免繁琐，特约定：术语“实例”可以省去。例如，“解释一个系统……”的意思是“解释一个系统实例……”。

1.3.2 环境

用 SDL 规定的系统根据从外界接收到的激励来行动，这个外界称为被规定系统的环境。

假设在环境中有一个或多个进程实例，因而从环境流向系统的信号具有与这些进程实例相关的标识，这些进程所具有的 PId 值不同于系统中的任何 PId 值(参见 § 5.6.10)。

尽管环境的行为是不确定的，但它必须遵循由系统规格所给定的限制。

1.3.3 错误

仅当一个系统规格满足 SDL 的语法规则和静态条件时，该系统规格才是合法的 SDL 系统规格。

如果一个合法的 SDL 规格被解释的时候违反了动态条件，则会出现错误。如果对系统规格的解释导致错误就意味着不可能从该规格推导出系统今后的行为。

1.4 呈现格式

1.4.1 正文划分

本建议的 § 2、3、4、5 节分为几个题目，对每个题目的描述先是引言(也可以没有引言)，随后是关于以下几方面的带有标题的详叙项：

a) *抽象文法*——由抽象语法及良形式的静态条件来描述。

b) *具体正文文法*——这包括用于 SDL/PR 及 SDL/GR 的公共正文文法和仅用于 SDL/PR 的文法两部分。此文法由正文语法、静态条件和用于正文语法的良形式规则、以及正文语法与抽象语法之间的关系来描述。

c) *具体图形文法*——它由图形语法、静态条件、用于图形语法的良形式规则、此语法与抽象语法间的关系、以及其它一些绘图规则(参见 § 2.2.4 中相应部分)来描述。

d) *语义*——给出一个类型的定义，提供该类型所具有的特性、对该类型的一个实例进行解释的方法、以及为了在 SDL 意义上该类型实例行为良好所必须满足的任何动态条件。

e) *模型*——给出用预先定义的严格具体语法构件表达出的简化符号的映象。

f) *举例*。

1.4.2 带有标题的详叙项

当一个题目具有引言部分并随之有一带有标题的详叙项时，则认为引言部分是建议的非正式部分，该部分仅用于帮助理解，而不是使建议完整。

若带有标题的详叙项没有正文，则该项完全被略去。

本节的剩下部分描述用于每一带标题详叙项及所用标题的其它特殊形式。也可以认为这一部分为上面定义的初级带标题详叙项的一个印刷版面的例子。这里的正文可看成是引言一节的一部分。

抽象文法

在 § 1.5.1 中给出了抽象语法符号的定义。

如果省略了“*抽象文法*”的标题详叙项，则对于所介绍的题目来说，没有另外的抽象语法，

并且，具体语法将由另一个序号正文节所定义的抽象语法来描述。

可以从各标题详叙项引用抽象语法中的规则，办法是用楷体（英文用斜体）来书写规则的名字。

形式符号的规则后或许有数段文字，这些文字定义的条件是一个良形式的 SDL 定义所必须满足的，并且这些条件可以在不须对实例进行解释的情况下进行校验。这里静态条件只与抽象语法有关。仅与具体语法有关的静态条件将在具体语法之后定义。抽象语法的静态条件与抽象语法一道定义了此语言的抽象文法。

具体正文文法

具体正文语法用扩展的巴科斯-瑙尔范式的语法来说明，它在建议 Z.200 的 2.1 节中定义（还可参看 § 1.5.2）。

正文语法后跟有数段文字，这些文字规定了一些静态条件，一个良形式的正文必须满足这些条件。并且，这些条件可以在不用对实例进行解释的情况下进行校验。对于抽象文法，静态条件（如果有的话）也是适用的。

在很多情况下，具体语法与抽象语法之间有一种简单的关系，这是由于具体语法的一条规则可以简单地由抽象语法中的一条规则来表示。当在抽象语法与具体语法中使用同样的名字以表明它们代表同样的概念时，那么，在语言描述中就意味着含有这样的句子："具体语法中的〈x〉代表抽象语法中的 X"，而该句子通常是被省略掉的。在这个意义上，除加下划线的语义子范畴有效外，其余情况可以忽略。

非简化形式的具体正文语法（用其它 SDL 构件构成的导出语法）是严格的具体正文语法。仅对于严格的具体正文语法规定了具体正文语法和抽象语法的关系。

如果所定义的题目是由其它 SDL 构件（见下面的*模型*一段）构成的简化形式，则具体正文语法与抽象语法的关系便省略掉。

具体图形文法

具体图形语法用扩展的巴科斯-瑙尔范式的语法来说明。后者的定义在 § 1.5.3 中给出。

具体图形语法后跟有数段文字，这些文字规定了一些静态条件。良形式的 SDL/GR 必须满足这些条件，并且，可以在不用对实例进行解释的情况下校验这些条件。对于抽象文法，静态条件（如果有的话）也是适用的。

如果所定义的项目是用其它 SDL 构件（见下面*模型*一段）构成的简化形式，则具体图形语法与抽象语法的关系便省略掉。

在很多情况下，具体图形文法图与抽象语法定义之间有一种简单的关系。当具体文法中的非终结名字用词"图"来结尾而在抽象文法中有一名字除了用词"定义"结尾外其它部分均相同，则这两个名字所确定的规则表示同一个概念。例如，具体文法中的〈系统图〉与抽象文法中的"*系统定义*"相当。

具体语法方面的扩展起源于这样一些机制，如间接定义（§ 2.4.1）、宏指令（§ 4.2）、字面值映象（§ 5.4.1.15）等等，这种扩展必须在具体语法与抽象语法对应之前考虑。

语义

在良形式规则中，使用包括类型或与那个类型有关的其它类型的特性。

例如，一个进程合法输入信号标识符的集合是一种特性。该特性用在下述的静态条件之中："对每一状态节点，所有输入信号标识符(在合法输入信号集内)或出现在一保存信号集中、或出现在一输入节点中"。

除特殊方法形成的外，所有实例都具有标识特性，此标识特性的确定如同§2节中关于标识的一般规则一节所定义的那样。因此，作为一种标识特性，通常是不提及的。从抽象语法的角度来看，也不必叙述包含在定义中的子定义部分，这是因为这种子定义部分为谁所拥有是显而易见的。例如，功能块定义显然"已经"包含了进程定义和(或)功能块子结构定义。

如果特性可以在不须解释 SDL 系统规格的情况下确定，则特性为静止的；如果为了确定某些特性需要解释 SDL 系统规格，则这些特性就是动态的。

解释可以用运行的方式来描述。每当用抽象语法给出一个列表，则对此列表的解释要按给定的顺序进行。这就是说，本建议描述了怎样从系统定义创建实例、以及在"抽象的 SDL 自动机"中怎样解释这些实例。

动态条件是这样一种条件，它在解释期间必须得到满足，并且，不进行解释不可能进行校验。动态条件可能导致错误(参见§1.3.3)。

模型

某些构件被认为是代表其它等效具体语法构件的"导出具体语法"(或简化)。例如，一个信号的一个输入后面如果跟随着回到原状态的空跃迁则可以忽略此信号的输入。这就是导出具体语法。

若经扩展，这样的"导出具体语法"有时会呈现极大的(可能是无限的)代表性，尽管如此，这样一种规格的语义是可以确定的。

举例

标题详叙项"举例"包含了一些例子。

1.5 元语言

考虑到特性的定义和 SDL 的语法，可根据特别的需要使用不同的元语言。

下面将介绍所用的元语言，在适当的地方有时仅指出参考书或 ITU 专刊。

1.5.1 *Meta* Ⅳ

下面的 Meta Ⅳ 子集用来描述 SDL 的抽象语法。

抽象语法给出的一个定义可以看作是一个命名复合实物(一棵树)，用来定义一组子分量。

例如，关于变量定义的抽象语法是

Variable-definition :: *Variable-name Sort-reference-identifier*
(即 变量定义 :: 变量名 类别引用标识符)

它为名为变量定义的复合实物(树)确定了域。此实物由两个子分量构成，这两个子分量又可以是树。

Meta Ⅳ 定义

Sort-reference-identifier = *Identifier*
(即 类别引用标识符 = 标识符)

表示一个类别引用标识符是一个标识符，因此，在语法上它与其他标识符不加区分。

一个实物也可能是某种基本的(非复合的)域。在 SDL 范围内，这些是：

a) 整数实物

例如

Number-of-instances :: *Intg Intg*
(即 实例数 :: 整数 整数)

实例数表征一复合域，此域包含两个整数值，用来表示实例的初始数与最大数。

b) 引用文实物

黑体大写字母和数字的任意序列表示引用文。
例如

Destination-process = *Process-identifier* | **ENVIRONMENT**
(即 目的地进程 = 进程标识符 | **ENVIRONMENT**)

目的地进程或是一个进程标识符，或是由引用文 **ENVIRONMENT** 表示的环境。

c) 记号实物

Token 表示记号的域。可以认为此域包含一个潜在的无穷集合，由各不相同的基本实物组成。这些基本实物不需要用别的东西来表示。
例如

Name :: *Token*
(即 名字 :: 记号)

一个名字由一个基本实物记号构成以便使任一名字能够与任何别的名字区分开。

d） 未定实物

未定实物指的是可能有某种表示的域，但是在本建议中，此表示是没有意义的。

例如

Informal-text :: …

（即 非形式正文 :: …）

非形式正文包括一个没有解释的实物。

下面的 *BNF*（参见 § 1.5.2）运算符（构件符）也用于抽象语法中：“ * ”表示该表可能是空表，“+”表示该表非空，“|”表示或，“ [”“]”表示任选。

圆括号用来表示在逻辑上相关的成组的域。

最后要说的是：抽象语法采用另一个后缀运算符 **“set”** 来形成一个集合（各不相同实物的无序集合）。例如

Process-graph :: *Process-state-node State-node*-**set**

（即 进程图 :: 进程起动节点 状态节点 **set**）

一个进程起动节点和状态节点的一个集合构成了一个进程图。

1.5.2 *BNF*

在 BNF（巴科斯-瑙尔范式）中，终结符号或是通过不括在尖括号（即小于号和大于号亦即〈和〉）中指明，或是〈名字〉和〈字符串〉这两种表示之一。要注意的是：〈名字〉和〈字符串〉这两种特殊的终结符可能具有如下定义所强调的语义。

尖括号及它所括起来的词或者是非终结符，或者是〈名字〉或〈字符串〉两种终结符之一。由括在尖括号中的一个或多个词表示的非终结符属语法范畴。对每一个非终结符都给出了一种产生式规则，或者是用具体正文文法或者用图形文法给出。

例如：

〈视见表达式〉::=

VIEW（〈变量标识符〉,〈表达式〉）

非终结符的产生式规则包括在符号::=左边的非终结符以及在右边的一个或多个构件，这些构件由非终结符和（或）终结符组成。例如，上面例子中的〈视见表达式〉,〈变量标识符〉和〈表达式〉都是非终结符；而 VIEW、圆括号及逗号都为终结符。

有的符号含有带下划线的部分，此下划线用于强调那个符号的语义。例如，〈变量标识符〉在语法上是与〈标识符〉相同的，但在语义上，它要求此标识符为变量标识符。

在符号::=的右边，可以给出几种供选择的终结符产生式，它们由一竖杠（|）分开。

例如：

〈功能块区〉::=

〈图形功能块引用〉

|〈功能块图〉

表示一个〈功能块区〉或者是一个〈图形功能块引用〉，或者是一个〈功能块图〉。

应用波形括号（{和}）可以把语法元素组合起来，这里的波形括号类似于Meta Ⅳ（参见§1.5.1）中的圆括号。一对波形括号中可以含有一个或多个竖杠用来表示可选语法元素。

例如：

〈功能块相互作用区〉::=

{〈功能块区〉|〈信道定义区〉}+

波形括号组合的重复由星号（*）或加号（+）表明。星号表示此波形括号组合是可以不出现也可以出现的，并可以进一步重复任意次数；加号表示该波形括号组合必须出现，并可进一步重复任意次数。上面的例子表示〈功能块相互作用区〉至少含有一个〈功能块区〉或一个〈信道定义区〉，也可以含有多个〈功能块区〉和〈信道定义区〉。

用方括号（[和]）括起来的语法元素是任选的，

例如：

〈进程标题〉::=

PROCESS〈进程标识符〉[〈形式参数〉]

表示〈进程标题〉可以含有、但不一定含有〈形式参数〉。

1.5.3 图形文法元语言

针对图形文法，我们将§1.5.2节中描述的元语言进行扩展，增加了如下的元符号：

a) **contains**

b) **is associated with**

c) **is followed by**

d) **is connected to**

e) **set**

元符号**set**是一个后缀运算符，它作用于紧靠其前的在波形括号中的语法元素，用来表示一些项的（无序）集合。每一项可以是任一组语法元素，在此情况下，它必须在应用**set**元符号之前进行扩展。

例如：

{{〈系统正文区〉}*{〈宏指令图〉}*〈功能块相互作用区〉}**set**

是这样一种集合，它有零个或多个〈系统正文区〉、零个或多个〈宏指令图〉、以及一个〈功能块相互作用区〉。

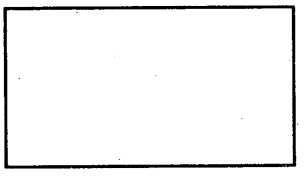
所有其它元符号均为中缀运算符，它们有一个图形非终结符作为左边的变元。右边变元或为位于波形括号中的一组语法元素或为单一语法元素。如果一个产生式规则的右边有一图形非终结符作为第一个元素、并含有这些中缀运算符中的一个或多个，则图形非终结符是每一个这种中缀运算符的左边变元。一个图形非终结符是一个非终结符，它带有一个字“symbol”（“符”）紧靠在大于号＞之前。

元符号 **contains** 表示其右边的变元应当放在其左边变元之中和可能有的附加〈正文扩展符〉之中。

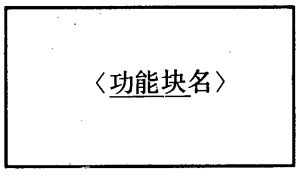
例如：

〈图形功能块引用〉::=
〈功能块符号〉 **contains** 〈功能块名〉

〈功能块符号〉::=



其含义如下



元符号 **is associated with** 表示其右边变元在逻辑上是与其左边变元相关联的（仿佛它“包含”在那个变元之中，其明确的联系由适当的绘图规则来保证）。

元符号 **is followed by** 的意思是：其右边变元（在逻辑上和画法上）跟随在其左边变元的后面。

元符号 **is connected to** 的意思是：其右边变元（在逻辑上和画法上）连接到其左边的变元。

2 基本 SDL

2.1 引言

一个 SDL 系统有一组功能块。功能块通过信道相互连接和与其环境连接。在每一个功能块中，有一个或多个进程，这些进程通过信号互相通信并设想为并行执行。

§2 节分成八个主题目：

a) 一般规则

基本的 SDL 概念，例如词法规则和标识符、视见度规则、非形式正文、图的划分、绘图规则、注释、正文扩展、正文符号。

b) 基本数据概念

基本的 SDL 数据概念，例如值、变量、表达式。

c) 系统结构

包括 SDL 概念涉及此语言的一般性结构概念，这些概念是系统、功能块、进程、过程。

d) 通信

包括 SDL 中所采用的通信机制，例如信道、信号路由、信号。

e) 行为

与进程的行为相关的构件是：进程或过程图形的一般连接规则、变量定义、启动、状态输入、保存、标号、跃迁。

f) 动作

动作构件有：任务、进程创建、过程调用、输出、判定。

g) 定时器

定时器定义和定时器原语。

h) 举例

从其它题目引用的例子。

2.2 一般规则

2.2.1 词法规则

词法规则定义词法单元。词法单元是具体正文语法的终结符。

〈词法单元〉 ::=
〈名字〉
| 〈字符串〉
| 〈专用词〉
| 〈复合专用词〉
| 〈注〉
| 〈关键字〉

〈名字〉 ::=
〈字〉{〈下划线〉〈字〉}*

〈字〉::=
{〈字母数字〉|〈句号〉}*
〈字母数字〉
{〈字母数字〉|〈句号〉}*

〈字母数字〉 ::=
〈字母〉
| 〈十进制数字〉
| 〈国标字符〉

〈字母〉 ::=
A|B|C|D|E|F|G|H|I|J|K|L|M
| N|O|P|Q|R|S|T|U|V|W|X|Y|Z
| a|b|c|d|e|f|g|h|i|j|k|l|m
| n|o|p|q|r|s|t|u|v|w|x|y|z

〈十进制数字〉 ::=
0|1|2|3|4|5|6|7|8|9

〈国标字符〉 ::=
#
| `
| ¤
| @
|〈左方括号〉
| \
|〈右方括号〉
|〈左波形括号〉
|〈竖杠〉
|〈右波形括号〉
|〈跨线〉
|〈向上箭头〉

〈左方括号〉 ::=
[

〈右方括号〉::=
]
〈左波形括号〉::=
{
〈竖杠〉::=
|
〈右波形括号〉::=
}
〈跨线〉::=
~
〈向上箭头〉::=
^
〈句号〉::=
.
〈下划线〉::=
_
〈字符串〉::=
〈撇号〉{〈字母数字〉
| 〈其它字符〉
| 〈专用字符〉
| 〈句号〉
| 〈下划线〉
| 〈空格〉
| 〈撇号〉〈撇号〉}*〈撇号〉
〈正文〉::=
{〈字母数字〉
| 〈其它字符〉
| 〈专用字符〉
| 〈句号〉
| 〈下划线〉
| 〈空格〉
| 〈撇号〉}*
〈撇号〉::=
'
〈其它字符〉::=
? |&|%
〈专用字符〉::=
+|-|! |/|>|*| (|) |" |, |; |<|=|:

〈复合专用符号〉::=
==
| ==>
| /=
| <=
| >=
| //
| :=
| =〉
| —〉
| (.
| .)
〈注〉::=
/* 〈正文〉 */
〈关键字〉::=
ACTIVE
| ADDING
| ALL
| ALTERNATIVE
| AND
| AXIOMS
| BLOCK
| CALL
| CHANNEL
| COMMENT
| CONNECT
| CONSTANT
| CONSTANTS
| CREATE
| DCL
| DECISION
| DEFAULT
| ELSE
| ENDALTERNATIVE
| ENDBLOCK
| ENDCHANNEL
| ENDDECISION
| ENDGENERATOR
| ENDMACRO
| ENDNEWTYPE
| ENDPROCEDURE
| ENDPROCESS
| ENDREFINEMENT
| ENDSELECT
| ENDSERVICE
| ENDSTATE
| ENDSUBSTRUCTURE
| ENDSYNTYPE
| ENDSYSTEM
| ENV

| ERROR
| EXPORT
| EXPORTED
| EXTERNAL
| FI
| FOR
| FPAR
| FROM
| GENERATOR
| IF
| IMPORT
| IMPORTED
| IN
| INHERITS
| INPUT
| JOIN
| LITERAL
| LITERALS
| MACRO
| MACRODEFINITION
| MACROID
| MAP
| MOD
| NAMECLASS
| NEWTYPE
| NEXTSTATE
| NOT
| NOW
| OFFSPRING
| OPERATOR
| OPERATORS
| OR
| ORDERING
| OUT
| OUTPUT
| PARENT
| PRIORITY
| PROCEDURE
| PROCESS
| PROVIDED
| REFERENCED
| REFINEMENT
| REM
| RESET
| RETURN
| REVEALED
| REVERSE
| SAVE

| SELECT
| SELF
| SENDER
| SERVICE
| SET
| SIGNAL
| SIGNALLIST
| SIGNALROUTE
| SIGNALSET
| SPELLING
| START
| STATE
| STOP
| STRUCT
| SUBSTRUCTURE
| SYNONYM
| SYNTYPE
| SYSTEM
| TASK
| THEN
| TIMER
| TO
| TYPE
| VIA
| VIEW
| VIEWED
| WITH
| XOR

〈空格〉代表 CCITT 五号字符表中表示空格的字符。

上面给出的〈国标〉字符与 CCITT 五号字母表国际参考版本（建议 T.50）中的相同。定义这些字符的国标表示法是国家标准局的责任。

除在〈字符串〉中的外，所有〈字母〉总是当成大写对待（〈国标〉的处理方法可以由国家标准局确定）。

根据上面确定的语法，〈词法单元〉终止于第一个不能作为〈词法单元〉的一部分的那个字符。当一个〈下划线〉字符后跟有一个或多个控制字符（控制字符在建议 T.50 中定义）或空格时，所有这些字符（包括〈下划线〉）都被忽略掉。例如，A_B 与 AB 表示同一个〈名字〉。〈下划线〉的这种用法允许把〈词法单元〉拆开放在几行上。

在一个〈名字〉中当一个〈下划线〉字符后跟有一个〈字〉时，只要〈下划线〉字符前（或后）的〈字〉不构成一个〈关键字〉，则将允许用一个或多个控制字符或空格取代〈下划线〉字符。例如，AB 表示与 A_B 一样的名字。

然而，有时在〈名字〉中不写出〈下划线〉会使人误解，因此，采用了下面的一些规则：

1. 在一个〈路径项〉中，〈名字〉中的〈下划线〉必须显式地规定。

2. 对于可能有的一个〈类别〉(例如〈变量定义〉、〈视见定义〉)直接跟在一个或多个〈名字〉或〈标识符〉之后的情况，这些〈名字〉或〈标识符〉中的〈下划线〉必须显式地规定。

3. 当一个〈数据定义〉含有〈生成程序实例产生〉时，则跟在关键字 NEWTYPE 后的〈类别名〉中的〈下划线〉必须显式地规定。

一个控制字符具有与一个空格一样的意义。

在两个〈词法单元〉之间，控制字符与空格可以出现任意次数。两〈词法单元〉之间任意个数的控制字符和空格具有与一个空格一样的意义。

字符/之后紧接着一个星号 * 字符总是表示一个〈注〉的开始，而在〈注〉中的字符 * 之后紧接着字符/总是表示〈注〉的终结。一个〈注〉可以在任一〈词法单元〉之前或之后插入。

在〈宏指令体〉中要使用专门的词法规则。(参见 § 4.2.1)

2.2.2 视见度规则和标识符

抽象文法

```
标识符              ::  限定符名
限定符              =   路径项+
路径项              =   系统限定符|
                        功能块限定符|
                        功能块子结构限定符|
                        信号限定符|
                        进程限定符|
                        过程限定符|
                        类别限定符|
系统限定符          ::  系统名字
功能块限定符        ::  功能块名字
功能块子结构限定符  ::  功能块子结构名字
进程限定符          ::  进程名字
过程限定符          ::  过程名字
信号限定符          ::  信号名字
类别限定符          ::  类别名字
名字                ::  记号
```

具体正文文法

```
〈标识符〉::=
            [〈限定符〉]〈名字〉
〈限定符〉::=
```

〈路径项〉{/〈路径项〉}*

〈路径项〉::=

〈作用域单位种类〉〈名字〉

〈作用域单位种类〉::=

SYSTEM
| BLOCK
| SUBSTRUCTURE
| SIGNAL
| PROCESS
| PROCEDURE
| TYPE
| SERVICE

对于由 SERVICE 指明的〈作用域单位种类〉来说，没有对应的抽象语法。在〈服务定义〉中规定的实体的〈名字〉和〈标识符〉分别被转换成在含有〈服务定义〉的〈进程定义〉中规定的各别的〈名字〉和〈标识符〉。

〈限定符〉反映了从系统层次到定义范围的分层结构，把系统层次安排在正文的最左边。

允许省略一些最左边的〈路径项〉(除〈间接定义〉以外，参见§2.4.1) 或省略整个〈限定符〉。当整个〈限定符〉被省略，〈名字〉表示含有变量、同义词、字面值及运算符的实体类的一个实体时(参见下面的语义)，〈名字〉与一个定义的结合必须由实际的上下文来解决。在其它情况下，〈标识符〉结合于一个实体，该实体的定义范围是最近的包围作用域单位，而在这个作用域单位中，此〈标识符〉的〈限定符〉与指示该作用域单位的完整的〈限定符〉的最右边部分一样。若〈标识符〉没有包含〈限定符〉，则可省略〈限定符〉的匹配要求。

一个子信号必须受其父辈信号的限定，除非在那个地方没有其它可见信号具有同样的〈名字〉。

在下面的情况下，可以由上下文来解决：

a) 在其中使用了〈名字〉的作用域单位不是一个〈部分类型定义〉，并且，它含有一个具有那个〈名字〉的定义。该〈名字〉将表示那个定义。

b) 在其中使用了〈名字〉的作用域单位不含有任何具有那个〈名字〉的定义，或者此作用域单位是一个〈部分类型定义〉，并且，在整个〈系统定义〉中确实存在一个具有同样〈名字〉的实体的一个可见定义，如果这个名字可以与此可见定义结合。而不违反该〈名字〉出现于其中的结构的任何静态特性(类别相容性等)，则此〈名字〉将与那个定义结合。

除非是在〈视见定义〉中的〈变量标识符〉以及在一引用定义(取之于〈系统定义〉的一种定义)中用来取代〈名字〉的〈标识符〉，其余只能够使用可见标识符。

语义

确定作用域单位的纲要如下：

具体正文文法	具体图形文法
〈系统定义〉	〈系统图〉

〈功能块定义〉　　〈功能块图〉
〈进程定义〉　　〈进程图〉
〈过程定义〉　　〈过程图〉
〈功能块子结构定义〉　　〈功能块子结构图〉
〈信道子结构定义〉　　〈信道子结构图〉
〈服务定义〉　　〈服务图〉
〈部分类型定义〉
〈信号具体化〉

一作用域单位附有一个定义表，每一个定义确定了属于某一实体类并具有一个名字的一个实体。对一个〈部分类型定义〉来说，所附有的定义表包括〈运算符标记〉、〈字面值标记〉以及包括从父辈类别、从生成程序实例继承的或通过使用简化符号例如关键字 ORDERING 所蕴含的任何〈运算符标记〉和〈字面值标记〉(参见 § 5.4.1.8)。注意，〈视见定义〉并不确定一个实体。

尽管〈中缀运算符〉、带有惊叹号的〈运算符〉及〈字符串〉有它们自己的语法符号，但实际上它们都是〈名字〉，在*抽象语法*中，它们由*名字*来代表。在下面，我们把它们（也在语法上）作为〈名字〉来进行处理对待。然而，〈状态名〉、〈连接符名〉、〈生成程序形式名〉、等式中的〈值标识符〉、〈宏形式名〉和〈宏名〉具有特殊的视见度规则，因此，它们不能被限定。在〈进程体〉、〈过程体〉或〈服务体〉的外面，〈状态名〉和〈连接符名〉是不可见的。其它特殊的视见度规则将在适当的节中阐述。

每一个实体的定义范围就是对其定义的作用域单位，实体可通过〈标识符〉被引用。

一个〈标识符〉中的〈限定符〉唯一地确定了〈名字〉的定义范围。

有以下的一些实体类：

a) 系统
b) 功能块
c) 信道、信号路由
d) 信号、定时器
e) 进程
f) 过程
g) 变量（包括形式参数）、同义词、字面值、运算符
h) 类别
i) 生成程序
j) 进口实体
k) 信号表
l) 服务
m) 功能块子结构、信道子结构

在下列条件下一个〈标识符〉在一作用域单位中是可见的：

a) 如果该〈标识符〉的名字部分的定义范围就是那个作用域单位，或
b) 如果它在定义那个作用域单位的作用域单位中是可视见的，或

c) 如果此作用域单位包含一个〈部分类型定义〉在其中定义了此〈标识符〉，或

d) 如果此作用域单位包含一个〈信号定义〉在其中定义了此〈标识符〉。

在同一个作用域单位中和属于同一个实体类的两个定义不能具有相同的〈名字〉。在同一个〈部分类型定义〉中的〈运算符标记〉和〈字面值标记〉是一个例外（参见§5.2.2）：两个或多个运算符和（或）字面值可以具有同样的〈名字〉但具有不同的〈变元类别〉或不同的〈结果〉类别。

另一个例外是进口实体。对于这个实体类，在此作用域单位中的〈进口定义〉中的（〈进口名字〉、〈类别〉）偶对必须是不同的。

在具体正文文法中，一个定义的结尾关键字（ENDSYSTEM，ENDBLOCK 等）后的任选名字或标识符必须在语法上与跟在对应的起始关键字（分别为 SYSTEM，BLOCK 等）之后的名字或标识符相同。

2.2.3 非形式正文

抽象文法

非形式正文 :: …

具体正文文法

〈非形式正文〉 ::=
〈字符串〉

语义

如果在一个 SDL 系统规格中使用了非形式正文，这意味着此正文不是形式的 SDL，也就是说，SDL 不给予它任何语义。非形式正文的语义可以由某种别的方法来定义。

2.2.4 绘图规则

图形符号的尺寸可由使用者选择。

符号边界不能重叠或交叉。线形符号是这一规则的一个例外，也就是说，〈信道符〉、〈信号路由符〉、〈创建线符〉、〈流线符〉、〈实践联结符〉和〈虚线联结符〉可以相互交叉。在相互交叉的符号之间，没有逻辑上的联系。

元符号 **is followed by** 意味着一个〈流线符〉。

线形符号可以包含一段或多段直线。

当一个〈流线符〉进入另一个〈流线符〉、〈出连接符〉或〈下一状态符〉时，〈流线符〉上需要有一个箭头。在其它情况下，〈流线符〉上的箭头可有可无。〈流线符〉应画成垂直的或水平的。

〈输入符〉、〈输出符〉、〈注释符〉和〈正文扩展符〉的垂直镜象是允许的。

元符号 **is associated with** 的右边变元必须靠近其左边的变元，而不是靠近任何别的图形符

号。右边变元的语法元素必须是相互可区分的。

一图形符号中的正文必须从左上角开始，按从左到右的顺序阅读。此符号的右边界被解释为一个新行字符，它表示阅读必须在下一行（若有的话）的最左边处继续下去。

2.2.5 图的划分

下面的图形划分定义并不是具体图形文法的一部分，但使用了同样的元语言。

〈页〉 ::=
〈框架符〉 **contains**
{ 〈标题区〉〈页号区〉
{ 〈语法单元〉} * }
〈标题区〉 ::=
〈隐正文符〉 **contains** 〈标题〉
〈页号区〉 ::=
〈隐正文符〉 **contains** [〈页号〉 [(〈总页数〉)]]
〈页号〉 ::=
〈字面值名〉
〈总页数〉 ::=
〈自然数字面值名〉

〈页〉是一个起始非终结符，因此，在任何产生式规则中都不引用它。一个图可以划分成若干〈页〉，在这种情况下，限定该图范围的〈框架符〉和图〈标题〉将被每一〈页〉的〈框架符〉和〈标题〉所取代。

SDL 的使用者可以选择复制图形的介质的边界作为隐含的〈框架符〉。

为使〈标题区〉和〈页号区〉之间有一清晰的间隔，〈隐正文符〉没有显示出来，而是隐含起来。〈标题区〉位于〈框架符〉的左上角，〈页号区〉位于〈框架符〉的右上角。〈标题〉和〈语法单元〉取决于图的类型。

2.2.6 注释

注释是一个记号，用来表示与符号或正文有关的注解。

在具体正文文法中，使用了两种形式的注释。第一种形式是定义于§ 2.2.1 中的〈注〉。图 2.9.1 和图 2.9.3 中给出了例子。

第二种形式的具体语法是：

〈结束〉 ::=
[〈注释〉];
〈注释〉 ::=
COMMENT 〈字符串〉

图 2.9.2 中给出了一个例子。

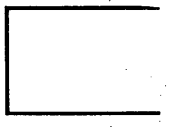
在具体图形文法中，使用了下面的语法：

〈注释区〉::=

〈注释符〉 **contains** 〈正文〉

is connected to 〈虚线联结符〉

〈注释符〉::=



〈虚线联结符〉::=

〈虚线联结符〉的一端必须连接到〈注释符〉垂直段的中点。

借助于〈虚线联结符〉，一〈注释符〉可以连接到任何图形符号。通过（在想象上）完成一长方形框以包围正文，可以把〈注释符〉看成是一封闭符号，它含有关于该图形符号的注释正文。

§2.9 节中的图 2.9.4 给出了一个例子。

2.2.7 正文扩展

〈正文扩展区〉::=

〈正文扩展符〉 **contains** 〈正文〉

is connected to 〈实线联结符〉

〈正文扩展符〉::=

〈注释符〉

〈实线联结符〉::=

〈实线联结符〉的一端必须连接到〈正文扩展符〉垂直段的中点。

借助于〈实线联结符〉，一个〈正文扩展符〉可以连接到任何图形符号。通过（在想象上）完成一长方形框，可以把〈正文扩展符〉看成是一闭合符合。

〈正文扩展符〉中所包含的正文是图形符号中正文的继续，并且可认为是包含在那个符号之中。

2.2.8 正文符号

〈正文符号〉可用于任一〈图〉中，其内容取决于图。

〈正文符号〉::=

2.3 基本数据概念

SDL 中的数据概念在§5 中定义。其概念包含 SDL 数据术语、定义新数据类型的方法和预定义数据的功能。

数据出现在数据类型定义、表达式、运算符应用、变量、值和字面值之中。

2.3.1 数据类型定义

SDL 中的数据主要与数据类型有关。一个数据类型规定了一些值的集合，规定了可以作用于这些值的一组运算符、以及规定了一组代数规则（等式），以便当这些运算符作用于值之时，确定运算符的行为。这些值、运算符和代数规则一起确定了此数据类型的特性。这些特性由数据类型定义规定。

SDL 允许定义任何需要的数据类型，包括复合结构（复合类型），只要需要，就可以形式地规定这样一个定义。相反，对于很多程序设计语言来说要考虑实现方面的问题，往往要求所提供的数据类型的集合特别是复合结构（数组结构等）是有限的。

2.3.2 变量

变量是这样一些实物，它可以通过赋值与一个值相联系。当访问变量时，就返回与其相联系的值。

2.3.3 值和字面值

一组具有某种特性的值称为一个类别。运算符的定义要规定从某种类别的值产生某种类别的值。例如，对于加号（“+”）运算符的应用，从整数类运算产生整数类值是合法的，而对布尔类的加法是非法的。

所有的类别至少有一个值。每一个值属于一个、也仅属于一个类别，这就是说，多个类别决不会有共同的值。

对大多数类别来说，有表示此类别的值的字面值形式（例如，对于整数“2”、不使用“1+1”）。可能有一个以上的字面值表示同样的值（例如可以用 12 和 012 来表示同样的整数值）。同样的字面值表示符号可能用于多个类别；例如，“A”既是一个字符，也是一个长度为 1 的字符串。某些类别可以没有字面值；例如，一个复合值常常没有自身的字面值，它的值由对它的分量的值进行复合运算来确定。

2.3.4 表达式

一个表达式表示一个值。如果一个表达式不包含变量，例如，如果它是一个给定类别的字面值，则此表达式的每一次出现都总是表示同样的值。一个含有变量的表达式在解释一个 SDL 系统的时候，可以解释为不同的值，这取决于与变量相关联的值。

2.4 系统结构

2.4.1 间接定义

〈间接定义〉是这样一种定义，它被从定义它的地方挪去，以便获得总体概观。这类似于一个宏定义（参见§4.2），但“调用”它的地方只有一个（就是定义它的地方），“调用”的方法是采用引用。

具体文法

〈间接定义〉 ::=
〈定义〉 | 〈图〉
〈系统定义〉 ::=
{ 〈正文系统定义〉 | 〈系统图〉}
{ 〈间接定义〉}*
〈定义〉 ::=
〈功能块定义〉
| 〈进程定义〉
| 〈过程定义〉
| 〈功能块子结构定义〉
| 〈信道子结构定义〉
| 〈服务定义〉
| 〈宏定义〉
〈图〉 ::=
〈功能块图〉
| 〈进程图〉
| 〈过程图〉
| 〈功能块子结构图〉
| 〈信道子结构图〉
| 〈服务图〉
| 〈宏图〉

除〈宏定义〉和〈宏图〉外，对每一个〈间接定义〉，必定有一个引用出现在〈系统定义〉、〈系统图〉或另一个〈间接定义〉中。

对每一个引用，必定存在一个对应的〈间接定义〉。

在每一个〈间接定义〉中，必须有一个〈标识符〉紧跟在开头的关键字之后。这个〈标识符〉中的〈限定符〉要么是完整的，要么就被省略掉。如果〈限定符〉被省略，则这个〈标识符〉中的〈名字〉在系统定义中且在此〈间接定义〉的实体类之中必须是唯一的。对于不是〈间接定义〉的定义，不允许在开头的关键字后的〈标识符〉中规定一个〈限定符〉（也就是说，

对于通常的定义，必须规定一个〈名字〉)。

语义

在可以对一个〈具体系统定义〉进行分析之前，每一个引用必须由对应的〈间接定义〉取代。在这种取代中，〈间接定义〉的〈标识符〉被引用中的〈名字〉替换。

2.4.2 系统

抽象文法

系统定义 :: 系统名字
功能块定义 **set**
信道定义 **set**
信号定义 **set**
数据类型定义
同义类型定义 **set**
系统名字 = 名字

一个系统定义具有一个可在限定符中使用的名字。

在系统定义中，必须至少包含有一个功能块定义。

用在与环境的接口中和系统的各功能块之间的所有信号、信道、数据类型、同义类型的定义都包含在系统定义之中。

所有预定义数据都应该在系统层次上作出定义。

具体正文文法

〈正文系统定义〉::=
SYSTEM〈系统名〉〈结束〉
{〈功能块定义〉
| 〈正文功能块引用〉
| 〈信道定义〉
| 〈信号定义〉
| 〈信号表定义〉
| 〈选择定义〉
| 〈宏定义〉
| 〈数据定义〉}+
ENDSYSTEM [〈系统名〉]〈结束〉

〈正文功能块引用〉::=
BLOCK〈功能块名〉REFERENCED〈结束〉

〈选择定义〉在§4.3.3中定义，〈宏定义〉在§4.2中，〈数据定义〉在§5.5.1中，〈功能块定义〉在§2.4.3中，〈信道定义〉在§2.5.1中，〈信号定义〉在§2.5.4中，〈信号表定义〉在§2.5.5中定义。

§2.9节中的图2.9.5给出了〈系统定义〉的一个例子。

具体图形文法

〈系统图〉::=
〈框架符〉 **contains**
{〈系统标题〉
{ {〈系统正文区〉}*
{〈宏图〉}*
〈功能块相互作用区〉} **set**}

〈框架符〉::=

〈系统标题〉::=
SYSTEM〈系统名〉

〈系统正文区〉::=
〈正文符〉 contains
{〈信号定义〉
| 〈信号表定义〉
| 〈数据定义〉
| 〈宏定义〉
| 〈选择定义〉}*

〈功能块相互作用区〉::=
{〈功能块区〉
| 〈信道定义区〉}+

〈功能块区〉::=
〈图形功能块引用〉
| 〈功能块图〉

〈图形功能块引用〉::=
〈功能块符〉 **contains** 〈功能块名〉

〈功能块符〉 ::=

〈选择定义〉在§4.3.3中定义，〈宏定义〉和〈宏图〉在§4.2中定义，〈数据定义〉在§5.5.1中定义，〈功能块图〉在§2.4.3中定义，〈信道定义区〉在§2.5.1中定义，〈信号定义〉在§2.5.4中定义，〈信号表定义〉在§2.5.5中定义。

抽象文法中的功能块定义 set 对应于一些〈功能块区〉，而信道定义 set 对应于〈信道定义区〉。

图 2.9.6 中给出了一个〈系统图〉的例子。

语义

一个系统定义是用 SDL 表示的一个系统的规格或描述。

一个系统由其边界与其环境分隔开，它含有一组功能块。系统与环境或系统内功能块之间的通信只可以利用信号进行。在一个系统内，这些信号在信道上传递。信道使功能块相互连接起来，或者是将功能块连接到系统边界。

在解释一个系统定义之前，先选择一个一致性子集（参见 § 3.2.1），这个子集称为此系统定义的一个实例。一个系统实例是由系统定义所规定的系统类型的一次实例化。抽象的 SDL 自动机对系统定义的一个实例进行解释，由此传递语义给 SDL 概念。为了解释系统定义的一个实例，必须：

a) 启动系统时间

b) 解释包含在所选的一致性划分子集中的功能块和解释连接它们的信道。

2.4.3 功能块

抽象文法

```
功能块定义    ::  功能块名
                  进程定义 set
                  信号定义 set
                  信道至路由连接 set
                  信号路由定义 set
                  数据类型定义
                  同义类型定义 set
                  [功能块子结构定义]
功能块名      =   名字
```

在功能块中，必须至少有一个进程定义和信号路由定义，除非在功能块定义中包含有功能块子结构定义。

对规定了功能块子结构定义的功能块进行划分是可行的；此语言的这个特点在 § 3.2.2 中讨论。

具体正文文法

```
〈功能块定义〉::=
        BLOCK { 〈功能块名〉 | 〈功能块标识符〉} 〈结束〉
            { 〈信号定义〉
          |   〈信号表定义〉
          |   〈进程定义〉
          |   〈正文进程引用〉
          |   〈信号路由定义〉
          |   〈宏定义〉
```

| 〈数据定义〉
| 〈选择定义〉
| 〈信道至路由连接〉}*
[〈功能块子结构定义〉|〈正文功能块子结构引用〉]
ENDBLOCK [〈功能块名〉|〈功能块标识符〉]〈结束〉

〈正文进程引用〉::=
PROCESS〈进程名〉[〈实例数〉] REFERENCED〈结束〉

〈信号定义〉在§2.5.4中定义，〈信号表定义〉在§2.5.5节，〈进程定义〉在§2.4.4节，〈信号路由定义〉在§2.5.2节，〈信道至路由连接〉在§2.5.3节，〈功能块子结构定义〉和〈正文功能块子结构引用〉在§3.2.2节，〈宏定义〉在§4.2.2节，〈数据定义〉在§5.5.1节中定义。

§2.9节的图2.9.7给出了〈功能块定义〉的一个例子。

具体图形文法

〈功能块图〉::=
〈框架符〉
contains {〈功能块名〉
{{〈功能块正文区〉}*{〈宏图〉}*
[〈进程相互作用区〉][〈功能块子结构区〉]}**set**}
is associated with{〈信道标识符〉}*

〈功能块图〉中的〈信道标识符〉标识了连接到一信号路由的某一信道。它位于〈框架符〉之外，靠近〈框架符〉处信号路由的端点。如果〈功能块图〉不含有〈进程相互作用区〉，则它必须含有〈功能块子结构区〉。

〈功能块标题〉::=
BLOCK{〈功能块名〉|〈功能块标识符〉}
〈功能块正文区〉::=
〈系统正文区〉
〈进程相互作用区〉::=
{〈进程区〉
| 〈创建线区〉
| 〈信号路由定义区〉}+
〈进程区〉::=
〈图形进程引用〉|〈进程图〉
〈图形进程引用〉::=
〈进程符〉**contains**{〈进程名〉[〈实例数〉]}
〈进程符〉::=

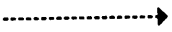
〈实例数〉定义在§2.4.4节中。

〈创建线区〉::=

〈创建线符〉

is connected to{〈进程区〉〈进程区〉}

〈创建线符〉::=



〈创建线符〉上的箭头指向一〈进程区〉,在此进程区上执行创建动作。

〈进程图〉在§2.4.4节中定义,〈信号路由定义区〉在§2.5.2中定义,〈功能块子结构区〉在§3.2.2中定义,〈宏图〉在§4.2.2中定义。

在§2.9节中,图2.9.8给出了〈功能块图〉的一个例子。

语义

一个功能块定义就象一个容器,它容纳了一个系统的一个或多个进程定义和(或)一个功能块子结构。功能块定义的目的是将多个进程组合起来,作为一个整体直接地或通过一个功能块子结构来执行某个功能。

一功能块定义提供了一个静态通信接口,其进程通过此接口方能与其它进程通信。此外,它还为进程定义提供了作用域。

解释一个功能块就是在此功能块中创建初始进程。

2.4.4 进程

抽象文法

进程定义 :: 进程名
实例数
进程形式参数 *
过程定义 **set**
信号定义 **set**
数据类型定义
同义类型定义 **set**
变量定义 **set**
视见定义 **set**
定时器定义 **set**
进程图形

实例数 :: 整数 整数

进程名 = 名字

进程图形 :: 进程起始节点
状态节点 set

进程形式参数 :: 变量名
类别引用标识符

具体正文文法

〈进程定义〉::=
PROCESS{〈进程标识符〉|〈进程名〉}
[〈实例数〉]〈结束〉
[〈形式参数〉〈结束〉][〈有效输入信号集〉]
{〈信号定义〉
| 〈信号表定义〉
| 〈过程定义〉
| 〈正文过程引用〉
| 〈宏定义〉
| 〈数据定义〉
| 〈变量定义〉
| 〈视见定义〉
| 〈选择定义〉
| 〈进口定义〉
| 〈定时器定义〉}*
{〈进程体〉
| 〈服务分解〉}
ENDPROCESS[〈进程名〉|〈进程标识符〉]〈结束〉

〈正文过程引用〉::=
PROCEDURE〈过程名〉REFERENCED〈结束〉

〈有效输入信号集〉::=
SIGNALSET[〈信号表〉]〈结束〉

〈进程体〉 ::=
〈启动〉{〈状态〉}*

〈形式参数〉 ::=
FPAR〈变量名〉{,〈变量名〉}*〈类别〉
{, 〈变量名〉{,〈变量名〉}*〈类别〉}*

〈实例数〉 ::=
([〈初始数〉],[〈最大数〉])

〈初始数〉 ::=
〈自然数简单表达式〉

〈最大数〉 ::=
〈自然数简单表达式〉

包含在实例数中的初始实例数和最大实例数都从〈实例数〉中导出。如果〈初始数〉被省略，

则表示〈初始数〉为1，如果省略〈最大数〉，则〈最大数〉为无限大。

用于演算中的〈实例数〉如下：

a) 如果对此进程没有〈正文进程引用〉，则使用〈进程定义〉中的〈实例数〉。如果它不含有〈实例数〉，则认为〈实例数〉中的〈初始数〉和〈最大数〉都省略了。

b) 如果〈进程定义〉中的〈实例数〉和〈正文进程引用〉中的〈实例数〉都省略了，则认为〈实例数〉中的〈初始数〉和〈最大数〉都省略了。

c) 如果〈进程定义〉中的〈实例数〉被省略，或者是〈正文进程引用〉中的〈实例数〉被省略，则〈实例数〉就是当前没有被省略的那一个。

d) 如果〈进程定义〉中的〈实例数〉和〈正文进程引用〉中的〈实例数〉都规定了，则这两个〈实例数〉必须是词法上相同的，并且，使用的就是这个〈实例数〉。

对于后面定义的〈进程图〉和〈图形进程引用〉中规定的〈实例数〉，类似上述的关系也适用。

〈信号定义〉在§2.5.4中定义，〈信号表定义〉在§2.5.5节，〈视见定义〉在§2.6.1.2节，〈变量定义〉在§2.6.1.1节，〈过程定义〉在§2.4.5节，〈定时器定义〉在§2.8节，〈宏定义〉在§4.2.2节，〈进口定义〉在§4.1.3节，〈选择定义〉在§4.3.3节，〈简单表达式〉在§4.3.2节，〈服务分解〉在§4.10.1节，〈数据定义〉在§5.5.1节中定义。

实例的〈初始数〉必须小于或等于〈最大数〉，〈最大数〉必须大于零。

〈合法输入信号集〉的使用在§2.5.2节模型之中规定。

§2.9节中的图2.9.9给出了一个〈进程定义〉的例子。

具体图形文法

〈进程图〉 ::=
〈框架符〉
contains{〈进程标题〉
{{〈进程正文区〉}*
{〈过程区〉}*
{〈宏图〉}*
{〈进程图形区〉|〈服务相互作用区〉}}**set**}
[**is associated with** {〈信号路由标识符〉}+]

〈信号路由标识符〉标识了一条外部信号路由，它连接到〈进程图〉中的一条信号路由。〈信号路由标识符〉应放置在〈框架符〉之外，靠近信号路由在〈框架符〉处的端点。

〈进程正文区〉 ::=
〈正文符〉 **contains** {
[〈合法输入信号集〉]
{〈信号定义〉
| 〈信号表定义〉
| 〈变量定义〉
| 〈视见定义〉
| 〈进口定义〉

| 〈数据定义〉
| 〈宏定义〉
| 〈定时器定义〉
| 〈选择定义〉} *}

〈进程标题〉::=

PROCESS {〈进程名〉 | 〈进程标识符〉}
[·〈实例数〉[〈结束〉]]
[〈形式参数〉]

〈进程图形区〉::=

〈起始区〉{〈状态区〉 | 〈入连接符区〉} *

〈信号定义〉在§2.5.4节中定义，〈信号表定义〉在§2.5.5节，〈视见定义〉在§2.6.1.2节，〈变量定义〉在§2.6.1.1节，〈过程区〉在§2.4.5节，〈定时器定义〉在§2.8节，〈宏定义〉和〈宏图〉在§4.2.2节定义，〈进口定义〉在§4.1.3节，〈选择定义〉在§4.3.3节，〈数据定义〉在§5.5.1节，〈起始区〉在§2.6.2节，〈状态区〉在§2.6.3节，〈入连接符区〉在§2.6.6节，〈服务相互作用区〉在§4.10.1节中定义。

§2.9节中的图2.9.10给出了〈进程图〉的一个例子。

语义

进程定义引入了一个进程的类型，其用意是表示一个动态行为。

在实例数中，第一个值表示系统被创建时就存在的进程的实例数，第二个值表示此进程类型能同时存在的最大实例数。

一个进程实例是一个通信的扩展有限态自动机。每当它处于一种状态的时候，根据对给定信号的接收，要执行某组动作（叫做跃迁）。跃迁的结束将导致进程处于另一个状态等待，而此状态并不一定与前一个状态不同。

有限状态自动机的概念已为人们所扩展，其扩展方面是：在一次跃迁之后所达到的状态除了受启动跃迁的信号的影响外，还可以受到对进程已知变量进行判定的影响。

同一个进程类型的几个实例可以同时存在，异步地和并行地执行，并且，可以与在系统中的不同进程类型的其他实例异步地和并行地执行。

当一个系统被创建时，初始的进程以随机的次序被创建。仅当所有的初始进程被创建后，进程间的信号通信方开始。这些初始进程的形式参数被初始化为不确定的值。

进程实例可以从创建系统时开始存在，或者是用创建请求动作来创建，创建请求动作启动了将被解释的进程；当对启动动作进行解释时，对这些进程实例的解释便开始；通过执行停止动作可以使进程消亡。

由进程实例接收的信号叫做输入信号，而向进程实例发送的信号叫做输出信号。

仅当进程实例处于一个状态时，它才可以把信号消耗掉。完整合法的输入信号集是一个并集。它包括在所有导向该进程的信号路由上的信号集合，包括〈合法输入信号集合〉，还包括隐式信号和定时器信号。

每一个进程实例有一个也仅有一个输入端口与之联系。当一个输入信号到达进程时，它被放入此进程实例的输入端口。

进程或是处于某状态正在等待，或是在活跃地执行一个跃迁。对每一个状态，有一保存信号集（也可参见§2.6.5）。当处于某种状态正在等待时，第一个输入信号（其标识符不在保存信号集之中）被从队列中取出并由进程来吸收并消耗掉。

输入端口可以保留任意个数的输入信号，这样，一进程实例的多个输入信号被排成一个队列。根据信号到达的先后次序，来安排信号在队列中的次序。如果两个或多个信号从不同路径同时到达，就任意安排它们的次序。

当一个进程被创建时，将给予它一个空的输入端口，并且，要创建本地变量并对它们赋值。

形式参数是变量，它们或是产生在系统被创建的时候（但这时没有实在参数的传递，因此，它们没有被初始化），或是产生在进程实例被动态创建的时候。

对于所有的进程实例，有四种表达式可以用来产生一 PId（参见§5.6.10）值：它们是 SELF、PARENT、OFFSPRING 和 SENDER。

a） 本进程实例（SELF）；

b） 创建本进程的进程实例（PARENT）；

c） 由本进程创建的最新进程实例（OFFSPRING）；

d） 发出最后一个被本进程消耗的输入信号的进程实例（SENDER）（也可参见§2.6.4）。

上述这些表达式将在§5.5.4.3节中进一步阐述。

SELF、PARENT、OFFSPRING 和 SENDER 可以用在进程实例内的表达式中。

对于所有在系统初始化时出现的进程实例，预定义的 PARENT 表达式所具有的值总为空（NULL）。

对于所有新创建的进程实例，预定义的 SENDER 和 OFFSPRING 表达式所具有的值为空（NULL）。

2.4.5 过程

过程由过程定义所确定。用一个“过程调用”引用过程定义来调用过程。调用过程时要使用参数：它们用于传递值，也用于为进程的执行控制变量的作用范围。至于哪些变量受进程解释的影响，由参数传递机制控制。

抽象文法

过程定义 :: 过程名
过程形式参数 *
过程定义 **set**
数据类型定义
同义类型定义 **set**
变量定义 **set**
过程图形

过程名 = 名字

过程形式参数 = *In* 参数 |
Inout 参数

In 参数 :: 变量名
类别引用标识符
Inout 参数 :: 变量名
类别引用标识符
过程图形 :: 过程启动节点
状态节点 **set**
过程启动节点 :: 跃迁

具体正文文法

〈过程定义〉 ::=
PROCEDURE {〈过程标识符〉 | 〈过程名〉} 〈结束〉
[〈过程形式参数〉〈结束〉]
{〈数据定义〉
| 〈变量定义〉
| 〈正文过程引用〉
| 〈过程定义〉
| 〈选择定义〉
| 〈宏定义〉} *
〈过程体〉
ENDPROCEDURE [〈过程名〉 | 〈过程标识符〉] 〈结束〉
〈过程形式参数〉 ::=
FPAR 〈形式变量参数〉
{, 〈形式变量参数〉} *
〈形式变量参数〉 ::=
[IN/OUT
| IN]
〈变量名〉 {, 〈变量名〉} * 〈类别〉
〈过程体〉 ::=
〈进程体〉

〈变量定义〉在§2.6.1.1节中定义，〈正文过程引用〉在§2.4.4节，〈宏定义〉在§4.2节，〈选择定义〉在§4.3.3节，〈数据定义〉在§5.5.1节，〈类别〉在§5.2.2节中定义。

在一个〈过程定义〉中，〈变量定义〉不能含有 REVEALED、EXPORTED、REVEALED EXPORTED、EXPORTED REVEALED 〈变量名〉(参见§2.6.1)。

图2.9.11给出了〈过程定义〉的一个例子。

具体图形文法

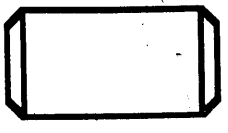
〈过程图〉 ::=
〈框架符〉 **contains** {〈过程标题〉
{ {〈过程正文区〉
| 〈过程区〉
| 〈宏图〉} *
〈过程图形区〉}**set**}

〈过程区〉 ::=
〈图形过程引用〉
| 〈过程图〉

〈过程正文区〉 ::=
〈正文区〉 **contains**
{〈变量定义〉
| 〈数据定义〉
| 〈选择定义〉
| 〈宏定义〉} *

〈图形过程引用〉 ::=
〈过程符〉 **contains** 〈过程名〉

〈过程符〉 ::=

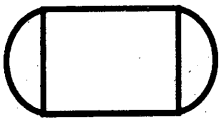


〈过程标题〉 ::=
PROCEDURE {〈过程名〉 | 〈过程标识符〉}
[〈过程形式参数〉]

〈过程图形区〉 ::=
〈过程起始区〉
{〈状态区〉 | 〈入连接符区〉} *

〈过程起始区〉 ::=
〈过程起始符〉 **is followed by** 〈跃迁区〉

〈过程起始符〉 ::=



〈变量定义〉在§2.6.1.1节中定义，〈跃迁区〉在§2.6.7.1节，〈起始区〉在§2.6.3节，〈入连接符区〉在§2.6.6节，〈宏定义〉和〈宏图〉在§4.2节，〈选择定义〉在§4.3.3节，〈数据定义〉在§5.5.1节中定义。

§2.9节中的图2.9.12给出了〈过程图〉的一个例子。

语义

过程是一种手段，它把一些项目的组合用一个名字来代表，并且用一个引用来表示这个组合。过程的规则将一种纪律加到了选择项目组合的方法上，并限制了在过程中定义的变量名字的作用范围。

过程变量是过程内部的局部变量，它既不可以透露，也不可以视见；不可以出口，又不可以进口。它在过程起始节点被解释时创建，而在过程图形的返回节点被解释时消失。

当一个过程定义被解释时，其过程图形就被解释。

仅当进程实例调用过程定义时，此过程定义才被解释，并且是由调用此进程定义的那个进程实例来解释。

过程定义的解释引起过程实例的创建，这一解释以下面的方式开始：

a) 对每一个 *In* 参数，创建一个局部变量，这个局部变量具有该 In 参数的名字和类别。此变量被赋予对应实在参数（它可能是未定义的）所给出的表达式的值。

b) 如果实在参数为空，则给予对应的形式参数的值是未定义的。

c) 没有明确属性的形式参数具有隐含的 IN 属性。

d) 对于过程定义中的每一个变量定义，创建一个局部变量，它具有变量定义的名字和类别。

e) 对于在实在参数表达式中给出的变量，每一个 Inout 参数表示一个同义名字。在整个过程图形解释的过程中。当涉及到变量的值时或给变量赋予新的值时，将要用到此同义名字。

f) 包含在过程起始节点中的跃迁被解释。

2.5 通信

2.5.1 信道

抽象文法

```
信道定义        ::  信道名
                    信道路径
                    [信道路径]
信道路径        ::  源功能块
                    目标功能块
                    信号标识符 set
源功能块        =   功能块标识符 |
                    ENVIRONMENT
目标功能块      =   功能块标识符 |
                    ENVIRONMENT
功能块标识符    =   标识符
```

信号标识符　　＝　标识符
信道名　　＝　名字

信号标识符 set 必须包含可以在信道所定义的信道路径上传递的所有信号的列表。
信道的两个端点中必须至少有一个是功能块。
如果两个端点都是功能块，则这两个功能块必须是不同的。
端点功能块必须由定义信道的作用域单位来定义。

具体正文文法

〈信道定义〉　::=
　　CHANNEL〈信道名〉
　　〈信道路径〉
　　[〈信道路径〉]
　　[〈信道子结构定义〉
　　|　〈正文信道子结构引用〉]
　　ENDCHANNEL [〈信道名〉]〈结束〉
〈信道路径〉　::=
　　{　FROM〈功能块标识符〉TO〈功能块标识符〉
　　|　FROM〈功能块标识符〉TO ENV
　　|　FROM ENV TO〈功能块标识符〉}
　　WITH〈信号表〉〈结束〉

〈信号表〉在§2.5.5节定义，〈信道子结构定义〉和〈正文信道子结构引用〉在§3.2.3节定义。

在定义两个〈信道路径〉的地方，一个信道路径必须与另一个信道路径方向相反。

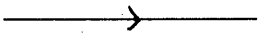
具体图形文法

〈信道定义区〉　::=
　　〈信道符〉
　　is associated with {〈信道名〉
　　{ [{〈信道标识符〉|〈功能块标识符〉}]
　　〈信号表区〉[〈信号表区〉]} **set**}
　　is connected to {〈功能块区〉{〈功能块区〉|〈框架符〉}
　　[〈信道子结构关联区〉]} **set**

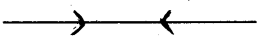
其中〈信道标识符〉用来标识一外部信道。此信道连接到由〈框架符〉圈定的〈功能块子结构图〉。〈功能块标识符〉用来标识一个外部功能块，此功能块对于由〈框架符〉圈定的〈信道子结构图〉来说是一个信道端点。

〈信道符〉　::=
　　〈信道符1〉
　　|　〈信道符2〉
　　|　〈信道符3〉

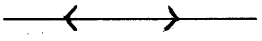
〈信道符1〉 ::=



〈信道符2〉 ::=



〈信道符3〉 ::=



〈信号表区〉在§2.5.5节定义，〈功能块区〉和〈框架符〉在§2.4.1节，〈信道子结构关联区〉在§3.2.3节定义。

对于〈信道符〉上的每一个箭头，必须有一个〈信号表区〉。〈信号表区〉必须尽可能靠近与它相关联的箭头，以免引起误解。

语义

信道表示信号的传输路由。一条信道可以认为是两个功能块之间、或功能块与其环境之间的一条或两条独立的单向信道路径。

通过信道传送的信号被送到目的地端点。

信号在信道的目的地端点出现的次序与它们在发端出现的次序相同。如果两个或多个信号同时在信道上出现，则可任意规定它们的次序。

信道可使在其上传送的信号延时，这意味着信道的每一个方向都关联着一个先入先出(FIFO) 延时队列。当一个信号在信道上出现时，它便加入延时队列。在一段不确定的、非恒定的时间间隔之后，队列中的第一个信号实例被释放，并交给连接到此信道的某一个信号路由或信道。

在两个端点之间，可以存在几个信道。相同的信号类型可以在不同的信道上传送。

2.5.2 信号路由

抽象文法

```
信号路由定义    ::  信号路由名
                    信号路由路径
                    [信号路由路径]
信号路由路径    ::  发端进程
                    目的地进程
                    信号标识符 set
发端进程        =   进程标识符|
                    ENVIRONMENT
目的地进程      =   进程标识符|
                    ENVIRONMENT
信号路由名      ==  名字
```

信号路由路径的端点必须至少有一个为进程。

如果两个端点均为进程，则这两个进程标识符必须不同。

端点进程与信号路由必须在相同的作用域单位中被定义。

具体正文文法

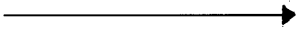
〈信号路由定义〉 ::=
SIGNALROUTE〈信号路由名〉
〈信号路由路径〉
[〈信号路由路径〉]
〈信号路由路径〉 ::=
{ FROM〈进程标识符〉TO〈进程标识符〉
| FROM〈进程标识符〉TO ENV
| FROM ENV TO〈进程标识符〉}
WITH〈信号表〉〈结束〉

〈信号表〉在§2.5.5节定义。

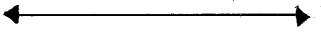
两个〈信号路由路径〉在同一地方定义时，其中一个路径必须与另一个路径方向相反。

具体图形文法

〈信号路由定义区〉::=
〈信号路由符〉
is associated with {〈信号路由名〉
{ [〈信道标识符〉]〈信号表区〉[〈信号表区〉]} **set**}
is connected to
{〈进程区〉{〈进程区〉|〈框架符〉}} **set**
〈信号路由符〉::=
〈信号路由符1〉|〈信号路由符2〉
〈信号路由符1〉::=



〈信号路由符2〉::=



信号路由符可以只在一端有箭头（单向），也可以在两端都有箭头（双向），用来表示信号流动的方向。

对于〈信号路由符〉上的每一个箭头，必须有一个对应的〈信号表区〉。〈信号表区〉必须毫不含糊地尽量靠近与它相关联的箭头。

当〈信号路由符〉连接到〈框架符〉时，则〈信道标识符〉标识了一条此信号路由所连接的信道。

语义

信号路由表示信号传输路由。信号路由可以认为是在两个进程之间，或在进程及其环境之间的一条或两条独立的单方向的信号路由路径。

由信号路由传送的信号被送到目的地端点。

在传递信号时，信号路由并不引入任何时延。

没有信号路由连接同一类型的进程实例的情况下，输出节点的解释意味着信号被直接放入目的地进程的输入端口。

在两个端点之间，可以有几条信号路由。相同的信号类型可以在不同的信号路由上传递。

模型

一个〈合法输入信号集〉包含进程所允许接收的信号。然而，〈合法输入信号集〉必须不含有定时器信号。如果一个〈功能块定义〉含有〈信号路由定义〉，则〈合法输入信号集〉(如果有的话) 可以不包含指向该进程的信号路由上的信号。

如果一个〈功能块定义〉没有包含〈信号路由定义〉，则此〈功能块定义〉中的所有〈进程定义〉必须含有一个〈合法输入信号集〉。在上述这种情况下，〈信号路由定义〉及〈信道至路由连接〉将从〈合法输入信号集〉、各个〈输出〉及在功能块边界处终止的信道中导出。在两个进程之间隐含信号路由中对应于一个给定方向的信号集是两个集合的交集；一个是定义在目的进程的〈合法输入信号集〉中的信号集合，另一个是发端进程的输出中所提及的信号集合。如果端点之一为环境，则关于此端点的输入集/输出集是通过信道朝给定方向传送的信号。

2.5.3 连接

抽象文法

信道至路由连接 :: 信道标识符
信道路由标识符 **set**

信号路由标识符 = 标识符

其它连接构件在§3节中给出。

每一个连接到包围功能块的信道标识符必须在一个且仅在一个信道至路由连接中叙述。信道至路由连接中的信道标识符必须表明连接到包围功能块的一个信道。

信道至路由连接中的每一个信号路由标识符必须与信道至路由连接在同一个功能块中定义，并且信号路由标识符有一个端点位于该功能块的边界上。在包围功能块中定义的每个信号路由标识符把环境看成是他们的一个端点。每一个信号路由标识符必须在一个且仅在一个信道至路由连接中叙述。

对于一个给定方向，信道至路由连接中之多个信号路由中的信号标识符集合的并集必须等同于在同一个信道至路由连接中及对应于同一个方向的信道标识符所传送的信号的集合。

具体正文文法

〈信道至路由连接〉 ::=
CONNECT 〈信道标识符〉
AND 〈信号路由标识符〉{，〈信号路由标识符〉}*
〈结束〉

在一个〈信道至路由连接〉之中，对一个〈信号路由标识符〉不可以叙述两次。

具体图形文法

在图形上，连接构件由关联于信号路由的、并包含在〈信号路由定义区〉之中的〈信道标识符〉来表示，(参见§2.5.2节具体图形文法)。

2.5.4 信号

抽象文法

信号定义 :: 信号名
类别引用标识符 *
[信号具体化]
信号名 = 名字
类别引用标识符在§5.2.2节定义。

具体正文文法

〈信号定义〉 ::=
SIGNAL {〈信号名〉[〈类别表〉][〈信号具体化〉]}
{,〈信号名〉[〈类别表〉][〈信号具体化〉]} *〈结束〉
〈类别表〉 ::=
(〈类别〉{,〈类别〉} *)
〈信号具体化〉在§3.3节定义，〈类别〉在§5.2.2节定义。

语义

信号实例是进程之间信息的流动，并且它是由信号定义规定的信号类型的一个实例，信号实例可以由其环境发送，也可由一个进程发送，它总是指向进程或指向环境。

每一个信号实例都关联于下列的一些值：两个表示发端进程和目的进程的 PId 值（参见§5.6.10)、在对应输出端规定的〈信号标识符〉以及其类别在信号定义中规定的其它值。

2.5.5 信号表定义

〈信号表标识符〉可以在〈信道定义〉、〈信号路由定义〉、〈信号表定义〉、〈有效输入信号集〉和〈保存表〉中用作为多个信号标识符和多个定时器信号的缩写形式。

具体正文文法

〈信号表定义〉::=
SIGNALLIST〈信号表名〉=〈信号表〉〈结束〉
〈信号表〉::=
〈信号项〉{,〈信号项〉} *
〈信号项〉::=
〈信号标识符〉|〈优先信号标识符〉|(〈信号表标识符〉)|〈定时器标识符〉

用〈信号表标识符〉所代表的那些〈信号标识符〉来取代表中的所有〈信号表标识符〉所构成的〈信号表〉与抽象文法中的信号标识符 **set** 相对应。在这样构成的每一个〈信号表〉中，每一个〈信号标识符〉必须是各不相同的。

具体图形文法

〈信号表区〉 ::=
〈信号表符〉 **contains** 〈信号表〉

〈信号表符〉 ::=

2.6 行为

2.6.1 变量

2.6.1.1 变量定义

抽象文法

变量定义 :: 变量名
类别引用标识符
[REVEALED]

变量名 = 名字

具体正文文法

〈变量定义〉 ::=
DCL [REVEALED|EXPORTED|REVEALED EXPORTED|EXPORTED REVEALED]
〈变量名〉{,〈变量名〉}*〈类别〉[:= 〈基本表达式〉]
{,〈变量名〉{,〈变量名〉}*〈类别〉[:= 〈基本表达式〉]}*〈结束〉

出口变量在§4.13节中定义。

语义

变量的语义在§2.3.2节定义。变量的值仅可以由其拥有者修改。变量的拥有者是申叙变量的进程（或过程）。变量的值仅为其拥有者知道除非此变量具有 REVEALED 属性。REVEALED 属性允许同一功能块的所有其它进程能够视见此变量，只要他们声明该变量具有视见定义。

模型

〈变量定义〉中的〈基本表达式〉或〈类别〉中的缺省值没有对应的抽象语法。在包围作用域单位的初始跃迁中，规定多个赋值语句的一个序列是派生语法。赋值语句把〈基本表达式〉赋给在〈变量定义〉中提及的所有〈变量名〉。如果〈类别〉中的某个缺省值以及该〈变量定义〉中的一个〈基本表达式〉都已确定，则应采用该〈变量定义〉中的那个〈基本表达式〉。

2.6.1.2　视见定义

抽象文法

视见定义　::　变量标识符
　　类别引用标识符

由变量标识符指明的变量定义必须具有 REVEALED 属性，并且它的类别必须和类别引用标识符所指的类别相同。

具体正文文法

〈视见定义〉　::=
　　VIEWED〈变量标识符〉{,〈变量标识符〉} * 〈类别〉
　　{,〈变量标识符〉{,〈变量标识符〉} * 〈类别〉} * 〈结束〉

在〈视见定义〉中，当下述条件满足时〈变量标识符〉中的限定符可以省略。这个条件是：如果在功能块中，有一个且仅有一个〈进程定义〉，它有一个〈变量定义〉规定了〈变量名〉。此〈变量名〉与〈视见定义〉中提及的〈变量名〉相同且具有 REVEALED 属性，并且它的〈类别〉与〈视见定义〉中所指的〈类别〉相同。

语义

视见机制使得一个进程实例可以连续地访问该视见变量值，仿佛它是局部定义的一样。然而，视见此变量的进程实例无权修改它。

2.6.2　启动

抽象文法

进程启动节点　::　跃迁

具体正文文法

〈启动〉　::=
　　START〈结束〉〈跃迁〉

具体图形文法

〈启动区〉　::=
　　〈启动符〉 **is followed by** 〈跃迁区〉

〈启动符〉　::=

语义

进程启动节点的跃迁要被解释。

2.6.3 状态

抽象文法

状态节点 :: 状态名
保存信号集
输入节点 set
状态名 = 名字

在进程图形（相应过程图形）中的状态节点具有不同的状态名。

对每一个状态节点，所有（在完整合法输入信号集中的）信号标识符或者出现在某个保存信号集中，或者出现在某个输入节点中。

在输入节点 set 中的多个信号标识符必须是不同的。

具体正文文法

〈状态〉 ::=
STATE 〈状态表〉〈结束〉
{ 〈输入部分〉
| 〈优先输入〉
| 〈保存部分〉
| 〈连续信号〉}*
[ENDSTATE [〈状态名〉] 〈结束〉]

〈状态表〉 ::=
{ 〈状态名〉 {, 〈状态名〉}* }
| 〈星号状态表〉

〈输入部分〉在§2.6.4节定义，〈保存部分〉在§2.6.5节，〈连续信号〉在§4.11节，〈星号状态表〉在§4.4节，〈优先输入〉在§4.10.2节。

当〈状态表〉含有一个〈状态名〉时，则此〈状态名〉表示一个状态节点。对于每一个状态节点，保存信号集由〈保存部分〉和任何隐含的信号保存来表示。对于每一个状态节点，输入节点 set 由〈输入部分〉和任何隐含的输入信号来表示。

仅在〈状态〉中的〈状态表〉只由一个〈状态名〉构成的情况下，才可以确定结束一个〈状态〉的任选〈状态名〉。在这种情况下，它必须与〈状态表〉中的〈状态名〉相同。

具体图形文法

〈状态区〉 ::=

〈状态符〉 **contains** 〈状态表〉 **is associated with**
{ 〈输入关联区〉
| 〈优先输入关联区〉
| 〈连续信号关联区〉
| 〈保存关联区〉} *

〈状态符〉 ::=

〈输入关联区〉 ::=

〈实线关联符〉 **is connected to** 〈输入区〉

〈保存关联区〉 ::=

〈实线关联符〉 **is connected to** 〈保存区〉

〈输入区〉在§2.6.4节定义，〈保存区〉在§2.6.5节，〈连续信号关联区〉在§4.11节，〈优先输入关联区〉在§4.10.2节定义。

一个〈状态区〉表示一个或多个状态节点。

源于〈状态符〉的〈固定关联符〉可以具有一个共同的起始路径。

语义

一个状态表示一种特定状况，在此状况中，一个进程实例可以消耗一个信号实例从而产生一个跃迁。若在此状态中没有保留的信号实例，则进程在状态中等待，直到接收到信号实例为止。

模型

当某个〈状态〉的〈状态表〉包含一个以上的〈状态名〉时，则为每一个这样的〈状态名〉创建一个此〈状态〉的备份。此后，〈状态〉由这些备份取代。

2.6.4 输入

抽象文法

输入节点 :: 信号标识符
[变量标识符] *
跃迁

变量标识符 = 标识符

[变量标识符]* 的长度必须与由信号标识符表示的信号定义中的类别引用标识符的数量一样。

变量的类别应该与那些可以由信号携带的值的类别通过位置来对应。

对于接收，不允许定义比信号实例传递的值的个数更多的变量。

具体正文文法

〈输入部分〉 ::=
INPUT〈输入表〉〈结束〉
[〈允许条件〉]〈跃迁〉

〈输入表〉 ::=
〈星号输入表〉
| 〈激励〉{,〈激励〉}*

〈激励〉 ::=
{〈信号标识符〉
| 〈定时器标识符〉}[([〈变量标识符〉]{,[〈变量标识符〉]}*)]

〈跃迁〉在§2.6.7节定义，〈允许条件〉§4.12节，〈星号输入表〉在§4.6节定义。

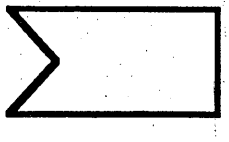
当〈输入表〉包含一个〈激励〉时，则〈输入部分〉表示一个〈输入节点〉。在抽象文法中，定时器信号(〈定时器标识符〉)也是由信号标识符表示。由于定时器信号和普通信号在许多方面具有类似的特性，所以，仅在适当的场合对它们进行区别。定时器信号的确切特性在§2.8节定义。

〈跃迁〉必须具有跃迁终端符，见§2.6.7.2节。

具体图形文法

〈输入区〉 ::=
〈输入符〉 **contains** 〈输入表〉
is followed by { [〈允许条件区〉]〈跃迁区〉

〈输入符〉 ::=



〈跃迁区〉在§2.6.7节定义，〈允许条件区〉在§4.12节定义。

其〈输入表〉包含一个〈激励〉的〈输入区〉对应于一个输入节点。〈输入符〉中的每一个〈信号标识符〉给出此〈输入符〉所表示的输入节点之一的名字。

语义

输入允许吸收所规定的输入信号实例。对输入信号实例的吸收使得此进程可以利用由信号传递的信息。与输入关联的变量被赋予所吸收信号传递的值。如果与输入关联的变量的类别都不同于信号中的类别，则弃掉信号中此类别的值。

把发送信号的进程实例的 PId 值赋给吸收信号的进程的 SENDER 表达式，此 PId 值是由信号实例所携带的。

从环境流向系统中某进程实例的信号实例，其所带的 PId 值总是与系统中任一 PId 值不同。用 SENDER 表达式可以访问此 PId 值。

模型

当某个〈输入部分〉的〈激励〉表包含不止一个〈激励〉时，则为每一个这样的〈激励〉创建一个〈输入部分〉的备份。此后，〈输入部分〉便由这些备份所取代。

2.6.5 保存

保存规定了一组信号标识符，这组信号标识符的实例在保存组所隶属的状态中与进程无关，需要保存起来以便以后处理。

抽象文法

保存信号集 :: 信号标识符 **set**

在每一个状态节点中，保存信号集中所含有的多个信号标识符必须是不同的。

具体正文文法

〈保存部分〉 ::=
SAVE〈保存表〉〈结束〉

〈保存表〉 ::=
{〈信号表〉 | 〈星号保存表〉}

〈保存表〉表示信号标识符 **set**。〈星号保存表〉是一种简化的表示法，将在 § 4.8节中说明。

具体图形文法

〈保存区〉 ::=
〈保存符〉 **contains** 〈保存表〉

〈保存符〉 ::=

语义

保存的信号根据信号到达的次序被先后保留在输入端口。

保存的效果仅对于保存所隶属的状态有效，在后继的状态中，原先被“保存”的信号实例将作为一般信号实例处理。

2.6.6 标号

具体正文文法

〈标号〉 ::=
〈连接符名〉:

定义在一个〈进程体〉中的所有〈连接符名〉必须是不同的。

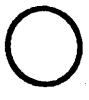
标号表示“跳转”的入口点，该“跳转”来自同一个〈进程体〉中具有同样〈连接符名〉的对应汇接语句。

“跳转”仅允许跳转到同一个〈进程体〉中的标号处。

具体图形文法

〈入连接符区〉 ::=
〈入连接符符号〉 **contains** 〈连接符名〉 **is followed by** 〈跃迁区〉

〈入连接符符号〉 ::=



〈跃迁区〉在§2.6.7.1节中定义。

〈入连接符区〉表示一〈流线符〉从对应的〈出连接符区〉连接到此〈入连接符区〉。此〈流线符〉来自同一个〈进程图形区〉或〈过程图形区〉中具有同样的〈连接符名〉的对应〈出连接符区〉。

2.6.7 跃迁

2.6.7.1 跃迁体

抽象文法

```
跃迁          ::  图形节点*
                  (终端符|判定节点)
图形节点      ::  任务节点|
                  输出节点|
                  创建请求节点|
                  调用节点|
                  置定时节点|
                  复位节点
终端符        ::  下一状态节点|
                  停止节点|
                  返回节点|
```

具体正文文法

```
〈跃迁〉      ::=
                  {〈跃迁串〉[〈终端符语句〉]}
              |   〈终端符语句〉
〈跃迁串〉    ::=
                {〈动作语句〉}+
〈动作语句〉  ::=
                [〈标号〉]〈动作〉〈结束〉
〈动作〉      ::=
                  〈任务〉
              |   〈输出〉
              |   〈优先输出〉
              |   〈创建请求〉
              |   〈判定〉
              |   〈跃迁任选〉
              |   〈置定时〉
              |   〈复位〉
              |   〈出口〉
              |   〈过程调用〉
〈终端符语句〉::=
                [〈标号〉]〈终端符〉〈结束〉
〈终端符〉    ::=
                  〈下一状态〉
              |   〈汇接〉
```

| 〈停止〉
| 〈返回〉

〈任务〉的定义见§2.7.1节，〈输出〉见§2.7.4节，〈创建请求〉见§2.7.2节，〈判定〉见§2.7.5节，〈置定时〉和〈复位〉见§2.8节。〈过程调用〉见§2.7.3节，〈下一状态〉见§2.6.7.2.1节，〈汇接〉见§2.6.7.2.2节，〈停止〉见§2.6.7.2.3节，〈返回〉见§2.6.7.2.4节，〈优先输出〉见§4.10.2节，〈跃迁任选〉见§4.3.4节，〈出口〉见§4.13节。

如果省略掉〈跃迁〉的〈终端符〉，则〈跃迁〉中的最后的动作必须含有一个用作为终端的〈判定〉(参见§2.7.5节)或〈跃迁任选〉。包含在〈判定〉和〈跃迁任选〉(〈跃迁任选〉的定义见§4.3.4节)中的所有〈跃迁〉除外。

〈终端符〉或〈动作〉不可以跟在〈终端符〉、终止〈跃迁任选〉或终止〈判定〉之后。

具体图形文法

〈跃迁区〉 ::=
[〈跃迁串区〉] **is followed by**
{〈状态区〉
| 〈下一状态区〉
| 〈判定区〉
| 〈停止符〉
| 〈合并区〉
| 〈出连接符区〉
| 〈返回符〉
| 〈跃迁任选区〉}

〈跃迁串区〉 ::=
{〈任务区〉
| 〈输出区〉
| 〈优先输出区〉
| 〈置定时区〉
| 〈复位区〉
| 〈出口区〉
| 〈创建请求区〉
| 〈过程调用区〉}
[**is followed by** 〈跃迁串区〉]

〈任务区〉的定义见§2.7.1节，〈输出区〉见§2.7.4节，〈创建请求区〉见§2.7.2节，〈判定区〉见§2.7.5节，〈置定时区〉和〈复位区〉见§2.8节，〈过程调用区〉见§2.7.3节，〈下一状态区〉见§2.6.7.2.1节，〈合并区〉见§2.6.7.2.2节，〈停止符〉见§2.6.7.2.3节，〈返回符〉见§2.6.7.2.4节，〈优先输出区〉见§4.10.2节，〈跃迁任选区〉见§4.3.4节，〈出口区〉见§4.13节，〈出连接符区〉见§2.6.7.2.2节。

跃迁包含一系列由进程执行的动作。

〈跃迁区〉对应于跃迁，〈跃迁串区〉对应于图形节点*。

语义

跃迁执行一系列动作。在一次跃迁期间，可以处理一个进程的数据和输出信号。跃迁结束时，或是停止，或是返回，或是进程进入一个状态。

2.6.7.2 跃迁终端符

2.6.7.2.1 下一状态

抽象文法

下一状态节点 :: 状态名

在下一状态中规定的状态名必须是同一个进程图或过程图中某个状态的名字。

具体正文文法

〈下一状态〉 ::=
NEXTSTATE〈下一状态体〉
〈下一状态体〉 ::=
{〈状态名〉|〈短横下一状态〉}
〈短横下一状态〉的定义见§4.9节中。

具体图形文法

〈下一状态区〉 ::=
〈状态符〉**contains**〈下一状态体〉

语义

下一状态表示跃迁的终端符。当跃迁终止时，它确定了进程实例将要达到的状态。

2.6.7.2.2 汇接

汇接被用来改变〈进程图〉或〈进程体〉中的流程，它规定下一个将被解释的〈动作语句〉是含有相同〈连接符名〉的语句。

具体正文文法

〈汇接〉 ::=
JOIN〈连接符名〉

在同一个〈进程体〉、〈过程体〉、或〈服务体〉中，相应地必须有一个且仅有一个〈连接

符名〉对应于一个〈汇接〉。

具体图形文法

〈合并区〉 ::=
〈合并符〉**is connected to**〈流线符〉
〈合并符〉 ::=
〈流线符〉
〈流线符〉 ::=
———————

〈出连接符区〉 ::=
〈出连接符符号〉**contains**〈连接符名〉
〈出连接符符号〉 ::=
〈入连接符符号〉

在一个〈进程图形区〉或〈过程图形区〉中每一个〈出连接符区〉必须有一个且仅有一个〈入连接符区〉具有同样的〈连接符名〉。

在具体正文文法中，一个〈出连接符区〉对应于一个〈汇接〉。如果一个〈合并区〉包含在一个〈跃迁区〉中，这等效于在该〈跃迁区〉中规定一个〈出连接符区〉它含有一个唯一的〈连接符名〉并把具有相同〈连接符名〉的一个〈入连接符区〉连接到该〈合并区〉中的〈流线符〉。

模型

在抽象语法中，〈汇接〉(或〈出连接符区〉)连接到一个〈跃迁串〉，该〈跃迁串〉中的第一个〈动作语句〉(或区）带有同样的〈连接符名〉。

2.6.7.2.3 停止

抽象文法

停止节点 :: ()
在一个过程图形中，必须不包含有停止节点。

具体正文文法

〈停止〉 ::=
STOP

具体图形文法

〈停止符号〉 ::=

语义

停止动作使得发出此命令的进程实例立即停止。这意味着输入端口的保留信号被放弃，并且为此进程所创建的变量和定时器、输入端口以及此进程将不再存在。

2.6.7.2.4 返回

抽象文法

返回节点 :: ()

在进程图形中，必须不含有返回节点。

具体正文文法

〈返回〉 ::= RETURN

具体图形文法

〈返回符号〉 ::

语义

以下面的方式解释返回节点：

a) 对过程启动节点的解释所创建的所有变量将不再存在。

b) 对返回节点的解释结束了对过程图形的解释，而且该过程实例不再存在了。

c) 此后，进行调用的进程（或过程）将在该调用之后的节点上继续解释下去。

2.7 动作

2.7.1 任务

抽象文法

〈任务节点〉 :: 赋值语句 |
非形式正文

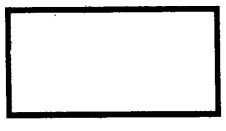
具体正文文法

〈任务〉 ::=
TASK〈任务体〉
〈任务体〉 ::=
{〈赋值语句〉{,〈赋值语句〉}*}
|{〈非形式正文〉{,〈非形式正文〉}*}

〈赋值语句〉的定义见§5.5.3节。

具体图形文法

〈任务区〉 ::=
〈任务符〉**contains**〈任务体〉
〈任务符〉 ::=



语义

任务节点的解释就是在§5.5.3节中阐述的赋值语句的解释，或是在§2.2.3节中阐述的非形式正文的解释。

模型

〈任务〉或〈任务区〉可以含有几个〈赋值语句〉或〈非形式正文〉。在上述情况下，这是派生语法用于规定一系列〈任务〉，每个任务对应于一个〈赋值语句〉或〈非形式正文〉，同时保留在〈任务体〉中定义它们的原来顺序。

在任何〈入口表达式〉扩展〈参见§4.13节〉之前，应先把此简化式进行扩展。

2.7.2 创建

抽象文法

创建请求节点 :: 进程标识符
[表达式]*
进程标识符 = 标识符

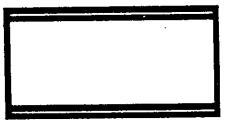
在[表达式]*中的表达式个数必须与进程标识符之进程定义中的进程形式参数的个数相同。而且每一个表达式的类别必须与在位置上相对应的进程形式参数的类别相同。

具体正文文法

〈创建请求〉 ::=
CREATE〈创建体〉
〈创建体〉 ::=
〈进程标识符〉[〈实在参数〉]
〈实在参数〉 ::=
([〈表达式〉] {, [〈表达式〉]}*)
〈表达式〉的定义见§5节。

具体图形文法

〈创建请求区〉 ::=
〈创建请求区〉 **contains** 〈创建体〉
〈创建请求符〉 ::=



〈创建请求区〉表示创建请求节点。

语义

当一个进程实例被创建时，将指定一个空的输入端口，创建变量，以给定的顺序解释实在参数表达式，并指定（如在§5.5.3节中定义的）对应的形式参数。如果缺少某个实在参数，则对应的形式参数值不确定。然后，进程开始解释进程图形中的启动节点。

然后被创建的进程与其它进程异步地和并行地执行。

此创建动作导致在同一个功能块中创建了一个进程实例。被创建进程的 PARENT 的 PId

值与创建进程的 SELF 的 PId 值是一样的。被创建进程的 SELF 和创建进程的 OFFSPRING 表达式都具有一个新创建的 PId 值（参见§5.6.10.1节）。

如果试图创建多于进程定义中所规定的最大实例数的进程实例，则不会创建出新的实例，而创建进程的 OFFSPRING 表达式将具有值 NULL，而解释继续进行。

2.7.3 过程调用

抽象文法

调用节点 :: 过程标识符
[表达式]*
过程标识符 = 标识符

[表达式]* 的长度必须与在过程标识符的过程定义中的过程形式参数的个数一致。

当一个表达式通过位置对应于一个 IN 进程形式参数，则它们必须具有相同的类别。

通过位置对应于一个 IN/OUT 进程形式参数的一个表达式必须是一个变量标识符，它具有与该进程形式参数相同的类别引用标识符。

对应于每一 IN/OUT 进程形式参数，必须有一个表达式。

具体正文文法

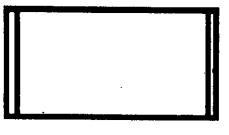
〈过程调用〉 ::=
CALL〈过程调用体〉
〈过程调用体〉 ::=
〈过程标识符〉[〈实在参数〉]

〈实在参数〉的定义见§2.7.2节。

§2.9节中的图2.9.13给出了一个〈过程调用〉的例子。

具体图形文法

〈过程调用区〉 ::=
〈过程调用符〉 **contains** 〈过程调用体〉
〈过程调用符〉 ::=



〈过程调用区〉表示调用节点。

§2.9节中的图2.9.14给出了〈过程调用区〉的一个例子。

语义

解释一个过程调用节点时，将转去解释在调用节点中引用的过程定义，并且那个过程图形将被解释。过程图形的节点的解释方法与一个进程图形的同样的节点的解释方法相同。

在被调用过程的解释完成以前，将暂停调用进程的解释。

按照给定的顺序解释实在参数表达式。

就数据和参数的解释而论，需要一种专门的语义（其说明包含在§2.4.4节中）。

2.7.4 输出

抽象文法

输出节点 :: 信号标识符
[表达式]*
[信号目的地]
传送经由

信号目的地 = 表达式

传送经由 = 信号路由标识符 **set**

[表达式]*的长度必须与在由信号标识符表示的信号定义中的类别引用标识符的个数一致。

每一个表达式的类别必须与信号定义中对应(通过位置对应)的类别引用标识符的类别相同。

对于每一个可能有的一致性子集(参见§3节)，必须至少存在一条到环境，或到一个进程类型的通信路径(既可以是隐含地通向自身的进程类型，也可以是显式地通过信号路由和可能的信道的通信路径)。此通信路径在其合法输入信号集中具有信号标识符并且是从使用了此输出节点的进程类型出发。

对于在传送经由中的每一个信号路由标识符，必须记住：在信号路由中，信号路由路径(或其中之一)中的发端进程必须与包含输出节点的进程具有相同的进程类型，并且信号路由路径必须在其信号标识符集中包括有信号标识符。

如果在传送经由中没有规定信号路由标识符，则任何一个进程，只要存在一条通信路径，都可以接收该信号。

具体正文文法

〈输出〉 ::=
OUTPUT〈输出体〉

〈输出体〉 ::=

〈信号标识符〉

[〈实在参数〉]{,〈信号标识符〉[〈实在参数〉]}*

[TO〈PId 表达式〉]

[VIA {〈信号路由标识符〉{,〈信号路由标识符〉}*

| {〈信道标识符〉{,〈信道标识符〉}* }]

〈实在参数〉的定义见§2.7.2节，〈表达式〉见§5.4.2.1节。

如果已为该功能块规定了任何信号路由的话，则不允许在此 VIA 构件中规定〈信道标识符〉。

对于一个〈输出〉中的每一个〈信道标识符〉，必须存在一条源于包围功能块的信道，它能够传递此〈输出〉中包含的〈信号标识符〉所指明的信号。

TO〈PId 表达式〉表示信号目的地。

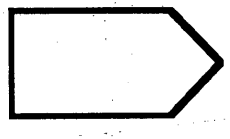
VIA 构件表示传送经由（的路由）。

具体图形文法

〈输出区〉 ::=

〈输出符〉 **contains** 〈输出体〉

〈输出符〉 ::=



语义

在输出节点中，对信号目的地 PId 表达式的解释在其它表达式之后进行。

由信号实例传递的值是输出中的实在参数的值。若在输出中对于信号定义中的一种类别没有实在参数，则由信号传递的是未确定的值。

由信号实例传递的发端 PId 值就是（执行输出动作的进程的）SELF 的值。由信号实例传递的目的地 PId 值是在输出中包含的信号目的地 PId 表达式的值。

然后把信号实例交给能够将它传递到规定的目的地进程实例的通信路径。

如果没有规定信号目的地，则必须存在一个且仅有一个接收者，它可以通过传送经由所规定的信号路由或信道来接收信号。由信号实例隐式地传递的目的地 PId 值就是此接收者的 PId 值。

环境总可以接收（导向环境之信道的信号集合中的）任何信号。

要注意的是：在两个输出节点的传送经由中，如果规定相同的信道标识符或信号路由标识符，并不自动地意味着当解释这两个输出节点时，这些信号在输入端口中排队的顺序是相同的。然而，如果两个信号是由连接发端进程与目的进程的同一个信道传递的，或者这两个进程是在同一个功能块中定义的，则将保持信号的顺序。

如果在信号定义中规定了一个同义类型，并且在输出中规定了一个表达式，则在§5.4.1.9.1节中定义的范围校验适用于此表达式。如果此范围校验等于假，则输出出错，而系统的将来行为不确定。

送往不存在（或不再存在）的进程实例的一个输出会产生一个解释错误。在对输出进行解释的同时，将检验进程实例是否存在。如果接收了信号的进程实例停止了，将使得输入端口放弃此信号，并且，不报告错误条件。

模型

如果在一个〈输出体〉中定义了几对（〈信号标识符〉〈实在参数〉），则可以以原来的〈输出体〉中的相同的顺序规定几个〈输出〉或〈输出区〉，每一个含有一对（〈信号标识符〉〈实在参数〉），这是派生语法。在每一个〈输出〉或〈输出区〉中，TO从句和VIA从句都要重复写出。应先扩展此种简化形式然后扩展在所含有的表达式中的任何其他简化形式。

2.7.5 判定

抽象文法

```
判定节点     ::  判定问题
                 判定回答 set
                 [Else 回答]
判定问题     =   表达式   |
                 非形式正文
判定回答     ::  (范围条件   |
                 非形式正文) 跃迁
Else-回答    ::  跃迁
```

这些判定回答必须是互不相容的。

如果判定问题是一个表达式，则判定回答的范围条件的类别必须与判定问题的类别相同。

具体正文文法

〈判定〉 ::=

DECISION〈问题〉〈结束〉〈判定体〉ENDDECISION

〈判定体〉 ::=

{〈回答部分〉〈Else 部分〉}

| {〈回答部分〉{〈回答部分〉}$^{+}$ [〈Else 部分〉]}

〈回答部分〉 ::=

(〈回答〉):[〈跃迁〉]

〈回答〉 ::=

〈范围条件〉|〈非形式正文〉

〈Else 部分〉 ::=

ELSE:[〈跃迁〉]

〈问题〉 ::=

〈问题表达式〉|〈非形式正文〉

〈范围条件〉的定义见§5.4.1.9.1节，〈跃迁〉见§2.6.7.1节，〈非形式正文〉见§2.2.3节。

如果〈判定体〉中的每一个〈回答部分〉和〈Else 部分〉含有一个跃迁（其中规定了一条〈终端符语句〉）；或者是含有一跃迁串（其最后一个〈动作语句〉包含一个终端判定或任选）的话，则该〈判定〉或〈跃迁任选〉（定义见§4.3.4节）是终结的。

具体图形文法

〈判定区〉 ::=

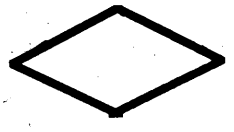
〈判定符〉 **contains** 〈问题〉

is followed by

{ {〈图形回答部分〉〈图形 Else 部分〉} **set**

| {〈图形回答部分〉{〈图形回答部分〉} + [〈图形 Else 部分〉]} **set**}

〈判定符〉 ::=



〈图形回答部分〉 ::=

〈流线符〉 **is associated with** 〈图形回答〉

is followed by 〈跃迁区〉

〈图形回答〉 ::=

〈回答〉 | (〈回答〉)

〈图形 Else 部分〉 ::=

〈流线符〉 **is associated with** ELSE

is followed by 〈跃迁区〉

〈跃迁区〉的定义见§2.6.7.1节，〈流线符〉见§2.6.7.2.2节。

〈图形回答〉和 ELSE 可以沿着所关联的〈流线符〉放置，或是放在断开的〈流线符〉之中。

从一个〈判定符〉出发的几个〈流线符〉可以有一条共同的出发路径。

〈判定区〉表示判定节点。

语义

经过判定，将解释转移到一条出路径，这条出路径的范围条件包含了对问题解释所得到的值。规定了一组对问题的可能的回答，每一个回答为所选择的路径规定了将被解释的动作的集合。

回答之一可以是对其它回答的补充。这可以通过定义 Else 回答来达到，当问题的表达式的值没有由在其它回答中定义的值或值的集合所包括时，就采用 Else 回答，它指出这时将被执行的行为的集合。

如果没有规定 Else 回答，则从问题表达式求值得出的值必须要与某一个回答匹配。

在〈问题〉和〈回答〉中的〈非形式正文〉和〈字符串〉之间，存在语法上的随意性。如果〈问题〉和所有的〈回答〉都是〈字符串〉，则所有这些都被解释为〈非形式正文〉。如果该〈问题〉或任一个〈回答〉是一个不与判定范围匹配的〈字符串〉，则此〈字符串〉表示〈非形式正文〉。判定的范围〈也就是类别〉可被确定而不需要考虑哪些是〈字符串〉的〈回答〉。

模型

如果某〈判定〉不是终端判定，则有如下的派生语法。所有的〈回答部分〉和〈Else 部分〉在它们的〈跃迁〉中都要插入一个〈汇接〉，以便跳转到跟随在此判定之后的第一个〈动作语句〉，或者，如果此判定是在一个〈跃迁串〉中的最后一个〈动作语句〉的话，则应跳转到随后的〈终端符语句〉。

2.8 定时器

抽象文法

定时器定义 :: 定时器名

类别引用标识符 *

定时器名 = 名字

置定时节点 :: 定时表达式

定时器标识符

表达式 *

复位节点 :: 定时器标识符
表达式*

定时器标识符 = 标识符

定时表达式 = 表达式

置定时节点和复位节点中的表达式*的类别必须通过位置对应于类别引用标识符*（即直接跟在由定时器标识符标志的定时器名后面的类别引用标识符*）。

在置定时节点或复位节点中的表达式必须按照给定的次序求值。

具体正文文法

〈定时器定义〉 ::=
TIMER〈定时器名〉[〈类别表〉]
{,〈定时器名〉[〈类别表〉]}* 〈结束〉

〈复位〉 ::=
RESET (〈复位语句〉{,〈复位语句〉}*)

〈复位语句〉 ::=
〈定时器标识符〉[(〈表达式表〉)]

〈置定时〉 ::=
SET〈置定时语句〉{,〈置定时语句〉}*

〈置定时语句〉 ::=
(〈定时表达式〉,〈定时器标识符〉[(〈表达式表〉)])

〈类别表〉和〈表达式表〉分别定义于§2.5.4节和§5.5.2.1节。

〈复位语句〉表示一个复位节点；〈置定时语句〉表示置定时节点。如果一个〈复位〉包含几个〈复位语句〉，则它们必须按照给定的次序进行解释。如果一个〈置定时〉包含几个〈置定时语句〉，则它们必须按照给定的次序进行解释。

具体图形文法

〈置定时区〉 ::=
〈任务符〉**contains**〈置定时〉

〈复位区〉 ::=
〈任务符〉**contains**〈复位〉

语义

一定时器实例是一个实物，它由一个进程实例所拥有，可以是活跃的或是不活跃的。后面跟有一个表达式表的一个定时器标识符的两次出现，只有在两个表达式表具有同样的值时，才是表示同一个定时器实例。

当对一个不活跃的定时器置定时的时候，即把一个时间值与此定时器联系在一起。若没有对此定时器复位或没有再进行其它的置定时，则在系统时间到达这个时间值时，就将一个信号

放入此进程的输入端口中。此信号的名字与此定时器名相同。如果把一个小于 NOW 的时间值置给此定时器，则产生同样的动作。在吸收了一个定时器信号后，从 SENDER 表达式得到的值和 SELF 表达式的值相同。当此定时器置定时时，如果给出了一个表达式表，则这些表达式的值将以相同的次序包含在此定时器信号之中。从置定时的时刻起，到定时器信号被吸收的时刻止，定时器都处于活跃状态。

如果在一个定时器定义中规定的一个类别是一个同义类型，则在§5.4.19.1中规定的范围检验在作用于置定时或复位中的对应表达式时必须得到真（True），否则，系统将出错，并且，系统将来的行为是不确定的。

当一个不活跃定时器复位时，它仍是不活跃的。

当一个活跃定时器复位时，它将失去与时间值的关联性，如果有一个对应的定时器信号保留在输入端口中，这个信号将被除去，并且，此定时器变成为不活跃的。

当对一个活跃定时器置定时的时候，这等效于复位此定时器，紧接着又对此定时器置定时。在这复位和置定时之间，此定时器仍是活跃的。

在定时器实例第一次置定时之前，它是不活跃的。

2.9 举例

```
---------
INPUT S1/ * 例子 * /;
TASK/ * 例子 * /T1:=0;
---------
```

图 2.9.1

注释举例（PR）

```
---------
INPUT I1 COMMENT '例子';
TASK T1:=0;
---------
```

图 2.9.2

注释举例（PR）

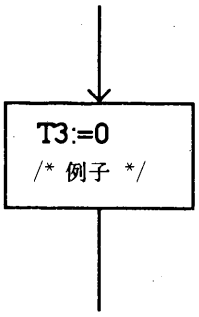


图 2.9.3

注释举例（GR）

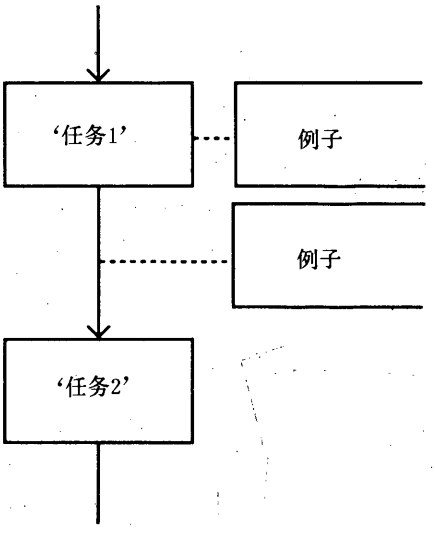


图 2.9.4

注释举例（GR）

```
SYSTEMDAEMON_GAME;

/* 此系统是一个游戏.....通过信号 "Endgame" 使游戏者脱机 */
SIGNAL Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score (Integer), Subscr, Endsubscr,
Bump;

CHANNEL C1
        FROM ENV TO Blockgame
        WITH Newgame, Probe, Result, Endgame;
        FROM Blockgame TO ENV
        WITH Gameid, Win, Lose, Score;
ENDCHANNEL C1;

CHANNEL C3 FROM Blockgame TO Blockdaemon
    WITH Subscr, Endsubscr;
ENDCHANNEL C3;

CHANNEL C4 FROM Blockdaemon TO Blockgame
    WITH Bump;
ENDCHANNEL C4;

BLOCK Blockgame REFERENCED;

BLOCK Blockdaemon REFERENCED;

ENDSYSTEM DAEMON_GAME;
```

图 2.9.5

系统规格（PR）举例

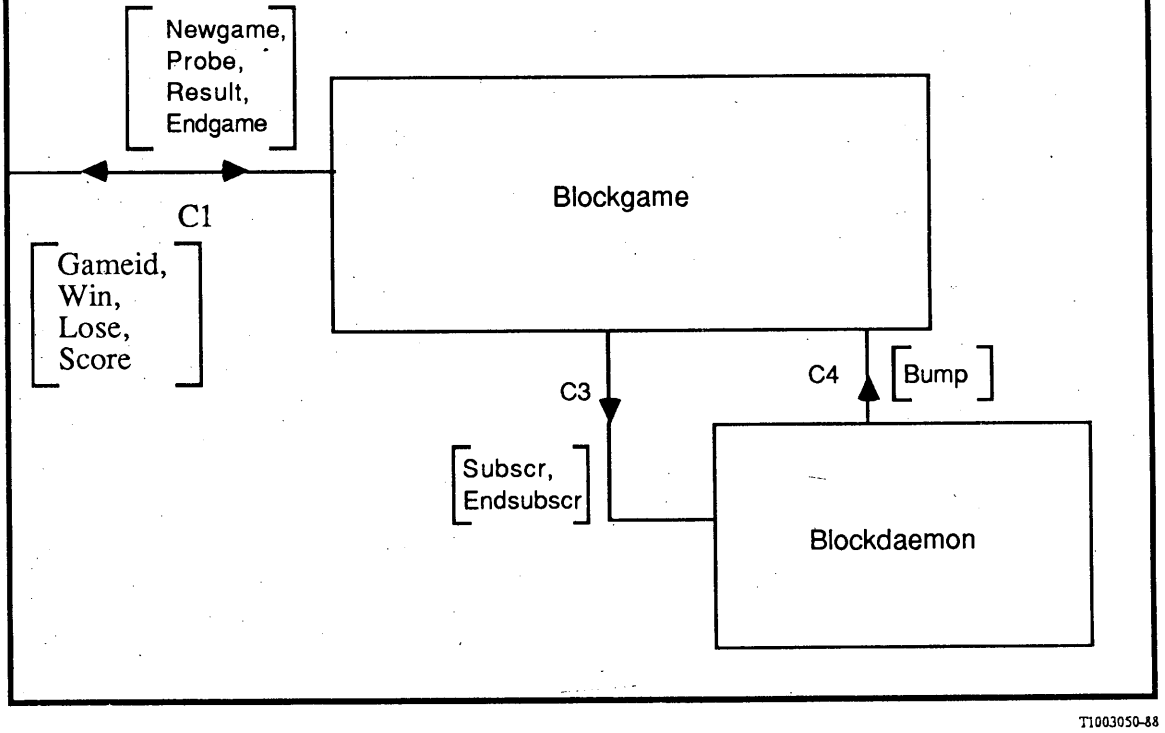
SYSTEM DAEMON_GAME 1(1)

/* 此系统是一个游戏，参加游戏的数是任意的。游戏者属于系统的环境。系统中有一个Daemon（精灵）随机地产生Bump(冲撞）信号。游戏者必须猜测已经产生的冲撞信号个数是奇数还是偶数。这种猜测通过送一个探测信号Probe 给系统来进行。如果当前已产生的冲撞信号数为奇数则系统用信号“Win”（赢）作出响应，否则，用信号“Lose”(输）作出响应。

系统跟踪每一个游戏者的得分“Score”。游戏者可以发出信号“Result”（结果）来询问他得分的当前值，这由系统用信号“Score”（得分）来回答。

在游戏者开始玩之前，他必须联机，这由信号“Newgame”（新游戏）来完成。通过信号“Endgame）（结束游戏）来使游戏者脱机。*/

SIGNAL Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score(integer), Subsrc, Endsubscr, Bump;



T1003050-88

图 2.9.6

系统规格（GR）举例

```
BLOCK Blockgame;

CONNECT C1 AND R1, R2, R3;
CONNECT C3 AND R4;
CONNECT C4 AND R5;
SIGNALROUTE R1 FROM ENV TO Monitor WITH Newgame;
SIGNALROUTE R2 FROM ENV TO Game
        WITH Probe, Result, Endgame;
SIGNALROUTE R3 FROM Game TO ENV
        WITH Gameid, Win, Lose, Score;
SIGNALROUTE R4 FROM Game TO ENV
        WITH Subscr, Endsubscr;
SIGNALROUTE R5 FROM ENV TO Game WITH Bump;

PROCESS Monitor (1, 1) REFERENCED;

PROCESS Game (0,) REFERENCED;

ENDBLOCK Blockgame
```

图 2.9.7

功能块规格（PR）举例

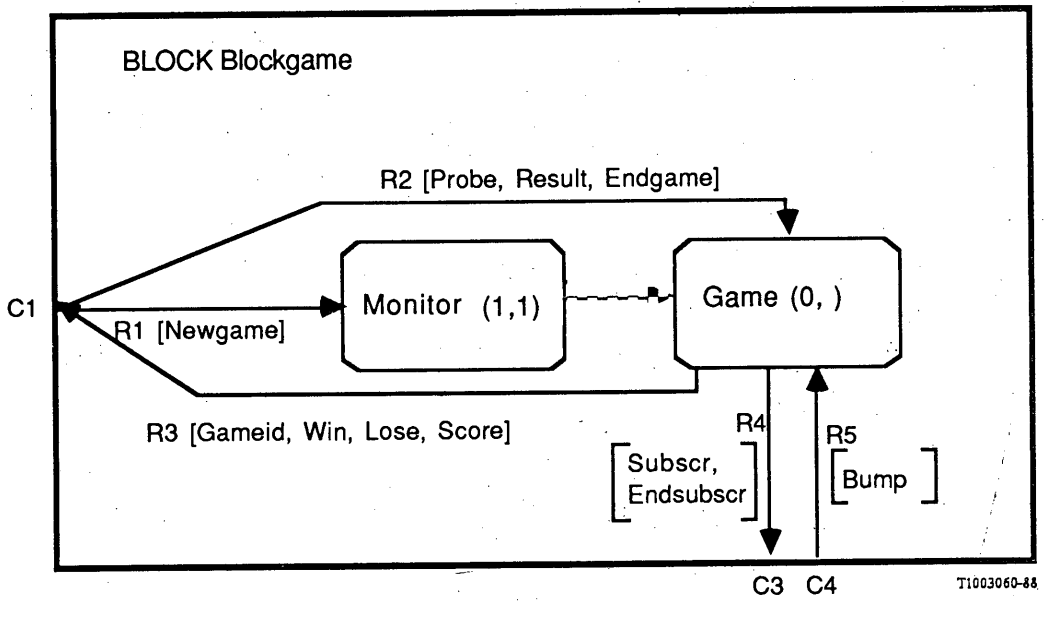


图 2.9.8

功能块图举例

```
PROCESS Game (0,);
FPAR Player Pid;

DCL
        Count Integer; /* 跟踪得分的计数器 */
START;

    OUTPUT Subscr;
    OUTPUT Gameid TO Player;
    TASK Count:=0;
NEXTSTATE Even;

STATE Even;

INPUT Probe;
    OUTPUT Lose TO Player;
    TASK Count:=Count-1;
NEXTSTATE-;

INPUT Bump;
NEXTSTATE Odd;

STATE Odd;

INPUT Bump;
NEXTSTATE Even;

INPUT Probe;
    OUTPUT Win TO Player;
    TASK Count:=Count+1;
NEXTSTATE-;

STATE * ;

INPUT Result;
    OUTPUT Score (Count) TO Player;
NEXTSTATE-;

INPUT Endgame;
    OUTPUT Endsubscr;
STOP;

ENDPROCESS Game;
```

图 2.9.9

进程规格（PR）举例

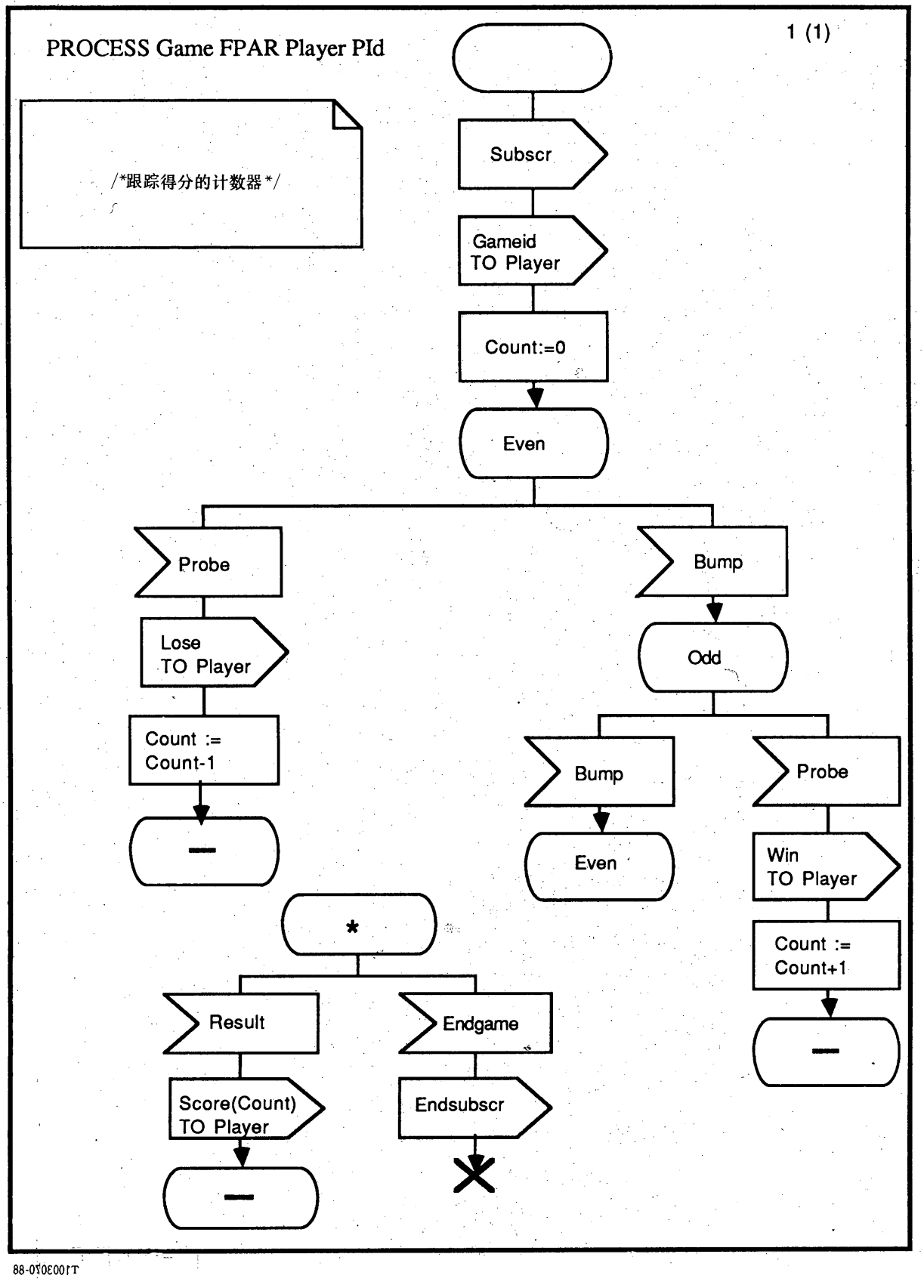


图 2.9.10

进程规格（GR）举例

```
PROCEDURE check;
/* 设信号定义如下:
    SIGNAL sigl (Boolean), sig2, sig3 (Integer, PId); */
        FPAR IN/OUT x, y Integer;
        DCL sum, index Integer,
            nice Boolean;
        START;
        TASK sum:=0,
            index:=1;
        NEXTSTATE idle;
        STATE idle;
            INPUT sig1 (nice);
            DECISION nice;
                (true): TASK' Calculate sum' ;
                        OUTPUT sig3 (sum, SENDER);
                        RETURN;
                (false): NEXTSTATE Jaj;
            ENDDECISION;
            INPUT sig2;
            ......
        ......
        ......
ENDPROCEDURE check;
```

图 2.9.11

过程规格片断举例

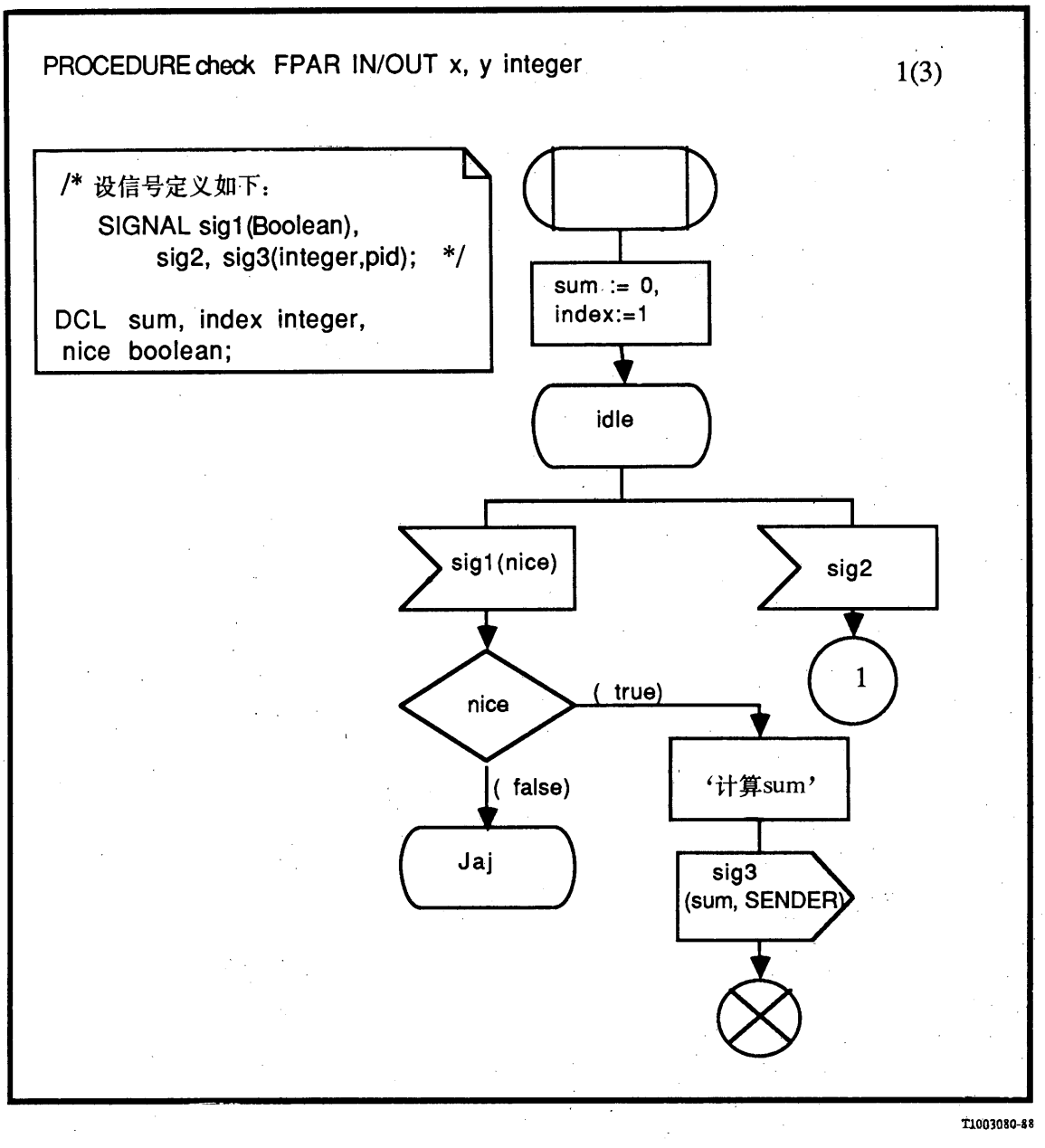


图 2.9.12

过程规格片断举例（GR）

```
/*设信号定义如下：
SIGNAL inquire (Integer, Integer, Integer); */
PROCESS alfa;
DCL a, b, c Integer;
    ......
    ......
    INPUT inquire (a, b, c);
    CALL check (a, b);
    ......
ENDPROCESS;
```

图 2.9.13

进程定义（PR）片段中过程调用举例

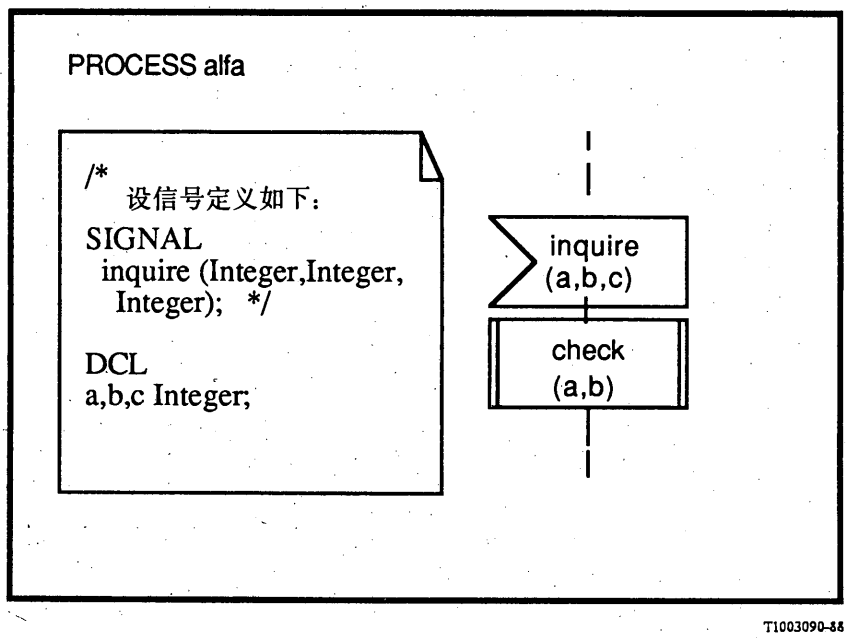


图 2.9.14

进程定义的一个片段中的过程调用举例（GR）

3 SDL 中的结构概念

3.1 引言

本节定义了一些在SDL中处理层次结构所需的概念。这些概念的基本点已在§2节中给予了定义，这里所定义的概念是对在§2节中定义的概念的严格的补充。

在本节中所引入的这些概念其目的是为SDL的用户提供手段以定义大型的和（或）复杂的系统。在§2节中所定义的概念适用于定义相对小些的系统，这种系统中的多个功能块便于在同一层次上理解和处理。当要规定一个大型或复杂系统规格时，有必要将此系统规格划分成易于处理的单元，这种单元便于分别独立地处理和理解。通常比较合适的划分方法是分若干个步骤来进行，得到的结果是用来规定系统规格的由多个单元组成的分层结构。

术语“划分”意味着把一个单元再分成若干小的子单元，它们都是此单元的组成部分。划分不影响单元的静态接口。除划分外，当系统规格的层次结构下降到较低层次时，还需要为系统的行为添加新的细节。这由术语“具体化”来表示。

3.2 划分

3.2.1 总则

一个功能块定义可以划分成一组子功能块定义、信道定义和子信道定义。同样地，一个信道定义可以划分成一组功能块定义、信道定义和子信道定义。这样，每一个功能块定义和信道定义可以有两种形式：一个未划分的形式和一个以具体语法表示的划分形式。然而，当映象成抽象语法时，信道子结构被改变了。这两种形式具有同样的静态接口，但是，由于输出信号的次序可能不一样，它们的行为在某种程度上可能有所不同。一个子功能块定义也是一个功能块定义。一个子信道定义也是一个信道定义。

在具体系统定义中，可以出现一个功能块定义的未划分形式和已划分形式，这和抽象系统定义一样。在这种情况下，一个具体系统定义包含几个一致性划分子集，每一个子集对应于一个系统实例。一个一致性划分子集是一个系统定义中功能块定义的一种选择，这样：

a) 如果它含有一个功能块定义，则它必须含有包围作用域单位的定义（如果有这样的包围作用域的话）；

b) 它必须包含所有在系统层上定义的功能块定义，并且，如果它含有一个功能块定义的子功能块定义，则它必须也含有那个功能块定义的所有其它子功能块定义。

c) 在最终的结构中，所有“叶子”功能块定义一定包含进程定义。

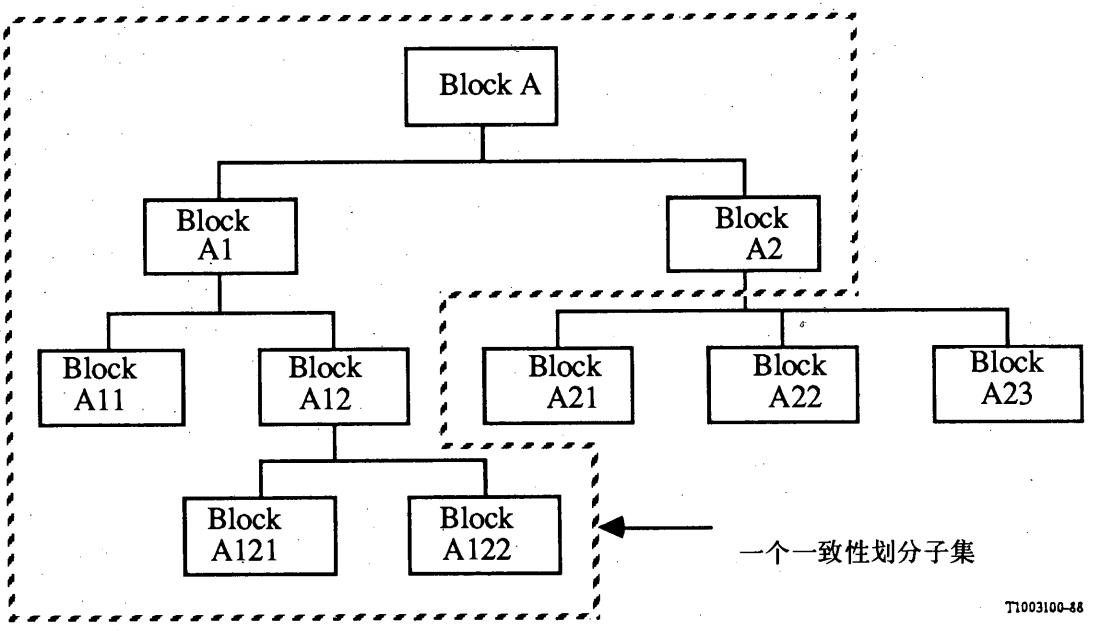


图 3.2.1

辅助图中给出的一致性划分子集举例

在解释系统时，一致性划分子集将被解释。

在一致性划分子集中，每一个叶功能块中的进程将被解释。如果这些叶功能块也含有子结构，则它们不影响解释。在非叶功能块中的子结构对视见度有作用，并且，在这些功能块中的进程将不被解释。

3.2.2 功能块划分

抽象文法

功能块子结构定义 :: 功能块子结构名
子功能块定义 **set**
信道连接 **set**
信道定义 **set**
信号定义 **set**
数据类型定义
同义类型定义 **set**

功能块子结构名 = 名字

子功能块定义 = 功能块定义

信道连接 :: 信道标识符
子信道标识符 **set**

子信道标识符 = 信道标识符

信道标识符 = 标识符

功能块子结构定义必须至少含有一个子功能块定义。在下述的内容中，若没有另外说明，则认为在此功能块子结构定义之中，含有一抽象语法项。

包含在信道定义中的一个功能块标识符必须表示一个子功能块定义。连接子功能块定义到功能块子结构定义的边界的信道定义称为子信道定义。

对于连到功能块子结构定义的每一个外部信道定义，必须有且仅有一个信道连接。在信道连接中的信道标识符必须标明这个外部信道定义。

对于从功能块子结构定义发出的信号，在一信道连接中包含的子信道标识符 **set** 所关联的那些信号标识符的并集必须与在该信道连接中包含的信道标识符所关联的那些信号标识符相同。对于进入功能块子结构定义的信号，同样的规则也是有效的。然而，此规则在信号具体化的情况下要作修改，参见§3.3节。

每一个子信道标识符必须出现在一个且仅出现在一个信道连接之中。

因为一个子功能块定义也是一个功能块定义，因此它又可以被划分。这种划分可以重复任意次数，最终得到的是一些功能块定义和它们的子功能块定义构成的一个分层树结构。一个功能块定义的子功能块定义在功能块树中位于下一个层次上，参见下面的图。

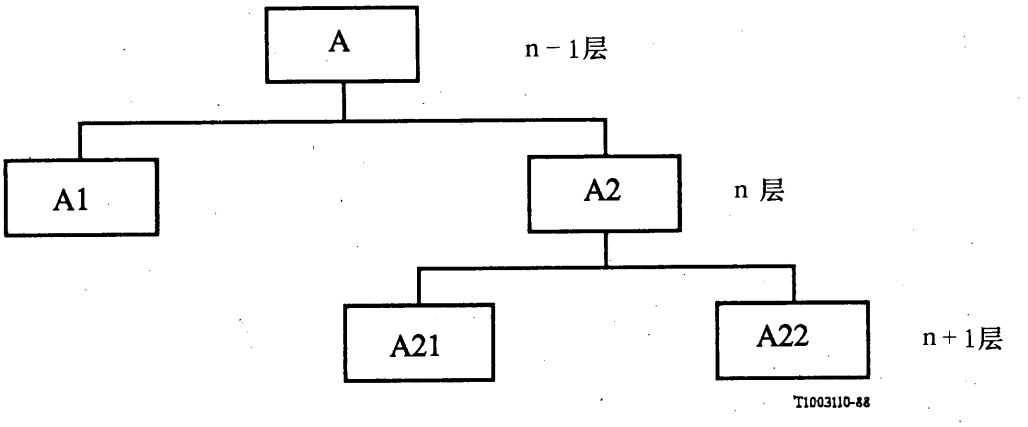


图 1/3.2.2

功能块树图

功能块树图是一种辅助图。

具体正文文法

〈功能块子结构定义〉 ::=

SUBSTRUCTURE { [〈功能块子结构名〉]

| 〈功能块子结构标识符〉} 〈结束〉

{ 〈功能块定义〉

| 〈正文功能块引用〉

| 〈信道定义〉

| 〈信道连接〉

| 〈信号定义〉

| 〈信号表定义〉

| 〈数据定义〉

| 〈选择定义〉

| 〈宏定义〉}+

ENDSUBSTRUCTURE [{ 〈功能块子结构名〉

| 〈功能块子结构标识符〉}] 〈结束〉

仅当关键字 SUBSTRUCTURE 之后的〈功能块子结构名〉与包围在〈功能块定义〉中的〈功能块名〉相同时，才可以将它省略。

〈正文功能块子结构引用〉 ::=

SUBSTRUCTURE 〈功能块子结构名〉 REFERENCED 〈结束〉

〈信道连接〉 ::=

CONNECT 〈信道标识符〉 AND 〈子信道标识符〉

{, 〈子信道标识符〉} * 〈结束〉

具体图形文法

〈功能块子结构图〉 :: =

〈框架符〉

contains { 〈功能块子结构标题〉

{ { 〈功能块子结构正文区〉} *

{ 〈宏图〉} *

〈功能块相互作用区〉} **set**}

is associated with { 〈信道标识符〉} *

此〈**信道**标识符〉用来标识一个信道。此信道连接到〈功能块子结构图〉中的一个子信道。此信道标识符放在〈框架符〉的外面，靠近在〈框架符〉上的子信道的端点。

在〈框架符〉之中，且连接到〈框架符〉的〈信道符〉表示一个子信道。

〈功能块子结构标题〉 ::=

SUBSTRUCTURE { 〈功能块子结构名〉 | 〈功能块子结构标识符〉}

〈功能块子结构正文区〉 ::=
〈系统正文区〉
〈功能块子结构区〉 ::=
〈图形功能块子结构引用〉
| 〈功能块子结构图〉
| 〈开功能块子结构图〉
〈图形功能块子结构引用〉 ::=
〈功能块子结构符〉 **contains** 〈功能块子结构名〉
〈功能块子结构符〉 ::=
〈功能块符〉
〈开功能块子结构图〉 ::=
{ { 〈功能块子结构正文区〉} *
{ 〈宏图〉} *
〈功能块相互作用区〉} **set**

当一个〈功能块子结构区〉是一个〈开功能块子结构图〉时，包围的〈功能块图〉必须不含有〈功能块正文区〉、〈宏图〉，也不含有〈进程相互作用区〉。

语义

参见§3.2.1节。

模型

把〈开功能块子结构图〉转变成〈功能块子结构图〉的方法是：在〈功能块子结构标题〉中，让〈功能块子结构名〉和〈功能块子结构标识符〉分别与包围〈功能块图〉中的〈功能块名〉和〈功能块标识符〉相同。

举例

下面给出了〈功能块子结构定义〉的一个例子。

```
BLOCK A;
    SUBSTRUCTURE A;
        SIGNAL s5 (nat), s6, s8, s9 (min);
        BLOCK a1 REFERENCED;
        BLOCK a2 REFERENCED;
        BLOCK a3 REFERENCED;
        CHANNEL c1 FROM a2 TO ENV WITH s1, s2; ENDCHANNEL c1;
        CHANNEL c2 FROM ENV TO a1 WITH s3;
            FROM a1 TO ENV WITH s1; ENDCHANNEL c2;
        CHANNEL d1 FROM a2 TO ENV WITH s7; ENDCHANNEL d1;
        CHANNEL d2 FROM a3 TO ENV WITH s10; ENDCHANNEL d2;
        CHANNEL e1 FROM a1 TO a2 WITH s5, s6; ENDCHANNEL e1;
        CHANNEL e2 FROM a3 TO a1 WITH s8; ENDCHANNEL e2;
```

```
        CHANNEL e3 FROM a2 TO a3 WITH s9; ENDCHANNEL e3;
        CONNECT c AND c1, c2;
        CONNECT d AND d1, d2;
    ENDSUBSTRUCTURE A;
ENDBLOCK A;
```

下面给出同一个例子的〈功能块子结构图〉

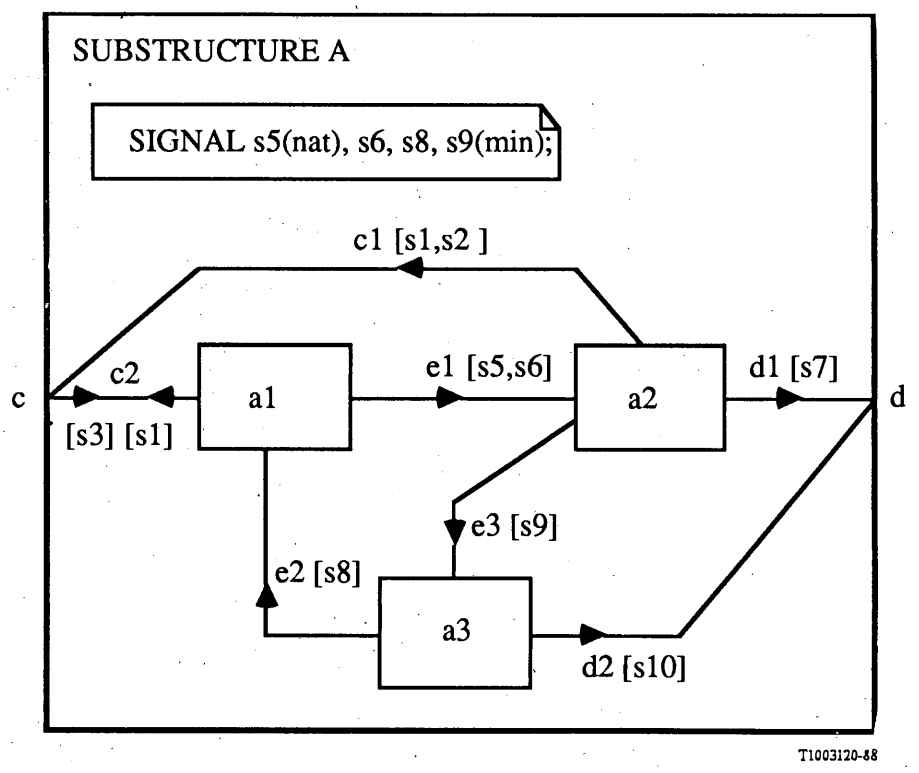


图 2/3.2.2

功能块 A 的功能块子结构图

3.2.3 · 信道划分

所有静态条件均用具体正文文法来陈述。对于具体图形文法也要满足类似的条件。

具体正文文法

```
〈信道子结构定义〉::=
    SUBSTRUCTURE {[〈信道子结构名〉]
        |〈信道子结构标识符〉}〈结束〉
    {〈功能块定义〉
        |〈正文功能块引用〉
        |〈信道定义〉
        |〈信道端点连接〉
        |〈信号定义〉
        |〈信号表定义〉
        |〈数据定义〉
        |〈选择定义〉
```

|〈宏定义〉 }+
ENDSUBSTRUCTURE[{〈信道子结构名〉
|〈信道子结构标识符〉}]〈结束〉

仅当关键字 SUBSTRUCTURE 后的〈信道子结构名〉与包围的〈信道定义〉中的〈信道名〉相同时，才可以将它省略。

〈正文信道子结构引用〉::=
SUBSTRUCTURE〈信道子结构名〉REFERENCED〈结束〉
〈信道端点连接〉::=
CONNECT {〈功能块标识符〉|ENV} AND〈子信道标识符〉
{,〈子信道标识符〉} * 〈结束〉

对被划分的〈信道定义〉的每一个端点，必须正好有一个〈信道端点连接〉。在一个〈信道端点连接〉中的〈功能块标识符〉或 ENVIRONMENT 必须是被划分的〈信道定义〉中的一个端点。

具体图形文法

〈信道子结构图〉::=
〈框架符〉
contains {〈信道子结构标题〉
{{〈信道子结构正文区〉} *
{〈宏图〉} *
〈功能块相互作用区〉} **set**}
is associated with {〈功能块标识符〉|ENV}+

此〈功能块标识符〉或 ENV 用来标识被划分的信道的一个端点。此〈功能块标识符〉应放在〈框架符〉之外，且靠近所关联的子信道连接到〈框架符〉的端点。在〈框架符〉之中且连到〈框架符〉的〈信道符〉表示子信道。

〈信道子结构标题〉::=
SUBSTRUCTURE {〈信道子结构名〉
|〈信道子结构标识符〉}
〈信道子结构正文区〉::=
〈系统正文区〉
〈信道子结构关联区〉::=
〈虚线关联符〉
is connected to〈信道子结构区〉
〈信道子结构区〉::=
〈图形信道子结构引用〉
| 〈信道子结构图〉

〈图形信道子结构引用〉::=

〈信道子结构符〉**contains**〈信道子结构名〉

〈信道子结构符〉::=

〈功能块符〉

模型

把包含一个〈信道子结构定义〉的〈信道定义〉变换成一个〈功能块定义〉和两个〈信道定义〉，使得：

a) 两个〈信道定义〉都各自连接到此功能块和连接到原先信道的一个端点。这些〈信道定义〉具有独自的新名字，并且，在 VIA 构件中，要用适当的新信道的引用来取代对原先信道的每一个引用。

b) 此〈功能块定义〉具有独自的新名字，并且，它仅含有一个具有同样名字的〈功能块子结构定义〉，所包含的定义与原先的〈信道子结构定义〉相同。新的〈功能块定义〉中的限定符将被改变，以包括此功能块名。从原先的〈信道子结构定义〉得到的两个〈信道端点连接〉用两个〈信道连接〉来表示，那里的〈功能块标识符〉或 ENV 被适当的新信道所取代。

这一变换必须在一般系统的那些变换之后立即进行，参见 § 4.3节。

举例

下面给出一个〈信道子结构定义〉的例子

```
CHANNEL CFROM A TOB WITH s1;
     FROM B TO A WITH s2;
     SUBSTRUCTURE C;
          SIGNAL s3 (hel), s4 (boo), s5;
          BLOCK b1 REFERENCED;
          BLOCK b2 REFERENCED;
          CHANNEL c1 FROMENV TO b1 WITH s1;
               FROM b1 TO ENV WITH s2; ENDCHANNEL c1;
          CHANNEL c2 FROM b2 TO ENV WITH s1;
               FROM ENV TO b2 WITH s2; ENDCHANNEL c2;
          CHANNEL e1 FROM b1 TO b2 WITH s3; ENDCHANNEL e1;
          CHANNEL e2 FROM b2 TO b1 WITH s4, s5; ENDCHANNEL e2;
          CONNECT A AND c1;
          CONNECT B AND c2;
     ENDSUBSTRUCTURE C;
ENDCHANNEL C;
```

这个例子的〈信道子结构图〉在下面给出。

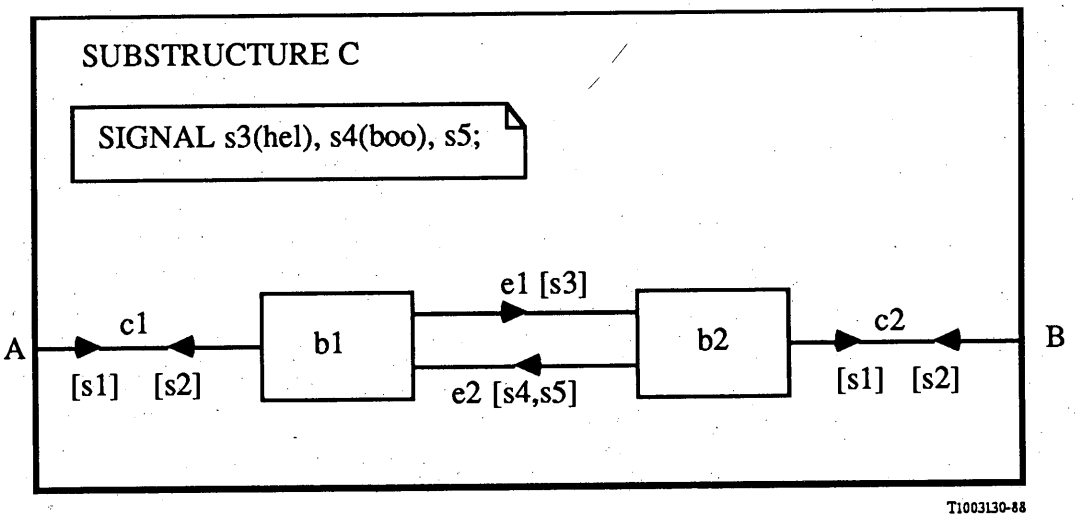


图 3.2.3

信道 C 的信道子结构图

3.3 具体化

具体化是通过将一个信号定义细分成一组子信号定义来完成的。子信号定义也就是信号定义，它又可以再细分。这种具体化可以重复任意次数，最后得到一个由若干信号定义及其子信号定义所组成的分层结构。要注意的是，并不把信号定义的子信号定义看成是此信号定义的分量。

抽象语法

信号具体化 :: 子信号定义 set
子信号定义 :: [REVERSE] 信号定义

对每一个信道连接，必须记住的是：对每一个关联于信道标识符的信号标识符，或者是此信号标识符至少关联于一个子信道标识符，或者是它的每一个子信号标识符至少关联于一个子信道标识符。这是关于划分的对应规则的一种变化。

在一个进程定义的完整有效输入信号集中或在一个进程定义的输出节点之中，两个信号不可以处在同一个信号的不同具体化层次上。

具体正文文法

〈信号具体化〉::=
REFINEMENT

{〈子信号定义〉}$^{+}$

ENDREFINEMENT

〈子信号定义〉::=

[REVERSE]〈信号定义〉

语义

当规定一个信号由一信道传递时，则此信道将自动地成为该信号的所有子信号的传递者。当此信道被划分或分解成子信道时，就可以进行具体化。在这种情况下，子信道将传递子信号以代替具体化的信号。子信号的方向由传递它的子信道确定，一个子信号的方向可能与具体化信号的方向相反，这由关键字 REVERSE 来指明。当一个信道被分解成信号路由时，信号不能被具体化。

当一个系统定义含有信号具体化时，应把概念限于一致性划分子集。这种系统定义含有几个一致性具体化子集。

一个一致性具体化子集是一种由下列规则限制的一致性划分子集：

当选择此一致性划分子集时，在连到信道某个端点的各个信号路由上的信号的集合必须不包含子信号的父辈信号，并且，除非此信道的另一个端点是此系统的 ENVIRONMENT 第一个端点的信号集合必须等于连接到另一端点的各个信号路由上的信号的集合。

举例

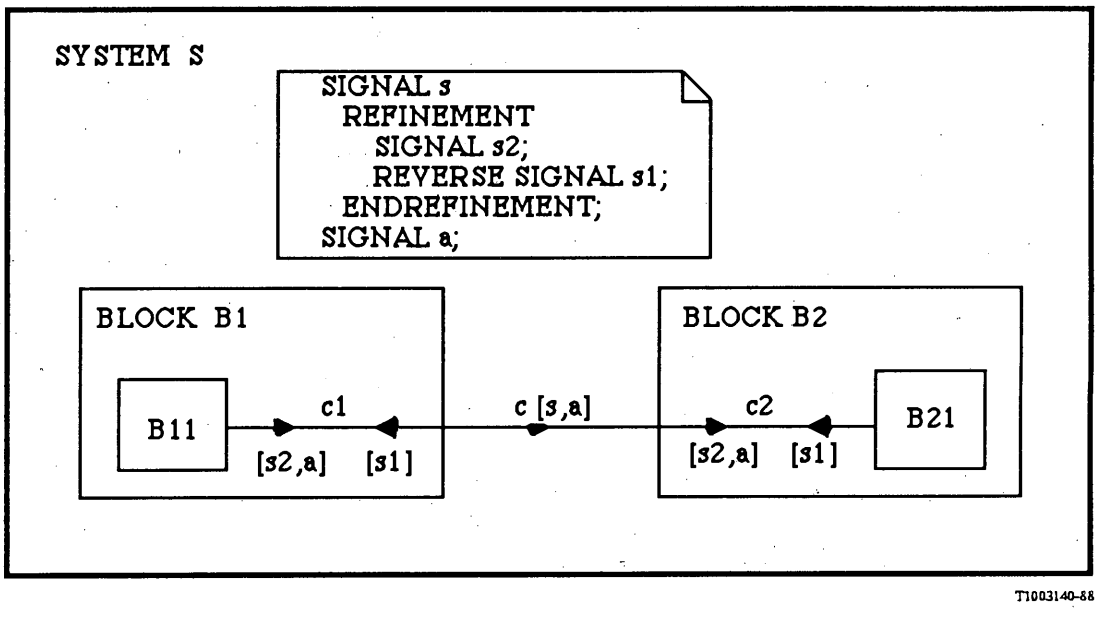


图 3.3

包含信号具体化的系统图

在上面的例子中，在功能块定义B1和B2中得到具体化的是信号s而不是信号a。在最高的具体化层次上，B1和B2中的进程利用信号s和信号a进行通信。在下一层次上，B 11和B 21中的进程利用信号s1、s2以及a进行通信。

注意：有两个功能块定义，B1和B2。仅在其中之一进行具体化是不允许的，这是因为在一个信号和其子信号之间，没有动态的转换，而仅有一种静态的关系。

4 SDL 中的补充概念

4.1 引言

本章将定义一些补充概念。这些补充概念都是标准的简化符号，并且用 SDL 的基本概念采用具体语法构造了这些概念的模型。它们的引入除了在本建议的其它章节中定义的简化符号之外，还将为 SDL 的用户提供一些方便。

简化符号的特性是用下面的方法得到的：即用基本概念来构造简化符号模型，或是把简化符号变换成基本概念。为了保证能方便地和无二义地应用简化符号以及减少几个简化符号结合在一起时的副作用，这些概念以下面规定的次序来进行变换：

1. 宏 § 4.2
2. 类属系统 § 4.3
3. 星号状态 § 4.4
4. 状态表 § 2.6.3
5. 状态的多次出现 § 4.5
6. 星号输入 § 4.6
7. 星号保存 § 4.7
8. 激励表 § 2.6.4
9. 输出表 § 2.7.4
10. 隐式跃迁 § 4.8
11. 下一状态短横 § 4.9
12. 服务 § 4.10
13. 连续信号 § 4.11
14. 允许条件 § 4.12
15. 进口和出口值 § 4.13

当定义本节中的概念时，上面的次序也应遵循。所规定的变换次序的意义是：在一个序号为 n 的简化符号的变换中，可以使用另一个序号为 m 的简化符号，条件是 m>n。

因为对于简化符号没有抽象语法，所以在定义简化符号时采用图形语法的术语或采用正文语法的术语。是选用图形语法术语还是选用正文语法术语，要根据实际的考虑。没有将简化符号的使用限于一种特定的具体语法。

4.2 宏

在以下正文中，按照一般的概念来使用术语宏定义和宏调用，既用于 SDL/GR 也用于 SDL/PR。一个宏定义是包含多个图形符号和（或）多个词法单元的一个组合，它可以出现在〈具体系统定义〉中的一个或多个地方。在每一个这样的地方用一个宏调用来指明。在对一个〈具体系统定义〉进行分析以前，每一个宏调用必须由对应的宏定义所取代。

4.2.1 词法规则

〈形式名〉::=

[〈名字〉%]〈宏参数〉

{%〈名字〉%〈宏参数〉|%〈宏参数〉}*[%〈名字〉]

4.2.2 宏定义

具体正文文法

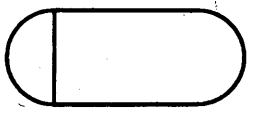
〈宏定义〉::=
MACRODEFINITION〈宏名〉
[〈宏形式参数组〉]〈结束〉
〈宏体〉
ENDMACRO[〈宏名〉]〈结束〉
〈宏形式参数组〉::=
FPAR〈宏形式参数〉{,〈宏形式参数〉}*
〈宏形式参数〉::=
〈名字〉
〈宏体〉::=
{〈词法单元〉|〈形式名〉}*
〈宏参数〉::=
〈宏形式参数〉
| MACROID

各〈宏形式参数〉必须是不同的。一个宏调用的各〈宏实在参数〉必须与相应的〈宏形式参数〉一一对应。

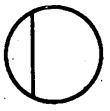
〈宏体〉必须不含有关键字 ENDMACRO 和 MACRODEFINITION。

具体图形文法

〈宏图〉::=
〈框架符〉**contains** {〈宏标题〉〈宏体区〉}
〈宏标题〉::=
MACRODEFINITION〈宏名〉[〈宏形式参数组〉]
〈宏体区〉::=
{{〈任意区〉}*
〈任意区〉[**is connected to** 〈宏体口1〉]} **set**
|{〈任意区〉 **is connected to** 〈宏体口2〉
〈任意区〉 **is connected to** 〈宏体口2〉
{〈任意区〉[**is connected to** 〈宏体口2〉]}* }**set**
〈宏引入线符〉::=



〈宏引出线符〉 ::=



〈宏体口1〉 ::=
〈引出线符〉 **is connected to** { 〈框架符〉
[**is associated with** 〈宏标号〉]
| 〈宏引入线符〉 [{**contains** | **is associated with**} 〈宏标号〉]
| 〈宏引出线符〉 [{**contains** | **is associated with**} 〈宏标号〉]}
〈宏体口2〉 ::=
〈引出线符〉 **is connected to** { 〈框架符〉
is associated with 〈宏标号〉
| 〈宏引入线符〉 {**contains** | **is associated with**} 〈宏标号〉
| 〈宏引出线符〉 {**contains** | **is associated with**} 〈宏标号〉}
〈宏标号〉 ::=
〈名字〉
〈引出线符〉 ::=
〈伪引出线符〉
| 〈流线符〉
| 〈信道符〉
| 〈信号路由符〉
| 〈实线关联符〉
| 〈虚线关联符〉
| 〈创建线符〉
〈伪引出线符〉 ::=
〈实线关联符〉
〈任意区〉 ::=
〈系统正文区〉
| 〈功能块相互作用区〉
| 〈信号表区〉
| 〈功能块区〉
| 〈功能块正文区〉
| 〈进程相互作用区〉
| 〈图形过程引用〉
| 〈进程正文区〉
| 〈进程图形区〉
| 〈合并区〉
| 〈跃迁串区〉
| 〈状态区〉
| 〈输入区〉
| 〈保存区〉
| 〈置定时区〉
| 〈复位区〉
| 〈出口区〉
| 〈正文扩展区〉
| 〈信道子结构关联区〉
| 〈信道子结构区〉
| 〈功能块子结构区〉

| 〈优先输入区〉
| 〈连续信号区〉
| 〈入连接符区〉
| 〈下一状态区〉
| 〈进程区〉
| 〈信道定义区〉
| 〈创建线区〉
| 〈信号路由定义区〉
| 〈图形进程引用〉
| 〈进程图〉
| 〈启动区〉
| 〈输出区〉
| 〈优先输出区〉
| 〈任务区〉
| 〈创建请求区〉
| 〈过程调用区〉
| 〈过程区〉
| 〈判定区〉
| 〈出连接符区〉
| 〈过程正文区〉
| 〈过程图形区〉
| 〈过程启动区〉
| 〈功能块子结构正文区〉
| 〈功能块相互作用区〉
| 〈服务区〉
| 〈服务信号路由定义区〉
| 〈服务正文区〉
| 〈服务图形区〉
| 〈服务启动区〉
| 〈注释区〉
| 〈宏调用区〉
| 〈输入关联区〉
| 〈保存关联区〉
| 〈任选区〉
| 〈信道子结构正文区〉
| 〈跃迁任选区〉
| 〈服务相互作用区〉
| 〈优先输入关联区〉
| 〈连续信号关联区〉
| 〈允许条件区〉

一个〈伪引出线符〉除了〈宏标号〉之外不得有任何东西与它关联。

如果一个〈引出线符〉不是一个〈伪引出线符〉，宏调用中的对应〈入口符〉必须是一个〈伪入口符〉。

〈宏体〉可以出现在〈任意区〉中提及的任一正文中。

语义

〈宏定义〉含有词法单元，而〈宏图〉含有语法单元。因此，正文语法和图形语法中两种宏构件的映象一般是不可能的。出于同样的原因，尽管正文语法和图形语法有某些共同的规则，但它们有各自的细则。

在整个系统定义中，不管宏定义在哪里出现，〈宏名〉都是可见的。宏调用可以出现在对应的宏定义之前。

宏定义内可以包含宏调用，但宏定义不能够直接调用自身，也不能够通过在其它宏定义中的宏调用来间接地调用自身。

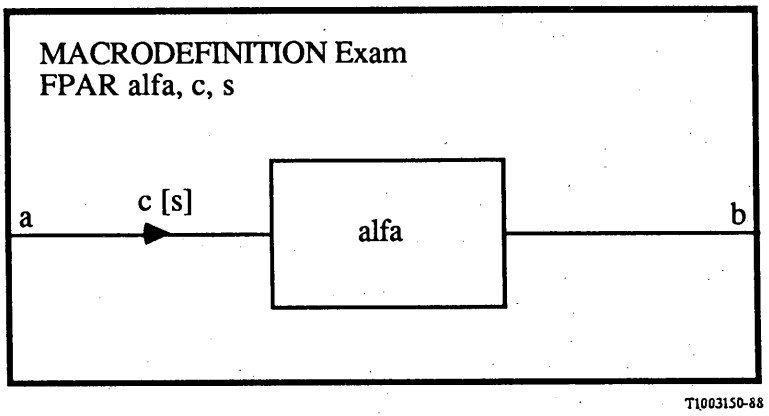
在每一个宏定义中，关键字 MACROID 可以用作为一种伪的宏形式参数。没有〈宏实在参数〉可以赋予给它，并且，对于一个宏定义的每一个扩展，它由一个唯一的〈名字〉所取代（在一个扩展中，对每一个 MACROID 的出现，将使用同样的〈名字〉）。

举例

下面给出了〈宏定义〉的一个例子：

```
MACRODEFINITION Exam
FPAR alfa, c, s, x;
    BLOCK alfa REFERENCED;
    CHANNEL c FROM x TO alfa WITH s; ENDCHANNEL c;
ENDMACRO Exam;
```

下面给出了同一个例子的〈宏图〉。然而，在此情况下，〈宏形式参数〉x 是不需要的。



4.2.3　宏调用

具体正文文法

〈宏调用〉::=

MACRO〈宏名〉[〈宏调用体〉]〈结束〉

〈宏调用体〉 ::=

(〈宏实在参数〉{,〈宏实在参数〉}*)

〈宏实在参数〉 ::=

{〈词法单元〉}*

〈词法单元〉不能够是一个逗点“,”或右圆括号“)”。如果在〈宏实在参数〉中需要这两个字符的任何一个的话，则此〈宏实在参数〉必须是一个〈字符串〉。如果〈宏实在参数〉是一个〈字符串〉，则当此〈宏实在参数〉取代〈宏形式参数〉时，将使用〈字符串〉的值。

〈宏调用〉可以出现在任何允许〈词法单元〉出现的地方。

具体图形文法

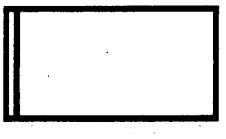
〈宏调用区〉 ::=

〈宏调用符〉 **contains** {〈宏名〉[〈宏调用体〉]}

[**is connected to**

{〈宏调用口1〉|〈宏调用口2〉{〈宏调用口2〉} +}]

〈宏调用符〉 ::=



〈宏调用口1〉 ::=

〈引入线符〉[**is associated with** 〈宏标号〉]

is connected to 〈任意区〉

〈宏调用口2〉 ::=

〈引入线符〉 **is associated with** 〈宏标号〉

is connected to 〈任意区〉

〈引入线符〉 ::=

〈伪引入线符〉

| 〈流线符〉

| 〈信道符〉

| 〈信号路由符〉

| 〈实线关联符〉

| 〈虚线关联符〉

| 〈创建线符〉

〈伪引入线符〉 ::=

〈实线关联符〉

除〈宏标号〉外，不能有任何关联于〈伪引入线符〉的东西。

对每一个〈引入线符〉，必须有一个〈引出线符〉在对应的〈宏图〉之中，关联于同一个〈宏标号〉。对于一个非〈伪引入线符〉的〈引入线符〉，对应的〈〈引出线符〉必须是一个〈伪引出线符〉。

除了〈伪引入线符〉和〈伪引出线符〉的情况外，可能有多个（正文的）〈词法单元〉关联于某个〈引入线符〉或〈引出线符〉。在这种情况下，最靠近〈宏图〉的〈宏调用符〉或〈框架符〉的〈词法单元〉被看作是关联于〈引入线符〉或〈引出线符〉的〈宏标号〉。

〈宏调用区〉可以出现在允许区出现的任何地方。然而，在〈宏调用符〉和任何其它靠近的图形符之间，需要有一定的空间。如果根据语法规则，这种空间必须不为空的话，那么就用〈伪引入线符〉把〈宏调用符〉连接到靠近的图形符上。

语义

系统定义可以包含宏定义和宏调用。在对这样的一个系统定义进行分析之前，必须扩展所有的宏调用。一个宏调用的扩展意味着用宏定义的一个备份来取代此宏调用。此宏定义与宏调用具有相同的〈宏名〉。

当一个宏定义被调用时，它便被扩展。这意味着创建了此宏定义的一个备份，并且，此备份中〈宏形式参数〉的每一次出现将由此宏调用的对应〈宏实在参数〉取代，然后，此备份中的宏调用（如果有的话）将被扩展。在〈宏形式参数〉被〈宏实在参数〉取代时，〈形式名〉中的所有百分符（%）都被除去。

在〈宏形式参数〉和〈宏实在参数〉之间，应当有一一对应的关系。

图形语法规则

〈宏调用区〉由〈宏图〉的备份以下面的方式取代之。

所有的〈宏引入线符〉和〈宏引出线符〉都被删掉。〈伪引出线符〉由具有同样的〈宏标号〉的〈引入线符〉取代。〈伪引入线符〉由具有同样的〈宏标号〉的〈引出线符〉取代。然后，附到〈引入线符〉和〈引出线符〉的〈宏标号〉被去掉。没有对应的〈宏调用口1〉或〈宏调用口2〉的〈宏体口1〉和〈宏体口2〉也被删去。

举例

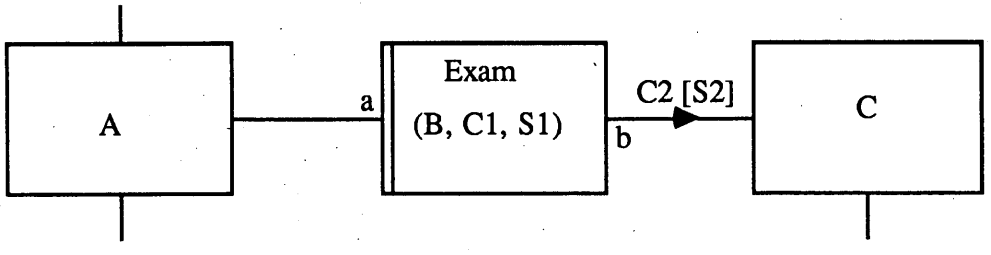
下面给出〈宏调用〉的一个例子，它位于一个〈功能块定义〉的某个片断之中。

```
.....
BLOCK A REFERENCED;
MACRO Exam (B, C1, S1, A);
BLOCK C REFERENCED;
CHANNEL C2 FROM B TOC WITH S2; ENDCHANNEL C2;
.....
```

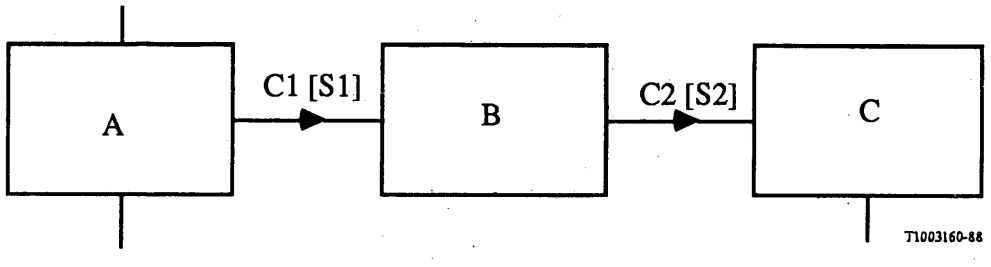
运用§4.2.2节中的例子对上面的宏调用进行扩展将给出下面的结果。

```
.....
BLOCK A REFERENCED;
BLOCK B REFERENCED;
CHANNEL C1 FROM A TO B WITH S1; ENDCHANNEL C1;
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2; ENDCHANNEL C2;
.....
```

下面给出了位于〈功能块相互作用区〉的某个片断中的同一个例子的〈宏调用区〉。



该宏调用的扩展给出如下结果。



4.3 类属系统

为满足各种需要，一个系统规格可以具有可选择部分和带有未定义值的系统参数。这样一种系统规格称为种类的，系属的，通用的。我们称这种系统为类属系统。其通用特性借助于外部同义词（它类似于过程定义中的形式参数）来确定。一个类属系统的规格可通过选择其合适的子集和为每一个系统参数提供一个值来裁剪，所得到的系统规格不含有外部同义词，并被称为一个特定的系统规格。

4.3.1 外部同义词

具体正文文法

〈外部同义词定义〉::=
SYNONYM〈外部同义词名〉〈预定义类别〉=EXTERNAL
〈外部同义词〉::=
〈外部同义词标识符〉

〈外部同义词定义〉可以出现在允许〈同义词定义〉的任何地方，参见§5.4.1.13节。〈外部同义词〉可以用在允许〈同义词〉的任何地方，参见§5.4.2.3节。预定义类别是：布尔 Boolean、字符 Character、字符串 Charstring、整数 Integer、自然数 Natural、实数 Real、进程标识符 PId、持续时间 Duration 或时间 Time。

语义

〈外部同义词〉是一个〈同义词〉，其值没有在系统定义中确定。这由关键字 EXTERNAL 来指明，它被用来代替〈简单表达式〉。

类属系统定义是一个系统定义，它含有一些〈外部同义词〉或含有在跃迁任选中的〈非形式正文〉（参见§4.3.4节）。一个特定系统定义是通过为这些〈外部同义词〉提供值和把〈非形式正文〉变换为形式构件从一个类属系统定义来创建。至于这是怎么完成的，与抽象语法的关系怎样，这些就不是此语言定义的部分了。

4.3.2 简单表达式

具体正文文法

〈简单表达式〉::=
〈基本表达式〉
〈简单表达式〉必须仅含有预定义类别的运算符、同义词和字面值。

语义

简单表达式是一种基本表达式。

4.3.3 任选定义

具体正文文法

〈选择定义〉::=
SELECT IF (〈布尔简单表达式〉)〈结束〉
{〈功能块定义〉
| 〈正文功能块引用〉
| 〈信道定义〉
| 〈信号定义〉
| 〈信号表定义〉
| 〈数据定义〉
| 〈进程定义〉
| 〈正文进程引用〉
| 〈定时器定义〉
| 〈服务信号路由定义〉
| 〈信道连接〉
| 〈信道端点连接〉
| 〈变量定义〉
| 〈视见定义〉
| 〈进口定义〉
| 〈过程定义〉
| 〈正文过程引用〉
| 〈服务定义〉
| 〈正文服务引用〉
| 〈信号路由定义〉
| 〈信道至路由连接〉
| 〈信号路由连接〉
| 〈选择定义〉
| 〈宏定义〉 }+
ENDSELECT〈结束〉

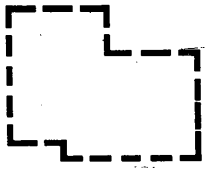
〈布尔简单表达式〉不应依赖于〈选择定义〉中的任何一个定义。一个〈选择定义〉必须只含有在那个地方语法上是允许的那些定义。

具体图形文法

〈任选区〉::=
〈任选符〉 **contains**
{SELECT IF (〈布尔简单表达式〉)
{〈功能块区〉
| 〈信道定义区〉
| 〈系统正文区〉
| 〈功能块正文区〉
| 〈进程正文区〉
| 〈过程正文区〉
| 〈功能块子结构正文区〉
| 〈信道子结构正文区〉
| 〈服务正文区〉
| 〈宏图〉

〈进程区〉
〈信号路由定义区〉
〈创建线区〉
〈过程区〉
〈任选区〉
〈服务区〉
〈服务信号路由定义区〉} +}

〈任选符〉是一个具有实线角的虚线多边形，例如：



〈任选符〉在逻辑上含有被其边界所切断的任意一维图形符号的整体（也就是说，符号的一个端点在边界外边）。

〈布尔简单表达式〉不应依赖于在〈任选区〉中的任何区或图。

除了在〈进程图形区〉、〈过程图形区〉和〈服务图形区〉中之外，〈任选区〉可以出现在任何地方。〈任选区〉中必须只含有在那个地方语法上是允许的那些区和图。

语义

如果〈布尔简单表达式〉的值为假（false），则不选择包含在〈选择定义〉和〈任选符〉中的构件。如果其值为真（true），则将选择这些构件。

模型

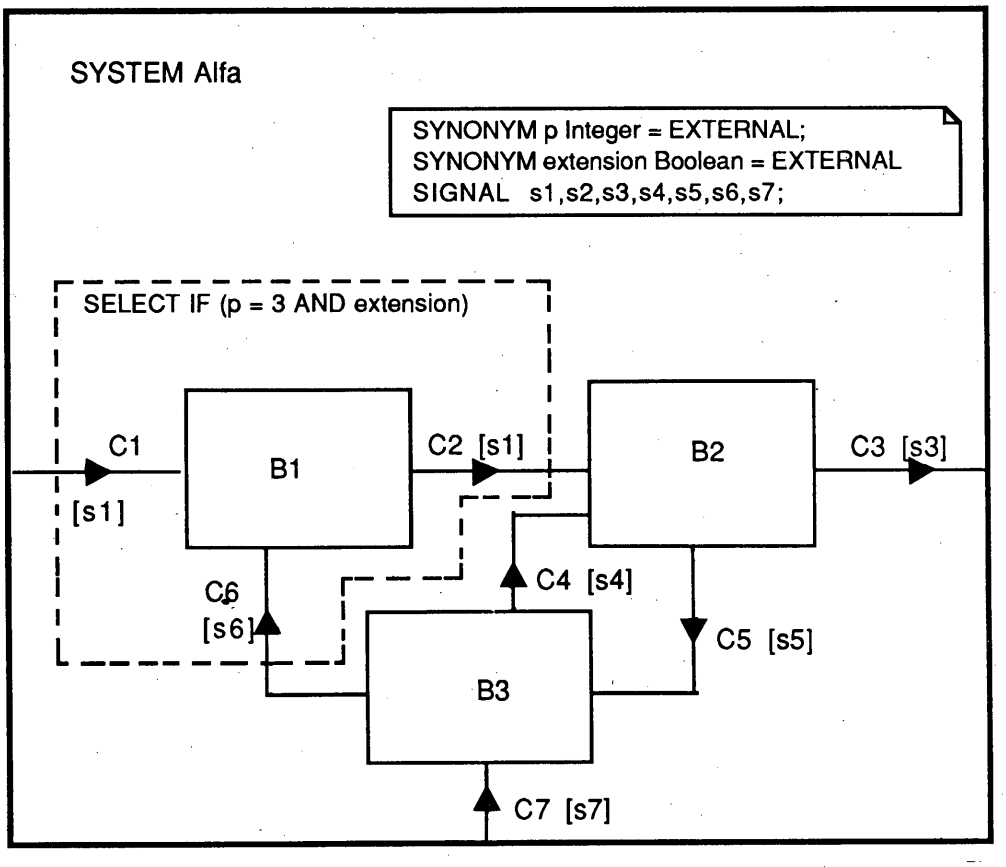
在变换时，〈选择定义〉和〈任选区〉被删除掉，并用所含有的被选出的构件（如果有的话）来取代该〈选择定义〉和该〈任选区〉。

举例

在系统 Alfa 中，有 B1、B2和 B3三个功能块。功能块 B1和连接到其上的信道是可选择的，它取决于外部同义词 P 和 extension 的值。在 SDL/PR 中，这个例子表达如下。

```
SYSTEM Alfa;
    SYNONYM p Integer=EXTERNAL;
    SYNONYM extension Boolean=EXTERNAL;
    SIGNAL s1, s2, s3, s4, s5, s6, s7;
    SELECT IF (p=3 AND extension);
      BLOCK B1 REFERENCED;
      CHANNEL C1 FROM ENV TO B1 WITH s1; ENDCHANNEL C1;
      CHANNEL C2 FROM B1 TO B2 WITH s2; ENDCHANNEL C2;
      CHANNEL C6 FROM B3 TO B1 WITH s6; ENDCHANNELC6;
    ENDSELECT;
    CHANNEL C3 FROM B2 TO ENV WITH s3; ENDCHANNEL C3;
    CHANNEL C4 FROM B3 TO B2 WITH s4; ENDCHANNEL C4;
    CHANNEL C5 FROM B2 TO B3 WITH s5; ENDCHANNEL C5;
    CHANNEL C7 FROM ENV TO B3 WITH s7; ENDCHANNEL C7;
    BLOCK B2 REFERENCED;
    BLOCK B3 REFERENCED;
ENDSYSTEM Alfa;
```

同样的例子用 SDL/GR 语法来表达如下。



4.3.4 任选跃迁串

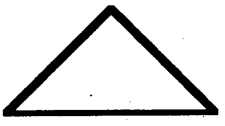
具体正文文法

〈跃迁任选〉::=
ALTERNATIVE〈备择问题〉〈结束〉
{〈回答部分〉〈Else 部分〉
| 〈回答部分〉{〈回答部分〉} + [〈Else 部分〉]}
ENDALTERNATIVE
〈备择问题〉::=
〈简单表达式〉
| 〈非形式正文〉

〈回答〉中的每一个〈基本表达式〉必须是一个〈简单表达式〉。在一个〈跃迁任选〉中的多个〈回答〉必须是互不相交的。如果〈备择问题〉是一个〈表达式〉，则各个〈回答〉的范围条件的类别必须与此〈备择问题〉的类别相同。

具体图形文法

〈跃迁任选区〉::=
〈跃迁任选符〉**contains** {〈备择问题〉}
is followed by {〈任选引出线1〉{〈任选引出线1〉 | 〈任选引出线2〉}
{〈任选引出线1〉} * }**set**
〈跃迁任选符〉::=



〈任选引出线1〉::=
〈流线符〉**is associated with**〈图形回答〉
is followed by〈跃迁区〉
〈任选引出线2〉::=
〈流线符〉**is associated with** ELSE
is followed by〈跃迁区〉

〈任选引出线1〉和〈任选引出线2〉中的〈流线符〉连接到〈跃迁任选符〉的底部。从一个〈跃迁任选符〉出发的〈流线符〉可以有一个共同的初始路径。〈图形回答〉和 ELSE 可以沿着相关联的〈流线符〉置放，或是放在断开的〈流线符〉之中。

在一个〈跃迁任选区〉中的〈图形回答〉必须是互不相交的。

语义

如果〈回答〉含有〈备择问题〉的值，则选择〈任选引出线1〉中的构件。如果没有一个〈回答〉含有〈备择问题〉的值，则选择〈任选引出线2〉中的构件。

如果没有提供〈任选引出线2〉，并且没有选择到出路径，则此选择是不合法的。

模型

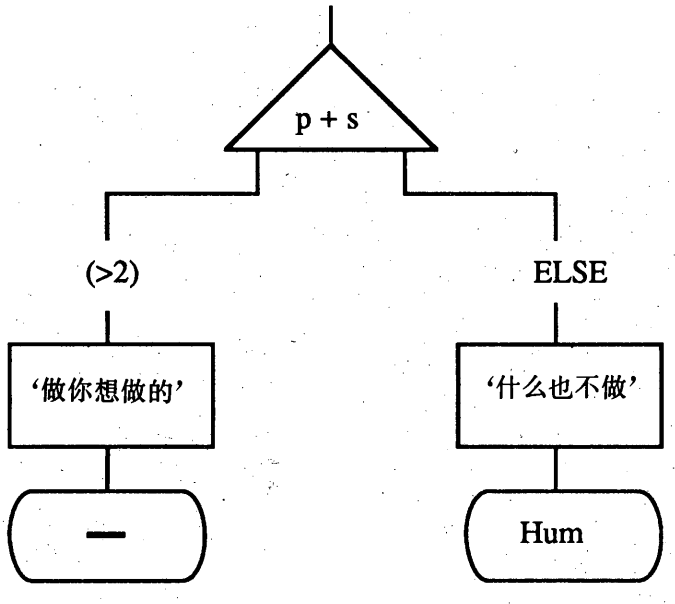
〈跃迁任选〉和〈跃迁任选区〉在变换时被删除，取代它们的是所包含的选择出来的构件。

举例

包含有〈跃迁任选〉的一个〈进程定义〉的一个片断如下所示。p 和 s 是同义词

```
.....
ALTERNATIVE p+s;
      (>2): TASK' 做你想做的' ;
              NEXTSTATE -;
      ELSE:TASK ' 什么也不做' ;
            NEXTSTATE HUM;
ENDALTERNATIVE;
.....
```

同一个例子用具体图形语法给出如下。



4.4 星号状态

具体正文文法

〈星号状态表〉::=
〈星号〉[(〈状态名〉{,〈状态名〉}*)]
〈星号〉::=
*

在一个〈进程体〉、〈过程体〉或〈服务体〉中，必须至少有一个〈状态表〉不同于〈星号状态表〉。一个〈星号状态表〉中的〈状态名〉必须是不同的，并且，必须包含在包围的〈进程体〉、〈过程体〉或〈服务体〉中的其它〈状态表〉之中

〈星号状态表〉中的那些〈状态名〉不能够把全部的〈状态名〉都包括进来。这里〈状态名〉指的是包围的〈进程体〉、〈过程体〉或〈服务体〉中的〈状态名〉。

具体图形文法

包含〈星号状态表〉的一个〈状态区〉必须与〈下一状态区〉不一致。

模型

〈星号状态表〉被变换成一个〈状态表〉，它包含所涉及的〈进程体〉、〈服务体〉或〈过程体〉的所有〈状态名〉，但不包括此〈星号状态表〉中的〈状态名〉。

4.5 状态的多次出现

具体正文文法

一个〈状态名〉可以出现在一个〈进程体〉、〈服务体〉或〈过程体〉的多个〈状态〉之中。

模型

当几个〈状态〉含有同一个〈状态名〉时，这些〈状态〉将被拼接成具有那个〈状态名〉的一个状态。

4.6 星号输入

具体正文文法

〈星号输入表〉::=
〈星号〉

一个〈状态〉最多可以含有一个〈星号输入表〉。一个〈状态〉不能同时包含〈星号输入表〉和〈星号保存表〉。

模型

一个〈星号输入表〉被变换成一个〈激励〉表，它包含包围的〈进程定义〉或〈服务定义〉的完整合法输入信号集，但不包含隐信号的〈信号标识符〉和包含在该〈状态〉中的其它〈输入表〉、〈保存表〉的〈信号标识符〉和§4.10节中〈服务定义〉的所有〈优先输入〉中的〈信号标识符〉。

4.7 星号保存

具体正文文法

〈星号保存表〉::=
〈星号〉

一个〈状态〉最多可以含有一个〈星号保存表〉。一个〈状态〉不能既包含〈星号输入表〉又包含〈星号保存表〉。

模型

一个〈星号保存表〉被变换成一个〈激励表〉，它包含包围的〈进程定义〉或〈服务定义〉的完整合法输入信号集，但不包含隐信号的〈信号标识符〉和包含在该〈状态〉中的其它〈输入表〉、和〈保存表〉，以及§4.10节中〈服务定义〉的所有〈优先输入〉中的〈信号标识符〉。

4.8 隐式跃迁

具体正文文法

在一个〈进程定义〉或〈服务定义〉中的完整合法输入信号集中的一个〈信号标识符〉，可以在一个〈状态〉中的〈输入表〉、〈优先输入表〉和〈保存表〉中的〈信号标识符〉的集合中省略。

模型

对每一个〈状态〉有一个隐式〈输入部分〉，它含有的一个〈跃迁〉仅包含返回到同一个〈状态〉的〈下一状态〉。

4.9 短横下一状态

具体正文文法

〈短横下一状态〉::=
〈短横〉

〈短横〉 ::=

包含在一个〈启动〉中的〈跃迁〉，一定不能直接地或间接地引导到〈短横下一状态〉。

模型

在〈状态〉的每一〈下一状态〉中，〈短横下一状态〉将由此〈状态〉的〈状态名〉所取代。

4.10 服务

在基本 SDL 中，一个进程的行为用一个进程图形来定义。服务概念通过一组服务定义为进程图形提供另一种选择。在多数情况下，服务定义可以降低整体的复杂性和增加进程定义的可读性。此外，每一个服务定义可以定义进程的部分行为，这在某些应用中是有用的。

4.10.1 服务分解

具体正文文法

〈服务分解〉 ::=
{〈服务信号路由定义〉
| 〈信号路由连接〉
| 〈服务定义〉
| 〈选择定义〉
| 〈正文服务引用〉}+
〈服务信号路由定义〉 ::=
SIGNALROUTE 〈服务信号路由名〉
〈服务信号路由路径〉
[〈服务信号路由路径〉]
〈服务信号路由路径〉 ::=
{FROM 〈服务标识符〉 TO 〈服务标识符〉
| FROM 〈服务标识符〉 TO ENV
| FROM ENV TO 〈服务标识符〉}
WITH 〈信号表〉〈结束〉
〈信号路由连接〉 ::=
CONNECT 〈信号路由标识符〉
AND 〈服务信号路由标识符〉{, 〈服务信号路由标识符〉}*〈结束〉
〈正文服务引用〉 ::=
SERVICE 〈服务名〉 REFERENCED 〈结束〉

当一〈进程定义〉包含〈服务分解〉时，它必须不含有在〈服务分解〉之外的〈定时器定义〉。

一个〈服务分解〉必须至少含有一个〈服务定义〉。

如同适用于〈信号路由〉一样，类似的良形式规则也适用于〈服务信号路由〉。

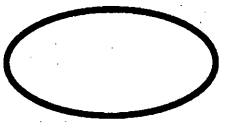
具体图形文法

〈服务相互作用区〉::=
{〈服务区〉|〈服务信号路由定义区〉}+

〈服务区〉::=
〈图形服务引用〉
|〈服务图〉

〈图形服务引用〉::=
〈服务符〉**contains**〈服务名〉

〈服务符〉::=



〈服务信号路由定义区〉::=
〈信号路由符〉
is associated with {〈服务信号路由名〉
[〈信号路由标识符〉]
〈信号表区〉
[〈信号表区〉]} **set**
is connected to {〈服务区〉
{〈服务区〉|〈框架符〉}} **set**

当〈信号路由符〉连接到〈框架符〉时，〈信号路由标识符〉标志了此信号路由连接到其上的一个外部信号路由。

语义

〈服务分解〉可用来替换〈进程体〉，并表达相同的行为。

模型

通过将〈服务分解〉变换成基本概念来构造服务概念的模型。变换〈服务信号路由定义〉和〈信号路由连接〉的结果是什么也没有。

4.10.2 服务定义

具体正文文法

〈服务定义〉::=
SERVICE {〈服务名〉|〈服务标识符〉}〈结束〉
[〈有效输入信号集〉]
{〈变量定义〉
|〈数据定义〉
|〈定时器定义〉
|〈视见定义〉
|〈入口定义〉
|〈选择定义〉
|〈宏定义〉
|〈过程定义〉
|〈正文过程引用〉}*
〈服务体〉
ENDSERVICE [{〈服务名〉|〈服务标识符〉}]〈结束〉

〈服务体〉::=
〈进程体〉

〈优先输入〉::=
PRIORITY INPUT〈优先输入表〉〈结束〉〈跃迁〉

〈优先输入表〉::=
〈优先激励〉{,〈优先激励〉}*

〈优先激励〉::=
〈优先信号标识符〉[([〈变量标识符〉] {, [〈变量标识符〉]}*)]

〈优先输出〉::=
PRIORITY OUTPUT〈优先输出体〉

〈优先输出体〉::=
〈优先信号标识符〉[〈实在参数〉]
{,〈优先信号标识符〉[〈实在参数〉]}*

一个信号是在一个进程中的高优先级信号，当且仅当此信号在该进程的〈服务定义〉的〈优先输入〉中被表述过。

〈服务定义〉中的〈变量定义〉不应含有关键字 EXPORTED 或 REVEALED。

〈优先输出〉中的〈优先信号标识符〉不应包含在〈输入部分〉或〈保存部分〉之中。〈优先输入〉中的〈优先信号标识符〉不应包含在〈输出〉之中。

在2.5.2中陈述的关于进程的合法输入信号集和服务信号路由的规则同样适用于服务。

仅当包围的〈功能块定义〉含有一些〈信号路由定义〉时，〈服务分解〉才可以含有一些〈服务信号路由定义〉。

在一个〈服务分解〉中，只允许一个〈服务定义〉有一个〈启动〉包含〈跃迁串〉。所有其它的〈启动〉必须仅含有〈下一状态〉。

在一个〈进程定义〉中，每个〈服务定义〉有一个完整合法输入信号集（这个集合是该〈服务定义〉的〈合法输入信号集〉和在进来〈服务信号路由〉上传递的信号集的并集）。这些

完整合法输入信号集必须是不相交的。

当包围的〈进程定义〉含有一个〈服务定义〉时，〈过程定义〉不允许有〈状态〉。多个服务可以视见的〈过程定义〉不应包含有 **VIA** 构件。

在一个〈服务分解〉的各种〈服务定义〉中，关联于各个〈连续信号〉的优先权的集合不能重叠。

适用于〈信道至路由连接〉的良形式规则也适用于〈信号路由连接〉。

如果包围的〈服务分解〉含有任何〈服务信号路由定义〉，则对于〈输出〉中的每一个〈信号路由标识符〉必须存在这样一条服务信号路由，它源于包围的服务，被连接到信号路由，并能够传递包含在〈输出〉中的〈信号标识符〉所表示的信号。

如果〈输出〉不含有 VIA 构件，则必须至少存在一条通信路径（或对于拥有的服务是隐含的，或经过（可能是隐式的）服务信号路由及可能的信号路由和信道），它从此服务出发，能够传递在〈输出〉中的〈信号标识符〉所指的信号。

对每一个〈优先输出〉，必须至少存在一个通信路径（或对拥有的服务是隐含的，或经过（可能是隐式的）服务信号路由），它从该服务出发，能够传递〈优先输出〉中的〈优先信号标识符〉所表示的信号。

仅在〈服务体〉中允许〈优先输入〉。仅在〈服务体〉和〈过程体〉中允许〈优先输出〉。

具体图形文法

〈服务图〉::=
〈框架符〉 **contains**
{〈服务标题〉
{ {〈服务正文区〉}*
{〈图形过程引用〉}*
{〈过程图〉}*
{〈宏图〉}*
〈服务图形区〉} **set**}

〈服务标题〉::=
SERVICE {〈服务名〉 | 〈服务标识符〉}

〈服务正文区〉::=
〈正文符〉 **contains**
{〈变量定义〉
| 〈数据定义〉
| 〈定时器定义〉
| 〈视见定义〉
| 〈进口定义〉
| 〈选择定义〉
| 〈宏定义〉}*

〈服务图形区〉::=
〈进程图形区〉

〈优先输入关联区〉::=

〈实线关联区〉 **is connected to** 〈优先输入区〉

〈优先输入区〉 ::=

〈优先输入符〉 **contains** 〈优先输入表〉

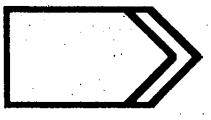
is followed by 〈跃迁区〉

〈优先输入符〉 ::=

〈优先输出区〉 ::=

〈优先输出符〉 **contains** 〈优先输出体〉

〈优先输出符〉 ::=



语义

根据下述要求来导出服务的特性：即取代〈进程体〉的〈服务分解〉所表达的行为应和〈进程体〉的行为相同。

在一个进程实例中，对在〈进程定义〉中的每一个〈服务定义〉有一个服务实例。服务实例是进程实例的分量，它不能够作为单独的事物来处理（创建、访问或撤消）。它们共享输入端口和此进程实例的表达式 SELF、PARENT、OFFSPRING 及 SENDER。

一个服务实例是一个有限状态自动机，但它不能与此进程实例的其它服务实例并行运行，也就是说，在一个进程实例中，任何时候仅有一个服务实例可以执行一个跃迁。

在〈优先输出体〉中，隐含有构件 TO SELF。优先信号是比普通信号具有较高优先权的一种特殊类别的信号。这些信号仅可以在同一进程实例中的服务实例之间传递。

来自输入端口的输入信号将交给能够接收此信号的服务实例。

模型

a) 定义的变换

通过用一个新名字代替服务中一个名字的每一次出现，便可将一个〈服务定义〉中的局部定义变换成为进程级的局部定义。限定符中每一个对服务的引用将消失。

包含相同的视见变量或进口变量的多个视见定义或进口定义被合并成为一个视见定义或进口定义。

b) 〈服务体〉的变换

多个〈服务体〉的集合可以被变换成为一个〈进程体〉。用几种办法可以做到这一点。这里选用了一个简单的变换方式，因为主要的目的是通过严格的具体语法来定义服务概念。出于实际的原因，一个〈服务体〉和一个〈进程体〉被认为是一个由状态、状态间的跃迁串、终止跃迁串和一个启动跃迁串构成的图形。一个跃迁串唯一地由一个启动状态、一个输入和一个结束状态来定义。

1) 状态

变换得到的进程图形中的状态由命名元组来识别。此元组的维数就是服务图形数。每一个元组分量唯一地归属于原来的服务图形之一，并且，此元组分量的值就是对应服务图形的状态名字。进程图形的状态的名字将是元组的集合。采用上述规则就可以构成这个元组集合。例如：

给出两个服务图形和它们的状态

f1：〈a〉〈b〉

f2：〈A〉〈B〉〈C〉

则结果进程图形具有下面的状态

〈a. A〉〈a. B〉〈a. C〉〈b. A〉〈b. B〉〈b. C〉

这种状态数的增加很快，是爆炸性的。通常可以相当明显地减少它的增加，但这里不讨论这个问题。

2) 跃迁串

服务图形中的每一个跃迁串在一个或多个地方被拷贝到进程图形中，其目的是用来连接满足以下条件的每一对状态元组：

— 启动状态元组的一个分量代表跃迁串的启动状态

— 结束状态元组的一个分量代表跃迁串的结束状态

— 这两个状态元组的其它分量值必须是相同的。

举例：

在前面的例子中，在f2服务图中有一跃迁串在状态〈B〉和〈C〉之间。在所得到的进程图形中，此跃迁串将把〈a. B〉连接到〈a. C〉、又把〈b. B〉连接到〈b. C〉。这可以更紧凑地表示为（利用具体语法的简化表示法）：

〈 * . B〉变换成〈-. C〉

3) 启动跃迁串

如果有一个服务图形含有一个启动跃迁串，则此跃迁串被变换成进程图形的启动跃迁串。进程图形的启动跃迁串导致一个状态元组，它把此服务图形的所有的初始状态名字用作为它的分量。

4) 停止跃迁串

每一个导致〈停止〉的跃迁被拷贝到进程图形，并且，它被连接到每一个状态元组。

这些状态元组各有一个分量对应于此跃迁的启动状态。

5) 优先信号

对优先信号的变换如下：
结果进程图形的每一个状态被分成两个状态。到起始状态的优先输入连接到第一个状态，所有其它输入连接到第二个状态，并保存在第一个状态中。然后，导向起始状态的跃迁串被引导到第一个状态。下面的动作串将被加入到这个跃迁串：

— 产生一个唯一的标志值，并将该值赋给 SAME_TOKEN 这个隐含变量
— 携带此标志值的隐含信号 X_CONT 被传递到 SELF。
一个隐含信号 X_CONT 的输入被加到第一个状态，后随下面的跃迁串：
将接收到的标志值与 SAME_TOKEN 的值用一个判定来比较。
如果两个值相等，则选择导向第二状态的一条路径，否则，选择回到第一个状态的路径。

举例

下面给出含有一个〈服务分解〉的〈进程定义〉的一个例子，还给出对应的〈服务定义〉。这个进程的行为与在§2.9节的图2.9.9中给出的例子相同。

```
PROCESS Game;
    FPAR Player pid;
    SIGNAL Proberers (integer);
    DCL A integer;

    SIGNALROUTE IR1 FROM Game_handler TO ENV WITH Score, Gameid;
    SIGNALROUTE IR2 FROM Game_handler TO ENV WITH Subscr, Endsubscr;
    SIGNALROUTE IR3 FROM ENV TO Game_handler WITH Result, Endgame;
    SIGNALROUTE IR4 FROM ENV TO Bump_handler WITH Probe;
    SIGNALROUTE IR5 FROM ENV TO Bump_handler WITH Bump;
    SIGNALROUTE IR6 FROM Bump_handler TO ENV WITH Lose, Win;
    SIGNALROUTE IR7 FROM Bump_handler TO Game_handler WITH Proberers;

    CONNECT R5 AND IR5;
    CONNECT R2 AND IR3, IR4;
    CONNECT R3 AND IR1, IR6;
    CONNECT R4 AND IR2;

    SERVICE Game_handler REFERENCED;
    SERVICE Bump_handler REFERENCED;

ENDPROCESS Game;

SERVICE Game_handler;

    /* 此服务管理一个游戏。
    它要进行各种动作
    如启动游戏、结束游戏、跟踪得分和通知得分 */
```

```
    DCL Count integer;
    /* 用来跟踪得分的计数器 */

    START;
        OUTPUT Subscr;
        OUTPUT Gameid TO Player;
        TASK Count:=0;
        NEXTSTATE STARTED;
    STATE STARTED;
        PRIORITY INPUT Proberers (A);
            TASK Count:=Count+A;
            NEXTSTATE-;
        INPUT Result;
            OUTPUT Score (Count) TO Player;
            NEXTSTATE-;
        INPUT Endgame;
            OUTPUT Endsubscr;
            STOP;
        ENDSTATE STARTED;
ENDSERVICE Game_handler;

SERVICE Bump_handler;

    /* 此服务要进行下列动作：登记冲撞 Bump 的次数，处理游戏者的探测 Probe，把探测结
    果通知游戏者和通知 Game_handler */

    START;
        NEXTSTATE EVEN;
    STATE EVEN;
        INPUT Probe;
            OUTPUT Lose TO Player;
            PRIORITY OUTPUT Proberers (-1);
            NEXTSTATE-;
        INPUT Bump;
            NEXTSTATE ODD;
    ENDSTATE EVEN;
    STATE ODD;
        INPUT Bump;
            NEXTSTATE EVEN;
        INPUT Probe;
            OUTPUT Win TO Player;
            PRIORITY OUTPUT Proberers (+1);
            NEXTSTATE-;
    ENDSTATE ODD;
```

ENDSERVICE Bump_handler;

下图所示与 SDL/GR 的例子相同。

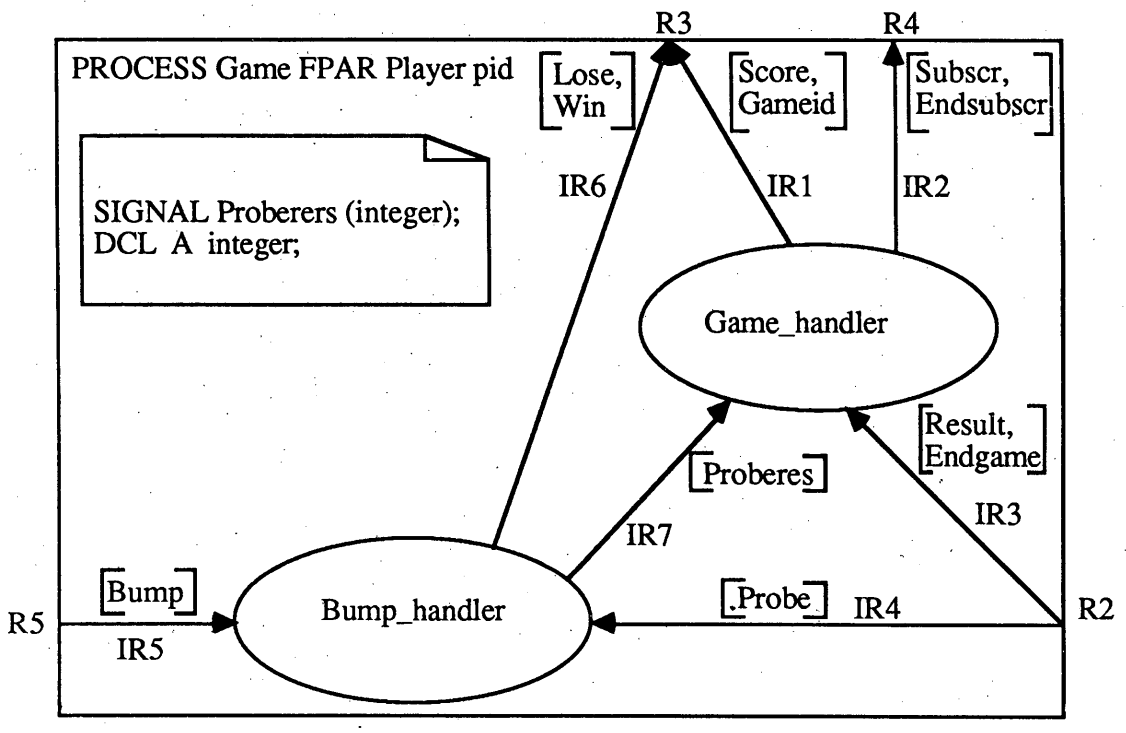


图 4.10.1

具有服务分解的一个进程图举例

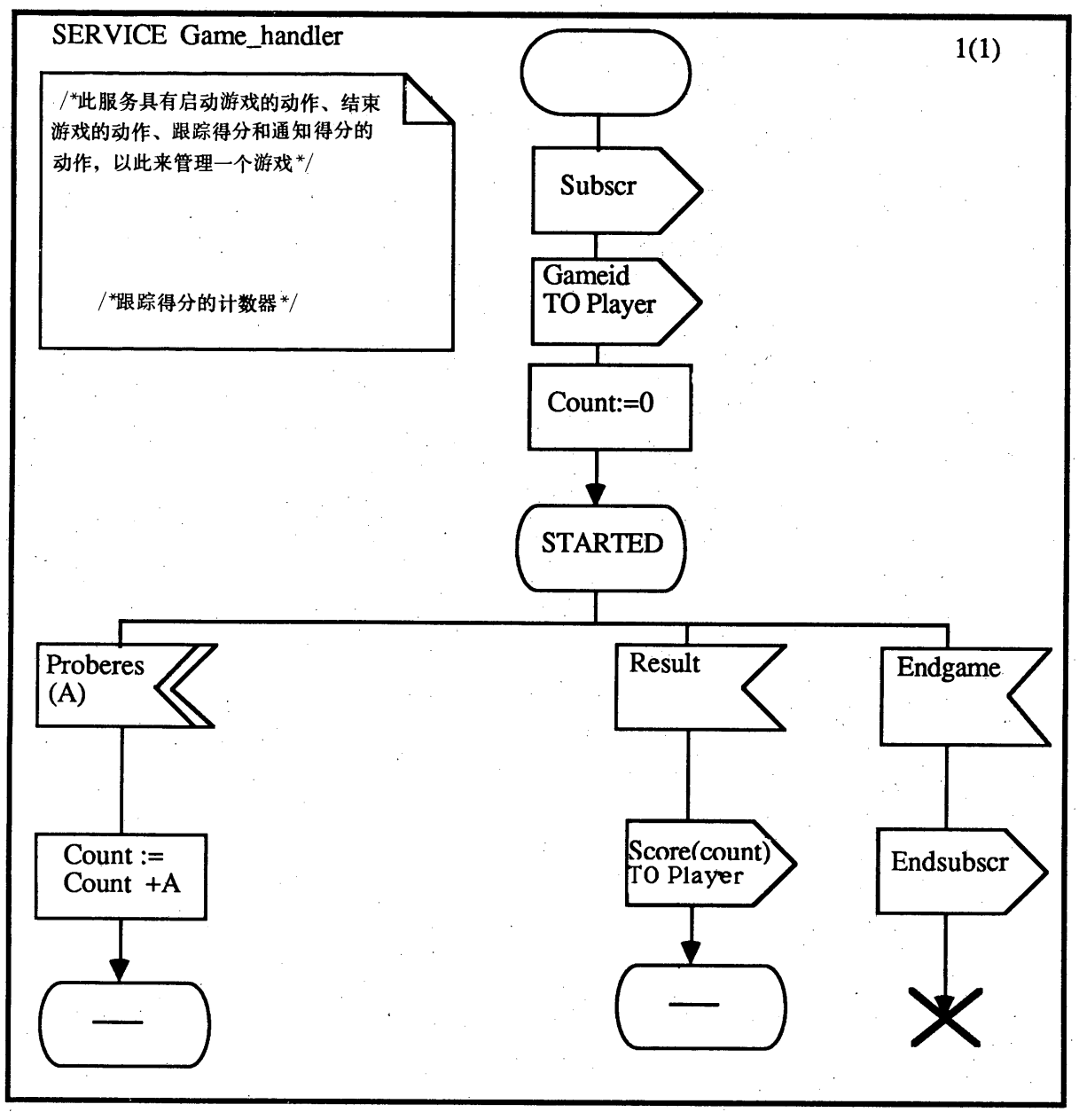


图 4.10.2

一个服务图举例

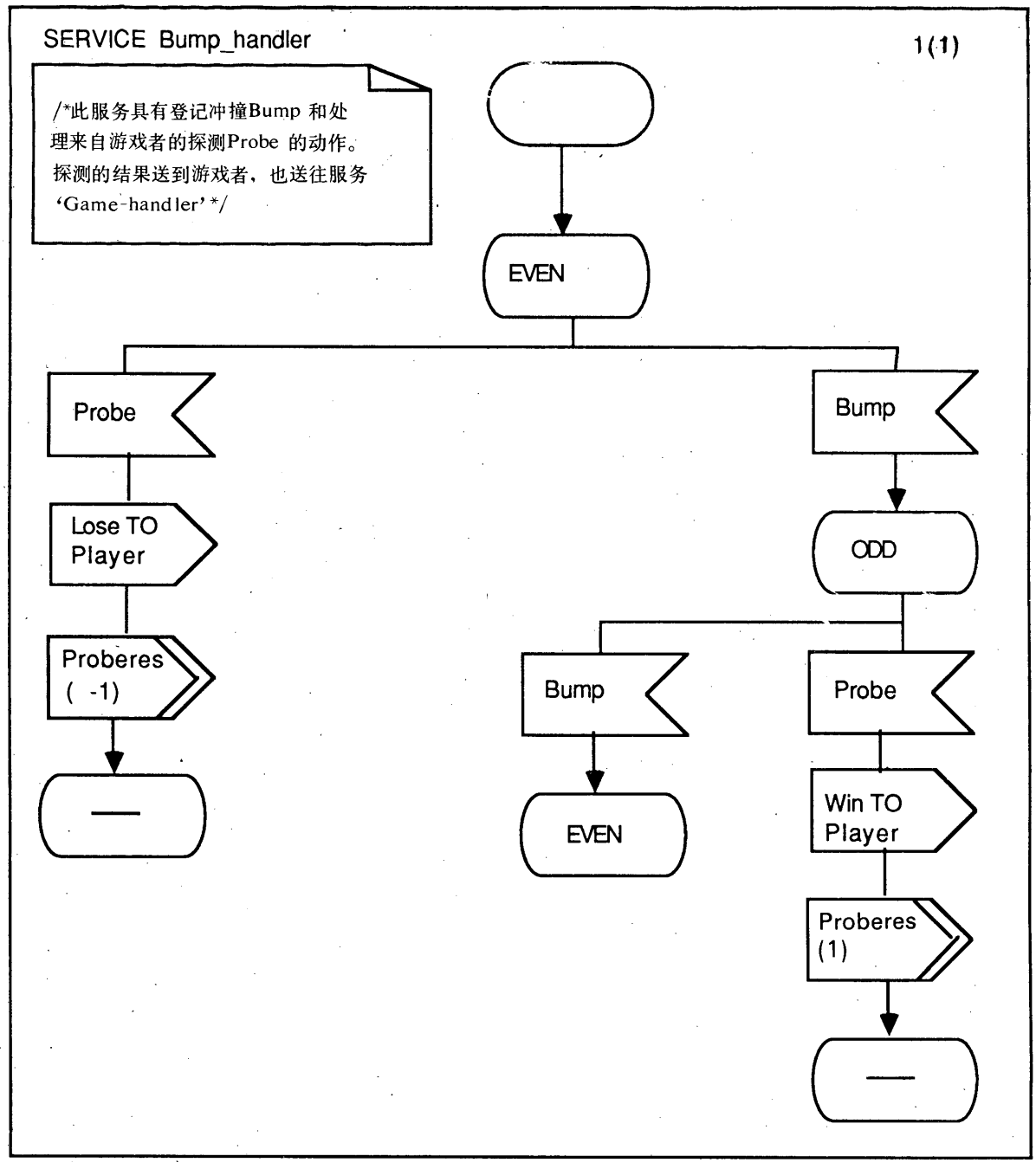


图 4.10.3
一个服务图举例

应用变换规则1—4，得到图4.10.4的进程图形；它仍含有还没有变换的优先信号。这些含有优先信号的跃迁，可以方便地予以简化，再利用星号状态概念，可以得到与§2.9节中图2.9.10的相同的进程（注意状态 EVEN 和 ODD 分别对应于状态 STARTED.EVEN 和 STARTED.ODD）。

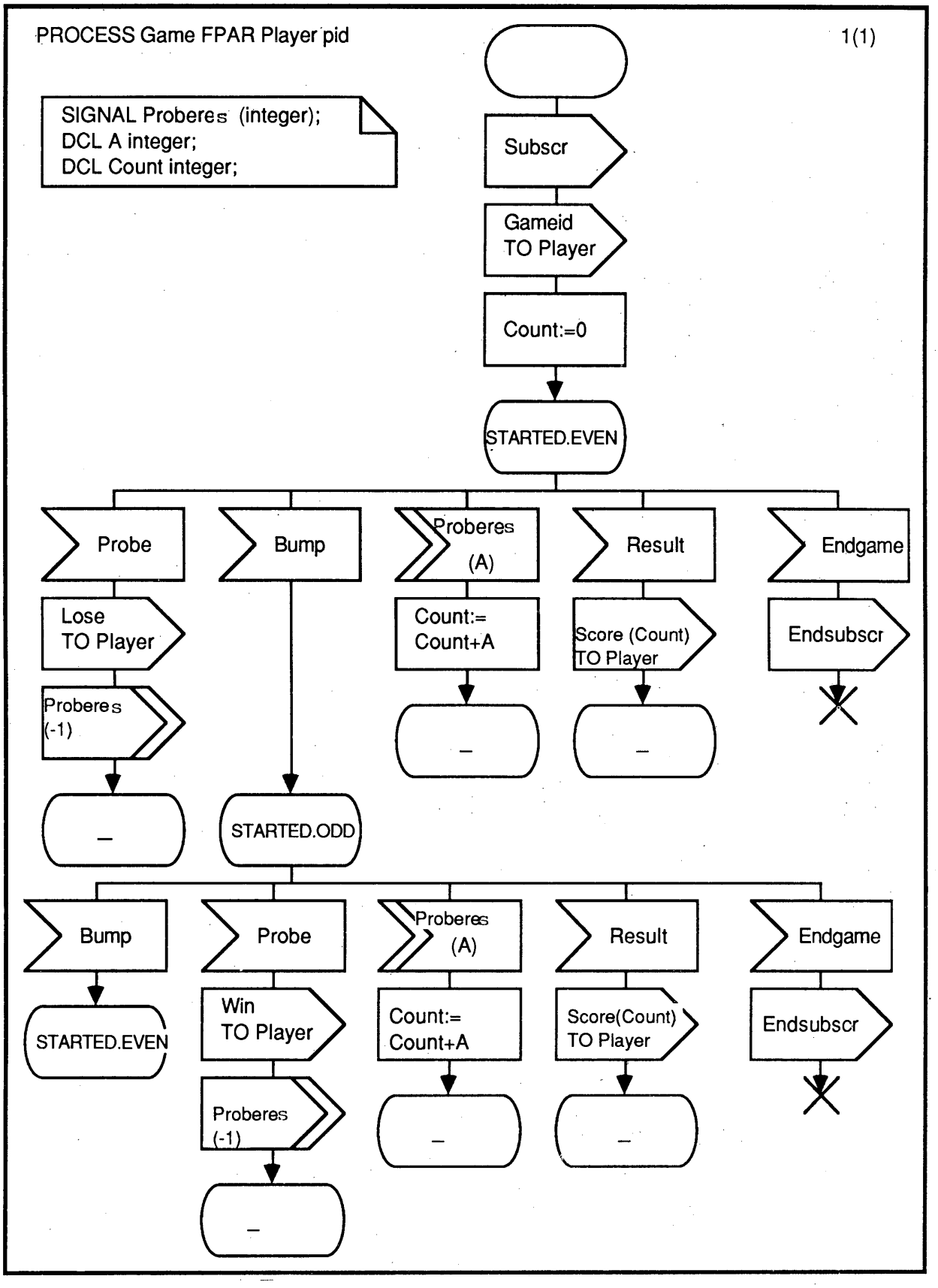


图 4.10.4

部分变换举例

4.11 连续信号

在用SDL描述系统时，可能出现这样的情况：用户喜欢指出某一个跃迁是直接由一布尔表达式的“真”(True)值引起的跃迁。实现这个目的的方法是在该状态中求表达式的值，如果求出的值为“真”，则启动此跃迁。这种方法简称为连续信号，它使得在某个条件满足时，就直接启动一个跃迁。

具体正文文法

〈连续信号〉::=
PROVIDED〈布尔表达式〉〈结束〉
[PRIORITY〈整数字面值名〉〈结束〉]〈跃迁〉

〈状态〉的〈连续信号〉中的〈整数字面值名〉的值必须是互不相同的。仅当〈状态〉含有一个且仅有一个〈连续信号〉时，PRIORITY构件才可以省略掉。

具体图形文法

〈连续信号关联区〉::=
〈实线关联符〉**is connected to**〈连续信号区〉

〈连续信号区〉::=
〈允许条件符〉
contains {〈布尔表达式〉[[〈结束〉] PRIORITY〈整数字面值名〉]}
is followed by〈跃迁区〉

语义

〈连续信号〉中的〈布尔表达式〉在进入它所关联的状态时就求值，并且当在此状态中等待时，只要在输入端口上所附着的〈输入表〉没有〈激励〉就要求表达式的值。如果〈布尔表达式〉的值为“真”，就启动跃迁。如果在多个〈连续信号〉中，〈布尔表达式〉的值同时为“真”，则应启动具有最高优先权的跃迁。即启动〈整数字面值名〉的值是最小的〈连续信号〉所确定的跃迁。

模型

设含有〈连续信号〉的状态其状态名为state-name。此状态的变换如下。此变换需要两个隐式变量n和newn。变量n被初始化为0。此外，需要一个传递整数值的隐式信号emptyQ。

1) 所有提及此state-name的〈下一状态〉由JOIN1取代；
2) 插入下面的跃迁：

```
1:   TASK  n:=n+1;
     OUTPUT emptyQ (n) TO SELF;
     NEXTSTATE state-name;
```

3) 下面的〈输入部分〉被加到〈状态〉state-name：

```
     INPUT emptyQ (newn);
```

还有一个〈判定〉含有下面的〈问题〉

```
(newn=n)
```

4) “假”〈回答部分〉含有

NEXTSTATE state-name;

4b) “真”〈回答部分〉含有一系列的〈判定〉对应于以优先次序排队的〈连续信号〉。〈整数字面值名〉的值越低表明优先级越高。

“假”〈回答部分〉含有下一个〈判定〉，最后一个〈判定〉是例外，这时“假”〈回答部分〉含有汇接 JOIN1；

这些〈判定〉的每一个“真”〈回答部分〉导致对应的〈连续信号〉的〈跃迁〉。

举例

参见§4.12节。

4.12 允许条件

在 SDL 中，如果在一个状态中接收到一个信号，就立即启动一个跃迁。采用允许条件的概念使我们能够为启动一个跃迁另外加上一个条件。

具体正文文法

〈允许条件〉::=

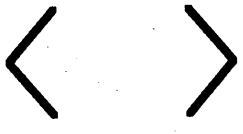
PROVIDED〈布尔表达式〉〈结束〉

具体图形文法

〈允许条件区〉::=

〈允许条件符〉**contains**〈布尔表达式〉

〈允许条件符〉::=



语义

〈允许条件〉中的〈布尔表达式〉在进入所讨论的状态之前就已求值，并且在每一次由于到达了一个〈激励〉而再次进入此状态时，都要再求值。在多重允许条件的情况下，将在进入此状态之前以任意顺序对这些允许条件求值。此变换模式通过发送额外的〈激励〉给输入端口，可以保证重复地对表达式求值。在〈输入表〉中指明的一个信号如果先于〈允许条件〉到达，则只有在对应的〈布尔表达式〉之值为“真”时，才可以启动跃迁，如果此值为“假”，则信号被保存起来。

模型

名字为 state_name 的含有〈允许条件〉的状态变换如下。此变换需要两个隐式变量 n 和 newn。变量 n 被初始化为0。此外，需要一个传递整数值的隐式信号 emptyQ。

1) 所有提及此 state_name 的〈下一状态〉用 JOIN1来取代；

2) 插入下面的跃迁：

1: TASK n:=n+1;

OUTPUT emptyQ (n) TO SELF;

对应于附于此状态的〈允许条件〉的每一个〈布尔表达式〉有一个对应的判定。这些判定以不确定的次序分层地添加，以便对于所有附在此状态上的允许条件，可以求得“真”值的所有组合。

每一个这样的组合导致一个新的不同状态。

3) 每一个这样的新状态具有一组〈输入部分〉，它包括不带允许条件的状态的〈输入部分〉的拷贝，还包括状态的〈允许条件〉的〈布尔表达式〉之值为“真”的〈输入部分〉。

至于剩下〈输入部分〉的〈激励〉构成一个〈保存表〉，用作为附在此状态上的一个新的〈保存部分〉。原先的状态的〈保存部分〉也拷贝给这个新状态。

4) 对每一个新状态要增加：

INPUT emptyQ (newn);

一个含有〈问题〉的〈判定〉(newn=n);

“假”〈回答部分〉含有一个〈下一状态〉，返回到这同一个新状态。

5) “真”〈回答部分〉含有一个汇接 JOIN 1

6) 如果在同一个〈状态〉中使用〈连续信号〉和〈允许条件〉，则应对〈连续信号〉中的〈布尔表达式〉求值。具体作法是用〈连续信号〉模型的步骤4b 来取代〈允许条件〉模型的步骤5。

举例

下面给出一个例子说明出现在一个状态中的连续信号和允许条件的变换。

注意在此例子中，为方便起见引入了连接符 ec。它不是变换模型的一部分。

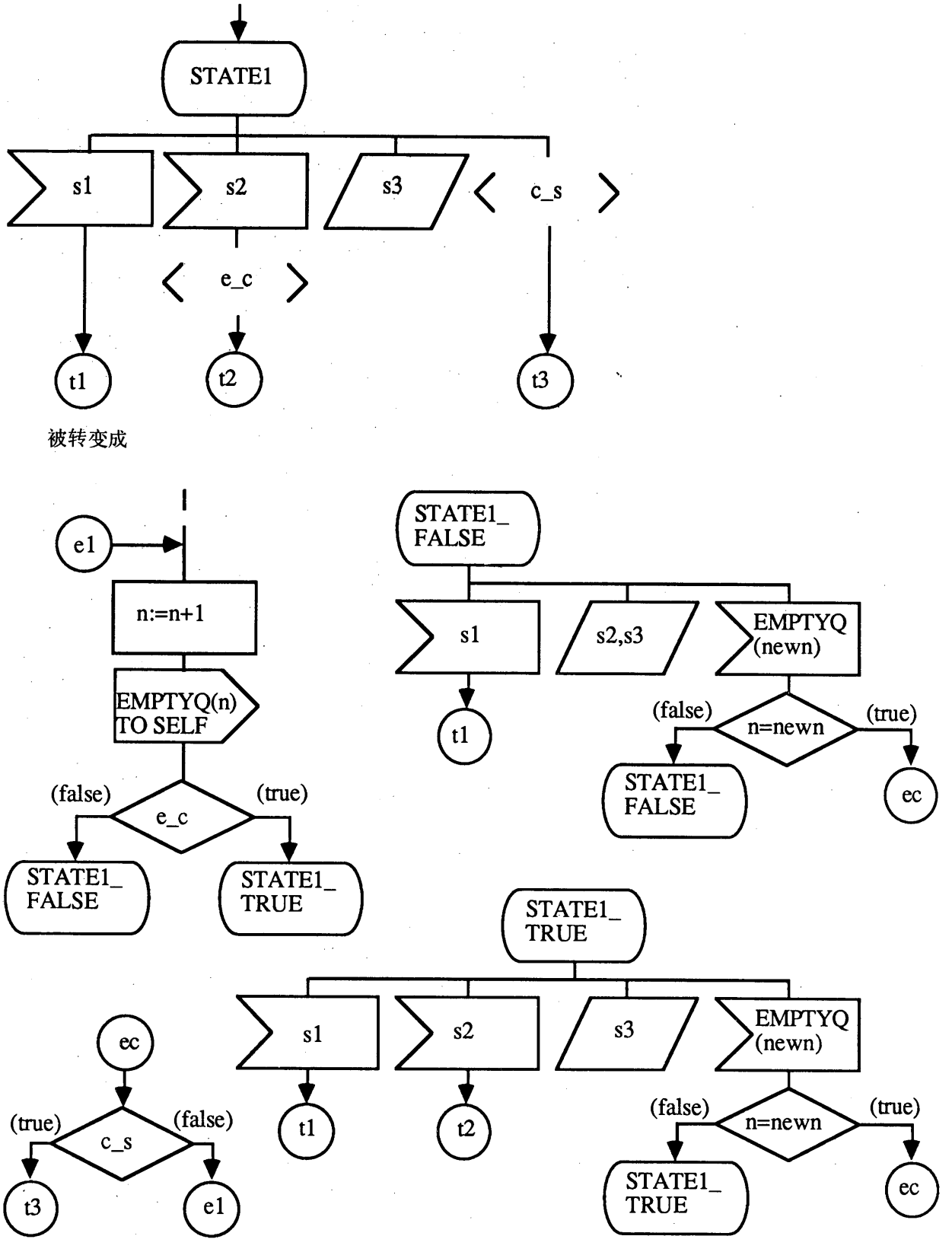


图 4.12.1

同一状态中连续信号和允许条件的变换

4.13 进口值和出口值

在SDL中，一个变量总是为一个进程实例所拥有，并局限于此进程实例，通常此变量仅对于拥有它的进程实例是可视见的。然而也可以声明一个变量具有共享值（参见§2节），这样就允许在同一个功能块中的其它进程实例对此变量的值进行访问。如果另一个功能块中的进程实例需要访问此变量的值，则需要与拥有此变量的进程实例交换信号。

这可以利用下面叫做进口值和出口值的简化表示法来实现。此简化表示法也可以把值出口给同一个功能块中的其它进程实例，在这种情况下，本来可以采用共享值，而现在又提供了另一种办法。

具体正文文法

〈进口定义〉::=
IMPORTED 〈进口名〉{,〈进口名〉}*〈类别〉
{,〈进口名〉{,〈进口名〉}*〈类别〉}*〈结束〉
〈进口表达式〉::=
IMPORT (〈进口标识符〉[,〈PId 表达式〉])
〈出口〉::=
EXPORT (〈变量标识符〉{,〈变量标识符〉}*)

具体图形文法

〈出口区〉::=
〈任务符〉**contains**〈出口〉

语义

一个进程实例叫做一个变量的出口者，如果它拥有此变量，其值被出口到其它进程实例。使用这些值的其它进程实例叫做此变量的进口者。此变量就叫做出口变量。

一个进程实例可以既是进口者，又是出口者，但它不能够从环境进口或出口给环境。

a) 出口操作

出口变量在其〈变量定义〉中有关键字EXPORTED，并且具有一个隐含的拷贝将被用于进口操作中。

出口操作是一次〈出口〉的执行，出口者通过它交出一个出口变量的当前值。一次出口操作把出口变量的当前值存储到其隐含拷贝中去。

b) 进口操作

对于进口者中的每一个〈进口定义〉，有一组隐式变量，它们都具有在〈进口定义〉中

给定的名字和类别。这些隐式变量用来存储进口的值。

一个进口操作要执行一个〈进口表达式〉，进口者通过这种操作来访问一个被出口变量的值。此值存储在一个隐式变量中，就是由〈进口表达式〉中的〈进口标识符〉所表示的那个隐式变量。包含此出口变量的出口者由〈进口表达式〉中的〈PId 表达式〉来确定。如果没有规定〈PId 表达式〉，则应仅有一个实例是出口那个变量的。出口者中的出口变量和进口者中的隐式变量之间的联系由在〈出口〉中和在〈进口表达式〉中相同的〈标识符〉来确定。此外，此出口变量和该隐变量必须具有同样的类别。

模型

可以通过信号的交换来构造进口操作的模型。这些信号是隐式的，它们在隐式信道和信号路由上传递。进口者发送一个信号给出口者，并等待回答。为了对此信号作出响应，出口者向进口者发回一个信号，信号中带有在出口变量之隐式拷贝中所含有的值。

如果一个缺省赋值附属于此出口变量，或者，如果此出口变量在被定义时已被初始化，则隐式拷贝也被初始化，其值与出口变量的值相同。

对于包含在一个系统定义中的所有〈进口定义〉中的〈进口名〉和〈类别〉的每一个组合，有两个隐式〈信号定义〉。这两个〈信号定义〉中的〈信号名〉分别由 *xtQUERY* 和 *xtREPLY* 表示，这里 *x* 表示一个〈进口名〉、*t* 表示一个〈类别〉。出口变量的隐式拷贝由 *imcx* 表示。

a) 进口者

〈进口表达式〉“IMPORT (x, pidexp)”被变换成下面的内容：

OUTPUT xtQUERY TO pidexp;

在状态 xtWAIT 等待，保存所有其他信号；

INPUT xtREPLY (x);

用 x（隐变量的〈名字〉）取代此〈进口表达式〉；

如果在〈表达式〉中，一个〈进口表达式〉出现不止一次，则用于每一次出现的是具有同样〈名字〉的一个不同的隐式变量。

b) 出口者

对于出口者的所有〈状态〉、包括隐式状态，都添加了下面的〈输入部分〉：

INPUT xtQUERY;

OUTPUT xtREPLY (imcx) TO SENDER;

/ * 下一状态相同 * /

〈出口〉“EXPORT (x)”被变换成下面内容：

TASK imcx:=x;

5 SDL 中的数据

5.1 引言

本引言给出了用于定义数据类型的形式模型的概要。并介绍§5节的其余部分是如何构成的。

在一个规格语言中，十分重要的是允许人们用数据类型的行为来形式地描述这些数据类型，而不是象某些程序语言那样从提供的原始类型来组合它们。后一种途径总是含有该数据类型的一种特定的实现方法，因此限制了实现者选择该数据类型的适当的表示方法的自由度。采用抽象数据类型的方法允许任何的实现办法，只要对于该规格是可行的和正确的。

5.1.1 数据类型的抽象

用于 SDL 中的所有数据都是以抽象数据类型为基础的。它们是根据数据的抽象特性来定义的，而不是根据某些具体实现方法来定义的。定义抽象数据类型的例子将在§5.6中给出，在那里定义了 SDL 语言的预定义数据功能。

尽管所有的数据类型是抽象的，并且预定义数据功能甚至可以被用户改写，但 SDL 仍试图提供一组预定义数据功能，其行为和语法方面都是人们熟悉的。下面的数据类型是预定义的：

a) 布尔型 Boolean
b) 字符 Character
c) 串 String
d) 字符串 Charstring
e) 整型 Integer
f) 自然数 Natural
g) 实数型 Real
h) 数组 Array
i) 幂集 Powerset
j) 进程标识符 PId
k) 持续时间 Duration
l) 时刻 Time

形成复合的数据类型时可以使用结构类别概念（STRUCT）。

5.1.2 构造数据体系的概要

通过基础代数来构造数据的模型。此代数规定了一些类别和一组运算符，用来把这些类别连系起来。每一个类别是由相关的运算符集合所能够产生的所有的值的集合。在此语言中，每一个值至少可以由仅含有字面值和运算符的一个表达式来表示（除了在 PId 值的特殊情况下）。字面值是运算符不带有变元的一种特殊情况。

类别和运算符以及数据类型的行为（由代数规则来规定）一起构成了数据类型的特性。一个数据类型由若干个部分类型定义构成，每一个部分类型用来定义一个类别和一些运算符，以及与此类别有关的一些代数规则。

用关键词 NEWTYPE 来引入一个部分类型定义，用来定义一个不同的新类别。可以创建一个类别，其特性从其它类别继承得来，但它具有不同的类别标识符和不同的运算符标识符。

可以引入一个同义类型来指定一个已经存在的类别的值的一个子集。

一个生成程序是一个不完整的 NEWTYPE 描述：在它成为一个类别之前，必须提供所缺少的信息，使它成为一个实例。

某些运算符作用于一个类别，从而产生此类别的（可能是新的）值。另外一些运算符通过映射到其它已定义的类别，从而给予原来的类别以含义。很多运算符从其它类别变换到布尔类别，但是不允许这些运算符扩展布尔类别，这是严格禁止的。

在 SDL 中，一个函数被认为是一种被动运算符，它对作为参数给出的变量的值没有影响。SDL 还定义了赋值语句，用它可以改变变量的值。

5.1.3 术语

在§5节或数据模型中采用的术语是经过选择的。使之与基础代数方面已公布的著作相一致。具体地说，用“数据类型”这个术语来表示一些类别和关联于这些类别的运算符的集合以及表示用代数方程来规定的这些类别和运算符的特性。一个“类别”是一组具有共同特性的值。一个“运算符”是类别间的一种关系。一个“等式”是有关一个类别中项与项之间相等的一个定义。一个“值”是相等的项的一个集合。一个“公理”是一个等式，它定义了一个布尔值为真。“公理”也被用来作为代表“公理”或“等式”的一个术语，并且，一个“等式”也可以是一个“公理”。

5.1.4 关于数据正文的划分

用于构造 SDL 数据的基础代数模型用这样的方式来描述，使得大多数的数据概念可以用 SDL 的抽象数据语言的一些核心数据来定义。

§5节的内容被分成此引言（§5.1）、数据核语言（§5.2）、基础代数模型（§5.3）、数据的被动使用（§5.4）、数据的主动使用（§5.5）以及预定义数据（§5.6）。

数据核语言定义了 SDL 中的数据部分，它直接对应于所用的基础代数方法。

关于基础代数的正文对这个方法的数学基础给出了更为详细的介绍。这在附录 I 中以更精确的数学方式作了确切的表达。

SDL 的被动应用包括 SDL 数据的隐含和简化特征，使之可用于定义抽象数据类型。它还包括不含有赋给变量之值的表达式的解释。这些“被动”表达式对应于此语言的功能性应用。

数据的主动应用扩展了此语言，使它包括赋值语句。这包括对于变量应用的赋值和对于变量初始化的赋值。当应用 SDL 对变量赋值、或访问变量中的值时，就叫做主动地应用。主动和被动表达式之间的区别在于：被动表达式的值与它被解释的时刻是没有关系的，而主动表达式可以被解释为不同的值，这取决于变量的当前值或系统的当前状态。

最后的内容是预定义数据。

5.2 数据核语言

数据核可以用来定义抽象数据类型。

除在需要对变量赋值之概念的地方外，用来定义数据类型更为方便的构件可以用数据核中定义的构件来进行定义。(错误和同义类型的概念可以用核来定义，但在§5.4.1.7节和§5.4.1.9节中，采用了更简洁的另外的定义方法)。

5.2.1 数据类型定义

在一个SDL规格中的任一个地方，都有一个适用的数据类型定义。此数据类型定义确定了表达式的有效性和表达式之间的关系。此定义引入了一些运算符和一些值的集合（类别）。

对于数据类型定义来说，具体的和抽象的语法之间，并没有一种简单的对应关系。这是因为具体语法逐一地引入数据类型定义，其侧重点在于类别。(可参见§5.3节)。

具体语法中的有些定义常常相互依赖，并且，不能够分入不同的作用域单位。

例如

```
NEWTYPE even LITERALS0;
    OPERATORS plusee     :even,even ->even;
              plusoo     : odd, odd ->even;
    AXIOMS    plusee(a,0)     ==a;
              plusee(a,b)     ==plusee(b,a);
              plusoo(a,b)     ==plusoo(b,a);
    ENDNEWTYPE even COMMENT  ' even "number"with plus-depends on odd' ;
NEWTYPE odd LITERALS 1;
    OPERATORS plusoe     :odd,even ->odd;
              pluseo     :even,odd ->odd;
    AXIOMS    plusoe(a,0)==a;
              pluseo(a,b)==plusoe(b,a);
    ENDNEWTYPE odd;/ * odd "number" with plusdepends on even  * /
```

每一个数据类型定义都是完全的；没有引用那些不包含在此数据类型定义之中的类别和运算符。此外，一个数据类型定义一定不能够违反在最近的包围作用域单位中的数据类型定义的语义。在被包围作用域单位中的一个数据类型仅充实了在外部作用域单位中定义的类别的运算符。在某个作用域单位中定义的一个类别的值可以在较低层的作用域单位中自由地使用，也可以在较低层的作用域单位之间自由地传递。因为预定义数据是在系统层定义的，所以，预定义类别（例如布尔类别和整数类别）可以在系统的任意地方自由地使用。

抽象文法

数据类型定义 :: 类型名
| 类型并集
| 类别集
| 标记 set
| 等式集

类型并集 = 类型标识符 set

类型标识符 = 标识符

类别集 = 类别名 set

类型名 = 名字

类别名 = 名字

等式集 = 等式 set

在一个数据类型定义之中，对于每一个类别必须至少有一个标记具有与此类别一样的结果（参见§5.2.2节）。

一个数据类型定义一定不能够把新的值添加到由类型并集所标识的数据类型中的任一类别。

如果根据由数据类型定义的类型并集所标识的数据类型，一个项不等效于另一个项的话，则这些项一定不能够由数据类型定义规定为等效的。

此外，两个布尔项 True 和 False 一定不能够（直接地或间接地）定义为等效的（参见§5.4.3.1节）。对于预定义类别 PId 也不允许减少其值的个数。

注 — 对于一个类型并集，抽象语法允许多个类型标识符，以适应于基础模型所用代数的更一般的类别。在 SDL 中，仅可引用一个类型，这是因为在具体语法中，可见的数据类型是由包围的〈作用域单位类〉隐含地定义的；因此，〈类型并集〉仅在抽象语法之中被引用，它或者是周围作用域单位的类型标识符，或者是在〈系统定义〉情况中的一个空集。

具体正文文法

〈部分类型定义〉::=
NEWTYPE〈类别名〉[〈扩展特性〉]〈特性表达式〉
ENDNEWTYPE[〈类别名〉]

〈特性表达式〉::=
〈运算符〉[AXIOMS〈公理〉][〈字面值映象〉][〈缺省赋值〉]

任选的〈扩展特性〉、〈字面值映象〉以及〈缺省赋值〉项不是数据核的一部分，它们分别在§5.4.1、§5.4.1.15和§5.5.3.3节予以定义。

数据类型定义包含当前〈作用域单位类〉中所有〈部分类型定义〉的集合，也包含周围

〈作用域单位类〉的类型并集所标识的数据类型定义。一个〈数据类型定义〉的类型名是隐含的，它没有具体语法的表示形式。一个类型并集的类型标识符不言而喻可用来标识周围作用域单位的数据类型定义。

下面的一些〈作用域单位类〉(参见§2.2.2)各代表抽象语法中的一个项，它含有一个数据类型定义：〈系统定义〉、〈功能块定义〉、〈进程定义〉、〈过程定义〉、〈信道子结构定义〉或〈功能块子结构定义〉或者在图形语法中所对应的图。一个〈服务定义〉中的〈部分类型定义〉表示在包围此〈服务定义〉的〈进程定义〉中的数据类型定义的一部分（参见§4.10)。

〈作用域单位类〉中的类别用〈类别名〉的集合来表示，此〈类别名〉的集合是由该〈作用域单位类〉的〈部分类型定义〉集合引入的。

〈作用域单位类〉的标记集和等式由该〈作用域单位类〉的〈部分类型定义〉的〈特性表达式〉来表达。

〈特性表达式〉的〈运算符集〉表示抽象语法中的标记集合的一部分。完整的标记集合是由该〈作用域单位类〉中的〈部分类型定义〉所规定的标记集合的并集。

〈特性表达式〉的〈公理〉表示抽象语法中的等式集合的一部分。这些等式是由〈作用域单位类〉中的〈部分类型定义〉所确定的等式集合的并集。

预定义数据类别在系统级具有它们所隐含的〈部分类型定义〉。

如果在关键字 ENDNEWTYPE 之后给出一个〈类别名〉，则此〈类别名〉必须与在关键字 NEWTYPE 之后给出的〈类别名〉相同。

语义

数据类型定义规定了数据类型。一个数据类型具有一组类型特性，即一组类别、一组运算符和一组等式。

在具体语法中，数据类型的特性由部分类型定义来规定。部分类型定义并没有规定数据类型的所有特性，它仅部分地规定某些特性，这些特性与在部分类型定义中引入的类别有关。为了找出数据类型的完整特性，可将所有的部分类型定义综合考虑，这些定义在含有此数据类型定义之作用域单位中是适用的。

一个类别是一组数据值。两个不同的类别没有共同的值。

数据类型定义的构成包括定义当前作用域单位之作用域单位的数据类型定义，还包括在当前作用域单位中定义的类别，运算符以及等式。在系统定义中含有预定义数据类别的定义。

除在〈部分类型定义〉、〈信号具体化〉或〈服务定义〉之中以外，在任何位置适用的数据类型定义就是在最近的包围那个位置的作用域单位所定义的数据类型。在〈部分类型定义〉或〈信号具体化〉中，适用的数据类型定义分别是包围此〈部分类型定义〉或〈信号具体化〉的作用域单位的数据类型定义。在〈服务定义〉中，适用的是包围〈服务定义〉之〈进程定义〉的数据类型定义（参见§4.10)。

一个数据类型的类别的集合是在当前作用域单位中引入的类别的集合加上由此类型并集标识的数据类型的类别的集合。一个数据类型之运算符的集合是在当前作用域单位中引入的运算符的集合，加上由此类型并集标识的数据类型之运算符的集合。一个数据类型的等式的集

合是在当前作用域单位中引入的等式的集合加上由此类型并集标识的数据类型的等式的集合。

在数据类型定义中引入的每一类别具有一个标识符，它是由此作用域单位中的某个部分类型定义所引入的名字。

数据类型有标识符，它是在抽象语法中由该作用域单位的标识符所限定的唯一的类型名。在具体语法中，数据类型没有名字。

举例：

```
NEWTYPE     telephone
/* 在某个其他地方定义的一些运算符和值的构造 */
ENDNEWTYPE  telephone
```

5.2.2 字面值和参数化的运算符

抽象文法

标记	=	字面值标记 |
		运算符标记
字面值标记	::	字面值运算符名
		结果
运算符标记	::	运算符名
		变元表
		结果
变元	=	类别引用标识符$^{+}$
结果	=	类别引用标识符
类别引用标识符	=	类别标识符 |
		同义类型标识符
字面值运算符名	=	名字
运算符名	=	名字
类别标识符	=	标识符

同义类型和同义类型标识符不是核的一部分（参见§5.4.1.9）。

具体正文文法

〈运算符〉::=

[〈字面值表〉][〈运算符表〉]

〈字面值表〉::=

LITERALS〈字面值标记〉{,〈字面值标记〉}* [〈结束〉]

〈字面值标记〉::=

〈字面值 运算符名〉

| 〈扩展字面值名〉

〈运算符表〉::=

OPERATORS

〈运算符标记〉{〈结束〉〈运算符标记〉}*[〈结束〉]

〈运算符标记〉::=

〈运算符名〉:〈变元表〉—〉〈结果〉

|〈序符〉

〈运算符名〉 ::=

〈运算符名〉

|〈扩展运算符名〉

〈变元表〉::=

〈变元类别〉{,〈变元类别〉}*

〈变元类别〉::=

〈扩展类别〉

〈结果〉::=

〈扩展类别〉

〈扩展类别〉::=

〈类别〉

|〈生成程序类别〉

〈类别〉 ::=

〈类别标识符〉

|〈同义类型〉

可选用的〈扩展运算符名〉、〈扩展字面值名〉、〈序符〉、〈生成程序类别〉和〈同义类型〉不是数据核的成分。它们分别在§5.4.1、§5.4.1、§5.4.1.8、§5.4.1.12.1、§5.4.1.12.1和§5.4.1.9节中予以定义。

字面值由列在关键字LITERALS后面的〈字面值标记〉引入。一个字面值标记的结果是由定义该字面值的〈部分类型定义〉引入的类别。

在关键字OPERATORS之后的〈运算符标记〉表中的每一个〈运算符标记〉代表一个有运算符名、变元表和结果的运算符标记。

〈运算符名〉对应于抽象语法中的运算符名，尽管该名字在具体语法之中可能不是唯一的，但在作出定义的作用域单位中它是唯一的。

在抽象语法中各别的运算符名或字面值运算符名来源于

a) 〈运算符名〉(或〈字面值 运算符名〉)，加上

b) 变元类别标识符表，加上

c) 结果类别标识符，加上

d) 部分类型定义的类别标识符，在此定义中规定了〈运算符名〉(或〈字面值 运算符名〉)。

无论何时，只要确定了一个〈运算符标识符〉，则将以同样的方式得出运算符标识符中的

唯一的运算符名，同时还从上下文得出变元类别表和结果类别。具有同一〈名字〉的两个运算符如果有一个或多个变元不同或是结果类别不同，则这两个运算符的名字不同。

在〈变元表〉中，每一个〈变元类别〉表示变元表中的类别引用标识符。〈结果〉表示结果的类别引用标识符。

每当〈运算符标识符〉(或〈字面值 运算符标识符〉)的〈限定符〉含有带有关键字 TYPE 的〈路径项〉时，则在此关键字之后的〈类别名〉并不构成该运算符标识符(或字面值运算符标识符)的限定符的一部分，而是被用来导出此标识符的唯一的名字。在这种情况下，该限定符将用位于关键字 TYPE 之前的〈路径项〉表来构成。

语义

一个运算符是“完全的”(“Total”)，意味着当把此运算符作用于具有那些变元类别的值的任意一个值表时，所得到的值是结果类别的一个值。

一个运算符标记规定了在表达中如何使用此运算符。运算符标记是此运算符标识符，加变元的类别表，再加结果的类别。正是这个运算符标记决定了一个表达式在此语言中是否为一个合法的表达式，这还要根据与变元表达式的类别匹配而要用到的规则。

不带变元的运算符称为字面值。

字面值表示属于此运算符的结果类别的固定值。

一个运算符有一个结果类别，它是由该结果标识的那个类别。

注 — 作为准则：一个〈运算符标记〉应该把由包围的〈部分类型定义〉引入的类别指明为〈变元〉、或指明为〈结果〉。

举例1

```
LITERALS free, busy;
```

举例2

```
OPERATORS
    findstate:Telephone -> Availability;
    /* 查找状态：电话号码 -> 是否可用； */
```

举例3

```
LITERALS empty_list
OPERATORS  add_to_list: list_of_telephones, telephone -> list_of_telephones;
                /* 把电话号码加入表中 */
           sub_list: list_of_telephones, telephone -> list_of_telephones
                /* 把电话号码从表中删去 */
```

5.2.3 公理

公理确定哪些项代表相同的值。在一个数据类型定义中，运算符的若干变元的值和结果值

之间的关系由公理来确定，并由此赋予此运算符意义。公理或是以布尔公理给出、或是以代数等式的形式给出。

抽象文法

```
等式                    =   非量化等式|
                            量化等式|
                            条件等式|
                            非形式正文
非量化等式              ::  项
                            项
量化等式                ::  值名字 set
                            类别标识符
                            等式
值名字                  =   名字
项                      =   基本项|
                            复合项|
                            错误项
复合项                  ::  值标识符|
                            运算符标识符项+|
                            条件复合项
值标识符                =   标识符
运算符标识符            =   标识符
基本项                  ::  字面值运算符标识符|
                            运算符标识符基本项+|
                            条件基本项
字面值运算符标识符      =   标识符
```

分别在以上的复合项和基本项的规则中的可选择的条件复合项和条件基本项不是数据核的一部分，尽管包含这些项的等式可以由用核语言写出的语义上等效的等式来取代。在以上的项的规则中可选择的错误项不是数据核的一部分，它在§5.4.1.7节中定义。

非形式正文和条件等式的定义分别在§2.2.3节和§5.2.4节中给出。

在一个运算符标识符后面的项表中的每一个项(或基本项)的类别必须同此运算符标记的变元表中对应（位置上）的类别相同。

一个非量化等式中的两个项必须具有相同的类别。

具体正文文法

```
〈公理〉 ::=
    〈等式〉 { 〈结束〉〈等式〉}* [ 〈结束〉]
〈等式〉 ::=
        〈非量化等式〉
```

| 〈量化等式〉
| 〈条件等式〉
| 〈非形式正文〉

〈量化等式〉 ::=
〈量化条件〉(〈公理〉)

〈量化条件〉 ::=
FOR ALL 〈值名〉{,〈值名〉}* IN 〈扩展类别〉

〈非量化等式〉 ::=
〈项〉 == 〈项〉
| 〈布尔公理〉

〈项〉 ::=
〈基本项〉
| 〈复合项〉
| 〈错误项〉
| 〈拼字项〉

〈复合项〉 ::=
〈值标识符〉
| 〈运算符标识符〉(〈复合项表〉)
| (〈复合项〉)
| 〈扩展复合项〉

〈复合项表〉 ::=
〈复合项〉{,〈项〉}*
| 〈项〉,〈复合项表〉

〈基本项〉 ::=
〈字面值标识符〉
| 〈运算符标识符〉(〈基本项〉{,〈基本项〉}*)
| (〈基本项〉)
| 〈扩展基本项〉

〈字面值标识符〉 ::=
〈字面值 运算符标识符〉
| 〈扩展字面值标识符〉

〈非量化等式〉规则中的可选的〈布尔公理〉,〈项〉规则中的〈错误项〉和〈拼音项〉,〈复合项〉规则中的〈扩展复合项〉,〈基本项〉规则中的〈扩展基本项〉,以及〈字面值标识符〉规则中的〈扩展字面值标识符〉,它们都不是数据核的一部分。它们分别在§5.4.1.5、§5.4.1.7、§5.4.1.15、§5.4.1、§5.4.1和§5.4.1节中予以定义。

〈量化条件〉中的〈类别〉表示在量化等式中的类别标识符。〈量化条件〉中的那些〈值名〉表示在量化等式中的值名集合。

一个〈复合项表〉表示一个项表。一个运算符标识符后面跟有一个项表仅是一个复合项，如果该项表至少含有一个值标识符。

如果一个〈标识符〉是在〈项〉中出现的一个未限定名字，则它表示：

a) 一个运算符标识符，如果它处于一个开圆括号的前面（或者它是一个〈运算符名〉，则它是一个〈扩展运算符名〉——参见§5.4.1节），否则它表示

b) 一个值标识符，如果那个名字有一个定义出现在包围该〈项〉的〈量化等式〉之一个〈量化条件〉中，且在该范围内具有合适的类别的话。否则它表示

c) 一个字面值运算符标识符，如果有一个可见字面值具有那个名字，且在该范围内具有合适的类别的话。否则它表示

d) 一个值标识符，在此〈非量化等式〉的抽象语法中，它有一个隐含的量化等式。

在一个〈等式〉中，同一个无界〈值标识符〉的二次或多次出现，仅意味着一个量化条件。

一个运算符标识符从上下文中导出，使得：如果该〈运算符名〉过载（过载就是同一个〈名字〉用于不止一个运算符），则它将是用来标识一个可见运算符的那个运算符名，该可见运算符具有同样的名字，且其变元类别及结果类别与此运算符应用相一致。如果〈运算符名〉过载，则可能需要从变元导出变元的类别，从上下文导出结果类别，以便确定该运算符名字。

在一个〈非量化等式〉中，对每一个隐含地量化的值标识符，必须有一个且仅有一个类别与其应用相一致。

一定可以将每一个未限定的〈运算符标识符〉或〈字面值 运算符标识符〉与一个（且仅有一个）已定义的运算符标识符或字面值运算符标识符连系起来，后者满足使用此〈标识符〉的构件中的条件。也就是说，结合应该是唯一的。

注 — 作为准则：一个公理应该符合包围它的部分类型定义的类别，这要通过指明一个运算符或字面值具有此类别的一个结果，或者指明具有此类别的一个变元的一个运算符；一个公理应该只定义一次。

语义

每一个等式是关于一些项的代数相等性的一个语句。它指明左边项与右边项是相等的，这样，在一个项出现的地方，就可用另一个项去取代它。当一个值标识符出现在一个等式中时，则对于此值标识符在那个等式中的每一次出现，可以同时用相同的项来替代。在这种替代中，此项可以是任意一个基本项，其类别与值标识符的类别相同。

在量化等式中，值标识符由值名引入。一个值标识符被用来代表属于该量化条件的类别的任意数据值。如果把值标识符在一个等式中的每一次出现同时地用相同的值取代，则这个等式将成立，而与选来取代之值无关。

基本项是一个不含有任何值标识符的项。基本项表示一个具体的、已知的值。对于一个类别中的每一个值，至少存在一个表示该值的基本项。

如果任何公理含有非形式正文，则表达式的解释不是由 SDL 形式地定义的，而是可以由解释者根据非形式正文来确定。我们认为如果规定了非形式正文，则等式集合是不完全的，因而还没有用 SDL 给出完全的形式规格。

在抽象语法中，值名总是由量化等式引入，并且，对应的值具有一个值标识符，此值标识符是由包围它的量化等式的类别标识符所限定的值名。例如

FOR ALL z, z IN X (FOR ALL z IN X...)) 仅引入了一个名为 z、类别为 X 的值标识符。

在具体语法中，不允许对值标识符规定一个限定符。

由量化等式引入的每一个值标识符所具有的类别就是在量化等式中用*类别引用标识符*来指明的类别。隐含的量化条件的类别就是出现无界标识符的上下文所要求的类别。如果在隐含量化条件的一个值标识符的上下文中允许不同的类别，则此标识符应具有一个类别，此类别与等式中对它的使用是一致的。

一个项有一个类别，它就是此值标识符的类别、或是该运算符（字面值）的结果类别。

除非可以从几个等式中推出两个字面值表示同一个值，否则，每一个字面值分别表示不同的值。

例 1

```
FOR ALL b IN logical(eq(b,b)==T)
```

例 2

```
neq(T,F) ==T;      neq(T,T) ==F;
neq(F,T) ==T;      neq(F,F) ==F;
```

例 3

```
eq(b,b)                    ==T;
eq(F,eq(T,F))              ==T;
eq(eq(b,a), eq(a,b))       ==T;
```

5.2.4 条件等式

采用条件等式允许规定一些等式，使这些等式仅在某些限制条件得到满足时方才成立。限制条件以简单等式的形式写出。

抽象文法

```
条件等式        ::   限制条件 set
                     限制等式
限制条件        =    非量化等式
限制等式        =    非量化等式
```

具体正文文法

〈条件等式〉::=

〈限制条件〉{，〈限制条件〉}＊＝＝〉〈限制等式〉

〈限制等式〉::＝

〈非量化等式〉

〈限制条件〉::＝

〈非量化等式〉

语义

一个限制等式规定了只有在条件满足时，某些项才表示相同的值。条件是：在此限制等式中的任意值标识符表示这样一个值，它可以从其它的等式说明是满足此限制条件的。一个值将满足一个限制条件，仅当对于这个值来说，该限制条件可以从其它等式中推导出来。

对一个数据类型；含有条件等式的一组等式的语义可以导出如下：

a) 量化条件可以去掉，采用的办法是产生每一个可以从量化等式中推出的基本项等式。因为这既适用于显式量化条件又适用于隐式量化条件，所以，可以产生出一组仅包含基本项的非量化等式。

b) 把一个条件等式称为可证明的条件等式，如果可以从不是限制等式的非量化等式来证明所有的限制条件（仅用基本项来表示）都得到满足。如果存在一个可证明的条件等式，则它将由可证明的条件等式的限制等式所取代。

c) 如果在该组等式之中仍有条件等式，并且，这些条件等式都不是可证明的条件等式，则删掉这些条件等式，否则返回到步骤（b）。

d) 剩余的非量化等式集合定义了该数据类型的语义。

例

z/＝0＝＝True＝＝〉(x/z)＊z＝＝x

5.3 基础代数模型（非形式描述）

SDL 中的数据定义是以在§5.2 节中所定义的数据核为基础的。除前面的定义之外，运算符和值还须给予一些更进一步的意义，以便可以解释表达式。例如，用在连续信号中的表达式，用在允许条件、过程调用、输出动作、创建请求、赋值语句、置定时和复位语句、出口语句、进口语句、判定和视见中的表达式。

通过采用在下面的§5.3.1 到§5.3.6 节中说明的基础代数体系，把必须添加的意义赋给表达式。①

在一个 SDL 规格中的任一地方，分层定义的最后数据类型将是适用的，但将有一组类别是可见的。这一组类别将是在层次上高于所述问题的地方的各个层次的所有类别的并集，如在§5.2 节中所解释的那样。

（在这一节中，符号＝被用作为一个等式相等的符号，而在 SDL 中，用符号＝＝来表示等

① §5.3.1 到§5.3.6 的正文已在 ISO 和 CCITT 之间达成了一致意见，把这段正文作为对于抽象数据类型的基础代数模型的公共的非形式描述。这部分正文不但出现在本建议之中，而且也是 ISO IS8807 的一个附录（在拓扑图形和数值方面有适当的改变）。

式相等，这样，符号＝可以用作为相等运算符。在本节中使用符号＝，是因为在已发表的基础代数的著作中它是所用的规范的符号。）

5.3.1 引言

基于基础代数的数据的意义和解释分三个步骤来说明：

a) 标记

b) 项

c) 值

5.3.1.1 表示法

不同的符号可以表示同样的概念，这种想法是很平常的。例如，通常人们都认为正的阿拉伯数字（1，2，3，4，……）和罗马数字（Ⅰ，Ⅱ，Ⅲ，Ⅳ，…）表示具有同样特性的数字集合。作为另一个例子，前辍函数符号的（plus（1，1））、中辍符号（1＋1）和逆波兰符号（11＋）都可以表示同样的运算符，这是大家公认的。更进一步说，不同的用户对于同一个概念可以使用不同的名字（或许是因为他们使用不同的语言），这使得｛true，false｝，｛T，F｝，｛0，1｝，｛vrai，faux｝这些对偶可以成为布尔类别的不同的表示法。

重要的是标识物之间的抽象关系而不是具体关系。这样，对于数字，感兴趣的是1和2之间的关系与Ⅰ和Ⅱ之间的关系是一样的。同样，对于运算符，感兴趣的是运算符标识，其它运算符标识以及与变元表之间的关系。具体的结构例如括号，它使得我们能够区别（a＋b）＊c和a＋（b＊c），我们对它感兴趣仅是因为通过它能够确定基础的抽象概念。

这些抽象概念包含在概念的一个抽象语法中。它可以由不止一个的具体语法来实现。例如，下面的两个具体例子都描述相同的数据类型特性，但用了不同的具体语法。

```
NEWTYPE bool LITERALS true, false;
OPERATORS "not" :bool->bool;
AXIOMS
    not(true)          ==false;
    nto(not(a))        ==a;
ENDNEWTYPE bool;

NEWTYPE int LITERALS zero,one;
OPERATORS    plus      :int,int->int;
             minus     :int,int->int;
AXIOMS
    plus(zero,a)              ==a;
    plus(a,b)                 ==plus(b,a);
    plus(a,plus(b,c))         ==plus(plus(a,b),c);
    minus(a,a)                ==zero;
    minus(a,zero)             ==a;
    minus(a,minus(b,c))       ==minus(plus(a,c),b);
    minus(minus(a,b),c)       ==minus(a,plus(b,c));
    plus(minus(a,b),c)        ==minus(plus(a,c),b);
ENDNEWTYPE int;

NEWTYPE tree LITERALS nil;
OPERATORS
      tip          :int        ->tree;
      isnil        :tree       ->bool;
      istip        :tree       ->bool;
      node         :tree,tree  ->tree;
      sum          :tree       ->int;
AXIOMS
    istip(nil)                ==false;
    istip(tip(i))             ==true;
    istip(node(tl,t2))        ==false;
    isnil(nil)                ==true;
    isnil(tip(i))             ==false;
    isnil(node(t1,t2))        ==false;
    sum(node(t1,t2))          ==plus(sum(t1),sum(t2));
    sum(tip(i))               ==i;
    sum(nil)                  ==zero,
ENDNEWTYPE tree;
```

例 1

```
TYPE          bool          IS
SORTS         bool
OPNS          true  :       ->bool
              false :       ->bool
              not   :bool   ->bool
EQNS OFSORT                 bool FOR ALL a :bool
      not(true)             =false;
      not(not(a))           =a
ENDTYPE

TYPE        int  IS  bool WITH
SORTS       int
OPNS        zero  :                 ->int
            one   :                 ->int
            plus  :    int,int      ->int
            minus :    int,int      ->int
EQNS OFSORT int FOR ALL a,b,c:int
      plus(zero,a)              =a;
      plus(a,b)                 =plus(b,a);
      plus(a,plus(b,c))         =plus(plus(a,b),c);
      minus(a,a)                =zero;
      minus(a,zero)             =a;
      minus(a,minus(b,c))       =minus(plus(a,c),b);
      minus(minus(a,b),c)       =minus(a,plus(b,c));
      plus(minus(a,b),c)        =minus(plus(a,c),b)
ENDTYPE

TYPE          tree  IS  int     WITH
SORTS         tree
OPNS          nil   :             ->tree
              tip   :int          ->tree
              isnil :tree         ->bool
              istip :tree         ->bool
              node  :tree,tree    ->tree
              sum   :tree         ->int
EONS OFSORT bool FOR ALL i:int,t1,t2:tree
          istip(nil)            =false;
          istip(tip(i))         =true;
          istip(node(t1,t2))    =false;
          isnil(nil)            =true;
          isnil(tip(i))         =false;
          isnil(node(t1,t2))    =false
OFSORT  int  FOR ALL i:int,t1,t2:tree
    sum(node(t1,t2))            =plus(sum(t1),sum(t2));
    sum(tip(i))                 =i;
    sum(nil)                    =zero
ENDTYPE
```

例 2

这个例子将用来说明一些概念。类别和字面值的定义将是最初要考虑的。应该注意的是，字面值被当作运算符的一种特殊情况，这就是不带参量的运算符。我们可以通过下面的方式以第一种形式引入某些类别和字面值。

```
NEWTYPE int LITERALS zero,one;…
NEWTYPE bool LITERALS true,false;…
NEWTYPE tree LITERALS nil;…
```

或是由下面的方式以第二种形式引入

```
…
SORTS     bool
OPNS      true :          ->bool
          false:          ->bool
…
SORTS     int
OPNS      zero :          ->int
          one  :          ->int
…
SORTS     tree
OPNS      nil  :          ->tree
…
```

在下文中将只使用第二种形式，因为它最接近于很多基础代数出版物中所使用的表述方式。应当注意的是在两种情况中，项的形式是一样的，最显著的区别是引入字面值的方式。应当记得，为了表述概念有必要采用一种具体的符号，然而代数的意义与所采用的符号无关，所以系统地把一些名字重新命名(保持同样的唯一性)和把前缀符号改变为波兰符号，并不会改变由此类型定义所确定的含意。

5.3.2 标记

与每一个类别有关的运算符可以是一个或多个。每一个运算符具有一种运算功能，它规定了把一个或多个输入类别与一个结果类别联系起来。

例如，下面的运算符可以加入上面所定义的类别

```
…
SORTS     bool
OPNS      true  :                ->bool
          false :                ->bool
          not   : bool           ->bool
…
SORTS     int
OPNS      zero  :                ->int
          one   :                ->int
          plus  : int,int        ->int
          minus : int,int        ->int
…
SORTS     tree
OPNS      nil   :                ->tree
          tip   : int            ->tree
          isnil : tree           ->bool
          istip : tree           ->bool
          node  : tree,tree      ->tree
          sum   : tree           ->int
```

…

可用的类型的标记就是类别的集合和可见的运算符(包括字面值和带有参数的运算符)的集合。

如果对于一个标记中的每一个运算符,其功能所产生的类别包含在此类型之类别集合之中,那么,我们说此类型的标记是完全的(闭合的)。

5.3.3 项和表达式

我们所关心的语言是这样一种语言,它的表达式可以是变量、字面值或者是作用于表达式的运算符。一个变量是关联于一个表达式的一个数据实物。一个变量的解释可以用关联于此变量的表达式的解释来取代。用这种方式可以把变量消掉,以便把一个表达式的解释简化为把各种运算符应用于字面值。

因此,在解释的时候,通过给开放的表达式(即一个含有变量的表达式)提供实在参数就可把它变成一个闭合的表达式(即一个不含有变量的表达式)。

一个闭合表达式对应于一个基本项。

一个类别所有可能的基本项的集合称为此类别的基本项集合。例如,对于上面所定义的布尔类,其基本项集合将包括

{true,false,not(true),not(false),not(not(true)),…}

可以看到,即使对于这种极简单的类别,其基本项集合也是无限的。

5.3.3.1 项的生成

给定类型的一个标记,就可能为这个类型生成基本项集合。

类型的字面值集合被认为是基本项的基本集合。每一个字面值有一个类别,因此,每一个基本项有一个类别。对于上面定义的类型,这个基本项的基本集合是:

{zero,one,true,false,nil}

对于此类型的运算符集合中的每一个运算符,通过取代每一个变元来产生基本项,要用所有先前为那个变元生成的正确类别的基本项来取代变元。每一个运算符的结果类别就是由那个运算符生成的基本项的类别。所得到的基本项的集合被加到事先已存在的基本项集合,以产生一个新的基本项集合。对于上面的类型,这就是:

{zero,	one,	true,	false,	nil,
plus(zero,zero),	plus(one,one),	plus(zero,one),	plus(one,zero),	
minus(zero,zero),	minus(one,one),	minus(zero,one),	minus(one,zero),	
not(true),	not(false),	tip(zero),	tip(one),	
isnil(nil),	istip(nil),	node(nil,nil),	sum(nil)	}

这个新的基本项集合被用作为事先已存在的基本项集合。然后进一步应用前述算法以进一步生成新的基本项集合。这个基本项集合将包括：

```
{zero,              one,              true,              false,              nil,
plus(zero,zero),    plus(one,one),    plus(zero,one),    plus(one,zero),     ···
plus(zero,plus(zero,zero)),           plus(zero,plus(one,one)),              ···
plus(zero,sum(nil)),                  ···
isnil(node(nil,nil)),                 istip(node(nil,nil)),  node(nil,node(nil,nil)),
···,                                                         sum(node(nil,nil))}
```

重复地运用这个算法来产生这个类型的所有可能的基本项，这就是这个类型的基本项集合。一个类别的基本项集合就是该类型中具有那个类别的基本项集合。

通常，这一过程将永无休止地继续下去，以产生无限多个项。

5.3.4 值与代数

一个类别的每一个项表示那个类别的一个值。从上面可以看到，如果我们不给出某种定义，规定一些项怎样是相等的（即它们表示相同的值），那么即使是一个简单的类别，如布尔类，都具有无限多个项，从而具有无限多个值。这种定义是由用项来定义的等式给出，而等式规定了项的关系。在没有 istip 和 isnil 的情况下，布尔类别可以由下面的等式把值限制为只有两个值：

```
not (true) =false;
not (false) =true
```

这样的等式规定了一些项是相等的，这样就可以获得这些项的两个等价类。

```
{true,   not (false),   not (not (true)),   not (not (not (false))), ···}
{false,  not (true),    not (not (false)),  not (not (not (true))),  ···}
```

于是每一个等价类代表一个值，而此等价类内的各个成员只是同一个值的不同表示法。

值得注意的是：除非项由等式定义它们相等，否则它们是不等的（也就是说它们不表示同样的值）。一种代数定义了满足此代数标记的项的集合。这种代数的等式把一些项相互联系起来。

通常，在一种代数中，对一个类别的每一个值会有多个表示法。

对于一种给定标记的一种代数是一种基础代数，当且仅当对于这种标记能够给出同样特性的任何其他代数都可以系统地变换成为这种基础代数。（正规地说，这种变换称为同态。）

假如不是这样，istip 和 isnil 所产生的值总是在 true 和 false 的等价类中，则对于布尔类的一种基础代数就是这一对字面值

```
{true, false}
```

并且，没有等式。

5.3.4.1 等式和量化

对一种类别，例如布尔类别，其中仅有有限个数的值，所有的等式都可以只用基本项写出，就是用仅含有字面值和运算符的项写出。

当一个类别包含多个值时，用基本项写出所有的等式是不实际的，并且，对于具有无限多个值的类别（例如整数），这样显式地枚举是不可能的。采用写量化等式的技术，可以只用一个量化等式来表示一个可能是无限的等式集合。

一个量化等式在项中包含有值标识符。这种项称为复合项。仅具有基本项的等式集合可以从此量化等式推导出来。办法是通过在量化等式中用此值标识符的类别的一个基本项代替每一个值标识符来系统地产生出一个又一个等式。例如

```
FOR ALL b:bool not (not (b)) =b
```

表示

```
not (not (true)) =true;
not (not (false)) =false
```

现在我们可以采用布尔类的另外一组等式如下：

```
FOR ALL b:bool
not (not (b)) =b;
not (true)    =false
```

当量化的值标识符的类别从上下文看来是明显的时候，通常可以省略定义此值标识符的子句，这样，上例变成

```
not (not (b)) =b;
not (true)    =false
```

5.3.5 代数规格和语义（意义）

代数规格包括一个标记和用于此标记的每一个类别的等式集合。这些等式集合引出相等性的关系，用来规定规格的意义。

用符号＝来表示一个等价关系，它满足自反，对称和传递性质以及替换性质。

连同类型给出的等式允许将一些项置于等价类中。在同一等价类中的任意两个项被认为具有相同的值。这一机制可以用来鉴别在语法上是不同的项，即使它们具有相同的给定的值。

同样类别的两个项 TERM1和 TERM2是在同一个等价类中，如果

a) 有这样一个等式

TERM1＝TERM2，

或

b) 从给定的一组量化等式导出的一个等式是

TERM1＝TERM2，

或

c) i) TERM1在含有 TERMA 的等价类中，并且

ii) TERM2在含有 TERMB 的等价类中，并且

iii) 存在下面的等式，或从一组给定的量化等式可以推导出下面的等式

TERMA＝TERMB，

或

d) 通过用 TERM1的一个子项的等价类中的一个项代替 TERM1的这个子项，产生出一个项 TERM1A，而且又可证明 TERM1A 和 TERM2是在同一个等价类中。

通过应用所有的等式，一个类别的所有的项被划分成一个或多个等价类。这个类别有多少个等价类，它就有多少个值。每一个等价类表示一个值，并且，一个等价类中的每一个成员表示相同的值。

5.3.6 值的表示法

对一个表达式的解释意味着首先要通过确定在解释点的表达式中所用的变量的实际值来导出基本项，然后找出这个基本项的等价类。此项的等价类确定了此表达式的值。

这样，通过确定给定的一组变元的结果值，将意义赋给了用在表达式中的运算符。

通常在等价类中选一个字面值用来表示此等价类的值。例如，布尔类可用 True 和 False 表示，而自然数用0、1、2、3等来表示。当没有字面值时，则通常使用最简单的项（运算符数最少）。例如，对负整数，通常采用的符号是－1、－2、－3等。

5.4 SDL 数据的被动使用

在§5.4.1节中，对在§5.2节中的数据定义构件作了扩展。若表达式是“被动的”(即不依赖于变量或系统状态)，在§5.4.2节中规定了如何解释表达式中抽象数据类型的使用。在§5.5节中规定了如何解释非被动的表达式（即“主动的”表达式）

5.4.1 扩展数据定义构件

在§5.2节中定义的构件是在下面阐述的更简洁形式的基础。

抽象文法

对这些构件的大部分来说，没有附加的抽象语法。在§5.4.1节和所有§5.4.1节的分节中，相关的抽象语法通常可在§5.2节中找到。

具体正文文法

〈扩展特性〉 ::=
〈继承规则〉
| 〈生成程序实例〉
| 〈结构定义〉

〈扩展复合项〉 ::=
〈扩展运算符标识符〉(〈复合项表〉)
| 〈复合项〉〈中缀运算符〉〈项〉
| 〈项〉〈中缀运算符〉〈复合项〉
| 〈一元运算符〉〈复合项〉
| 〈条件复合项〉

〈扩展基本项〉 ::=
〈扩展运算符标识符〉
(〈基本项〉{,〈基本项〉} *)
| 〈基本项〉〈中缀运算符〉〈基本项〉
| 〈一元运算符〉〈基本项〉
| 〈条件基本项〉

〈扩展运算符标识符〉 ::=
〈运算符标识符〉〈感叹号〉
| 〈生成程序形式名〉
| [〈限定符〉]〈引用文运算符〉

〈扩展运算符名〉 ::=
〈运算符名〉〈感叹号〉
| 〈生成程形形式名〉
| 〈引用文运算符〉

〈感叹号〉 ::=
!

〈扩展字面值名〉::=
〈字符串字面值〉
| 〈生成程序形式名〉
| 〈名字类字面值〉
〈扩展字面值标识符〉::=
〈字符串字面值标识符〉
| 〈生成程序形式名〉

规则〈扩展特性〉、〈扩展复合项〉、〈扩展基本项〉、〈扩展运算符名〉、〈扩展字面值名〉和〈扩展字面值标识符〉，分别在数据核中扩展了下述的规则：〈部分类型定义〉（§5.2.1节）、〈复合项〉（§5.2.3节）、〈基本项〉（§5.2.3节）、〈运算符名〉（§5.2.2节）、〈字面值〉（§5.2.2节）和〈字面值标识符〉（§5.2.3节）。上述规则通过〈继承规则〉（§5.4.1.11节）、〈生成程序实例〉（§5.4.1.12.2节）、〈生成程序形式名〉（§5.4.1.12.1节）、〈条件复合项〉（§5.4.1.6节）、〈条件基本项〉（§5.4.1.6节）、〈字符串字面值〉和〈字符串字面值标识符〉（§5.4.1.2节）以及〈名字类字面值〉（§5.4.1.14节）又被进一步扩展。在§5.4.1.1节中定义了以下规则：〈中缀运算符〉、〈一元运算符〉、〈引用文中缀运算符〉和〈引用文一元运算符〉。

〈生成程序形式名〉的替换项仅在〈生成程序正文〉中的〈特性表达式〉中有效（参见§5.4.1.12节）。此〈特性表达式〉把那个名字规定为形式参数。

带有一个前导“(”的〈生成程序形式名〉的〈扩展复合项〉和〈扩展基本项〉的替换项，仅当该〈生成程序形式名〉定义为 OPERATOR 类（参见§5.4.1.12）才是合法的。

仅当〈生成程序形式名〉定义为 LITERAL 类时（参见§5.4.1.12节），带有这个〈生成程序形式名〉的〈扩展字面值名〉的替换项才是合法的。

仅当＜生成程序形式名＞定义为 LITERAL 类或 CONSTANT 类时（参见§5.4.1.12节），带有该〈生成程序形式名〉的〈扩展字面值标识符〉的替换项才是合法的。

如果定义一个运算符名带有一个〈感叹号〉，则此〈感叹号〉在语义上是那个名字的一部分。

〈运算符名〉〈感叹号〉或者〈运算符标识符〉〈感叹号〉这种形式分别表式*运算符名*（参见§5.2.2节）和*运算符标识符*（参见§5.2.3节）。

语义

如果定义一个运算符名带有一个〈感叹号〉，它还是具有一个运算符的普通语义，但此运算符名仅在公理中可见。

5.4.1.1 *特殊运算符*

这些是具有特殊的语法形式的运算符名。为使算术运算符和布尔运算符能具有它们通常的语法形式，引入了特殊的语法。例如，使用者可以写“(1+1)＝2”，而不是被强迫使用 equal (add (1, 1), 2)。对每一个运算符，究竟哪个类别是有效的，这将决定于数据类型定义。

具体正文文法

〈引用文运算符〉::=

〈引号〉〈中缀运算符〉〈引号〉
| 〈引号〉〈一元运算符〉〈引号〉

〈引号〉 ::=
"

〈中缀运算符〉::=
=>
| OR
| XOR
| AND
| IN
| /=
| =
| >
| <
| <=
| >=
| +
| /
| *
| //
| MOD
| REM
| -

〈一元运算符〉::=
-
| NOT

语义

项中的中缀运算符具有运算符的通常的语义，但附有如上的中缀语法或用引号括起来的前缀语法。

项中的一元运算符具有运算符的通常的语义，但具有如上的前缀语法或用引号括起来的前缀语法。

中缀运算符或一元运算符用引号括起来的形式是运算符的合法名字。

这些中缀运算符具有一个优先次序，它决定了运算符的组合的次序。其组合次序与在§5.4.2.1节中所规定的〈表达式〉中运算符组合次序是一样的。

当这种组合次序含糊不清时，如在

a OR b XOR c;

中，则应从左至右进行组合，这样，上面的项等同于

(a OR b) XOR c;

值得注意的是，〈引用文运算符〉MOD 和 REM 没有预定义的语义，这是因为在预定义数据类别中没有定义它们。

模型

形式为

〈项1〉〈中缀运算符〉〈项2〉的一个项，是形式

"〈中缀运算符〉"（〈项1〉，〈项2〉）

的派生语法。它以"〈中缀运算符〉"作为一个合法的名字。"〈中缀运算符〉"表示一个运算符名。同样地，形式

〈一元运算符〉〈项〉

是形式

"〈一元运算符〉"（〈项〉）

的派生语法，它以"〈一元运算符〉"作为一个合法的名字，并表示一个运算符名。

（注意：不要把SDL相等符号（=）与SDL项的等价符号（==）混淆起来。）

5.4.1.2 字符串字面值

具体正文文法

〈字符串字面值标识符〉::=

[〈限定符〉]〈字符串字面值〉

〈字符串字面值〉::=

〈字符串〉

〈字符串〉是一个词法单位，其定义在§2.2.1节中给出。

〈字符串字面值标识符〉表示在抽象语法中的一个字面值运算符标识符。

〈字符串字面值〉表示在抽象语法中的一个唯一的字面值运算符名（§5.2.2）。它是从〈字符串〉导出的。

语义

字符串字面值标识符是用项和表达式中的字符串字面值构成的标识符。

字符串字面值用于预定义数据类别Character和Charstring（参见§5.6节）。它们与名字类字面值（参见§5.4.1.14节）和字面值映象（参见§5.4.1.15节）也有一种特殊的关系。这些字面值也可以规定用于其它场合。

〈字符串字面值〉有一个长度，它是〈字符串〉中的〈字母数字〉个数，加上〈其它字符〉个数，加上〈特殊符〉、〈句号〉、〈下划线〉与〈空格〉的个数，再加上〈撇号〉〈撇号〉对数（参见§2.2.1节）。

一个〈字符串字面值〉

a) 具有大于1的长度，并且

b) 具有一个子串。从〈字符串〉去掉最后一个字符（〈字母数字〉、〈其它字符〉、〈特殊符〉、〈句号〉、〈下划线〉、〈空格〉或〈撇号〉〈撇号〉对），就构成这个子串，并且

c) 定义那个子串为一个字面值，这样

子串//用撇号括起来的删去字符

是具有与该〈字符串字面值〉相同类别的一个合法项，

因此，存在一个由此具体语法给出的隐含等式，这就是：此〈字符串字面值〉等于这

个子串后面跟有中缀运算符“//”，后面再跟有用撇号括起来的删去的字符所构成的〈字符串〉。

例如，位于

```
NEWTYPE s
LITERALS 'ABC', 'AB', 'A', 'B','''';
OPERATORS "//": s, s -> s;
```

中的字面值 'ABC'，'AB''，和 'AB' 具有隐含等式

```
'ABC' == 'AB' // 'C';
'AB''' == 'AB' //'''';
'AB' == 'A' // 'B';
```

5.4.1.3 预定义数据

在§5.6节中定义的预定义数据中有布尔类别，它定义了两个字面值 True 和 False 的特性。相等性（§5.4.1.4）、布尔公理（§5.4.1.5）、条件项（§5.4.1.6）、排序（§5.4.1.8）以及同义类型（§5.4.1.9）等的语义依赖于布尔类别（§5.6.1）的定义。名字类字面值（如果使用〈有规律的间隔〉—§5.4.1.14）和字面值映象（§5.4.1.15）的语义，也分别依赖于字符（§5.6.2）和字符串（§5.6.4）的定义。

预定义数据应在系统层定义。

5.4.1.4 相等性

具体正文文法

在〈部分类型定义〉中引入的每一个类别名对于=和/=都有一个隐含的运算符标记，并且，对于这些运算符有一组隐含的等式。

引入一个名为S的类别的一个〈部分类型定义〉具有下面的一对隐含的运算符标记，

```
" =" :S, S -> Boolean;
" /=" :S, S -> Boolean;
```

这里 Boolean 是预定义布尔类别。

引入一个名为S之类别的一个〈部分类型定义〉具有一组隐含的等式

```
FOR ALL a, b, c IN S (
    a=a                                  ==True;
    a=b                                  ==b=a;
    ( (a=b) AND (b=c)) => a=c            ==True;
    a/=b                                 ==NOT (a=b);
    a=b==True  ==>        a              ==b )
```

最后的等式表示相等性的替换特性。

如果从一些等式（显式的、隐含的和导出的）可以推出

```
True==False
```

这就与布尔数据类型的假设特性相矛盾了，如果是这样，这个定义必定是错的。推导出

```
True==False
```

必定是不可能的。在数据类型定义之外所用到的每一个 Boolean 基本表达式必须解释为 True

或 False。如果不可能把这样一种表达式简化为 True 或 False，则该规格是不完全的，并且使得对此数据类型有几个解释。

语义

对于由部分数据类型定义引入的每一个类别，存在一个隐含的运算符定义和有关相等性的等式的定义。

在具体语法中，符号＝和/＝代表等于运算符和不等于运算符的名字。

5.4.1.5 布尔公理

具体正文文法

〈布尔公理〉::=
〈布尔项〉

语义

布尔公理是真理的陈述，它对于所定义的数据类型在所有条件下都成立，并由此可以用来规定数据类型的行为。

模型

形式为
〈布尔项〉;
的公理是具体语法等式
〈布尔项〉 ==True;
的派生语法；它具有抽象语法中一个等式的普通关系。

5.4.1.6 条件项

下面含有条件项的等式称为一个条件项等式。

抽象文法

条件复合项	=	条件项
条件基本项	=	条件项
条件项	::	条件
		结果项
		替换项
条件	=	项
结果项	=	项
替换项	=	项

条件的类别必须是预定义的 Boolean 类别，条件必须不是错误项。结果项和替换项必须有相同的类别。

一个条件项是条件复合项，当且仅当在该条件、该结果项或替换项中有一个或多个项是复

合项。

一个条件项是条件基本项，当且仅当在该条件中，在该结果项或替换项中，所有的项都是基本项。

具体正文文法

〈条件复合项〉 ::=
〈条件项〉
〈条件基本项〉 ::=
〈条件项〉
〈条件项〉 ::=
IF〈条件〉THEN〈结果项〉ELSE〈替换项〉 FI
〈条件〉 ::=
〈Boolean 项〉
〈结果项〉 ::=
〈项〉
〈替换项〉 ::=
〈项〉

语义

在一个等式中使用的条件项在语义上等价于两组等式，其中所有用 Boolean 项表示的量化值标识符均已被消去。

这一组等式可以这样来构成，即在整个条件项等式中同时用适当类别的各个基本项取代该条件中的各个值标识符。在这组等式中，该条件将总是由一个布尔基本项所取代。在下文中，这组等式被称为扩展基本集。

一个条件项等式等价于这样的一组等式，它包括：

a) 对于在扩展基本集中条件等于 True 的每一个等式，扩展基本集中的那个等式用（基本的）结果项来取代其条件项，而且

b) 对于在扩展基本集中条件等于 False 的每一个等式，扩展基本集中的那个等式用（基本的）替换项来取代其条件项。

注意 在具有下面形式的等式的特殊情况下

ex1==IF a THEN b ELSE c FI;

这等价于下面的两个条件等式

a==True ==〉 ex1==b;
a==False ==〉 ex1==c;

举例

IF i=j * j THEN posroot(i)ELSE abs(j)FI==IF positive(j)THEN j ELSE -j FI;

注 — 这仅是一个例子，还有规定这些特性的更好的方法。

5.4.1.7　错误

使用“错误”的目的是为了即使在运算符运算的结果没有确定的意义的情况下也能够完全规定一个数据类型的特性。

抽象文法

错误项　::　()

在复合项中，错误项必须不用作运算符标识符的变元项。

错误项必须不作为限定条件的一部分。

从一组等式推出一个字面值运算符标识符等于错误项是不可能的。

具体正文文法

〈错误项〉::=
ERROR〈感叹号〉

语义

项可以是一个错误，这样，可以规定一个运算符产生错误的情况。如果在解释期间出现了这些情况，则系统将来的行为是不确定的。

5.4.1.8　排序

具体正文文法

〈排序〉　::=
ORDERING

(〈排序〉参见§5.2.2.节)

语义

排序关键字是一个简写符号，用于显式地规定排序运算符，和规定一个部分类型定义的一组排序等式。

模型

一个〈部分类型定义〉用关键字 ORDERING 引入一个名为 S 的类别，意味着一个运算符标记集合等效于下列显式定义：

" <": S, S -〉 Boolean;
" >": S, S -〉 Boolean;
" <=": S, S -〉 Boolean;
" >=": S, S -〉 Boolean;

这里 Boolean 是预定义的布尔类别，同时也隐含了布尔公理。

```
FOR ALL a, b IN  S
        (
        " <" (a, a)       ==False;
        " <" (a, b)       ==" >" (b, a);
        " <=" (a, b)      ==" OR" (" <" (a, b)," =" (a, b));
        " >=" (a, b)      ==" OR" (" >" (a, b)," =" (a, b));
        " <" (a, b)       =>NOT (" <" (b, a));
        " <" (a, b)       AND" <" (b, c) =>" <" (a, c);
        );
```

当一个〈部分类型定义〉既包含〈字面值表〉，又包含关键字 ORDERING 时，这些〈字面值标记〉将按照从小到大的次序给出名字，即：

```
LITERALS A, B, C;
OPERATORS ORDERING;
```

意味着 A<B，B<C。

5.4.1.9　同义类型

同义类型规定了类别的值的集合。用作为一个类别的同义类型具有与此同义类型所引用的类别相同的语义，但还要验证同义类型的值是在该类别的值的集合之中。

抽象文法

```
同义类型标识符   =  标识符
同义类型定义     =  同义类型名
                    父辈类别标识符
                    范围条件
同义类型名       =  名字
父辈类别标识符   =  类别标识符
```

具体正文文法

```
〈同义类型〉  ::=
      〈同义类型标识符〉
〈同义类型定义〉  ::=
            SYNTYPE
             〈同义类型名〉 = 〈父辈类别标识符〉
            [〈缺省赋值语句〉][CONSTANTS 〈范围条件〉]
            ENDSYNTYPE [〈同义类型名〉]
          | NEWTYPE 〈同义类型名〉[〈扩展特性〉]
                〈特性表达式〉 CONSTANTS 〈范围条件〉
                ENDNEWTYPE [〈同义类型名〉]
〈父辈类别标识符〉 ::=
            〈类别〉
```

一个〈同义类型〉可用来替换一个〈类别〉(参见§5.2.2节)。

具有关键字SYNTYPE和"=〈同义类型标识符〉"的〈同义类型定义〉是下面定义的语法的导出语法。

具体语法中具有关键字SYNTYPE的〈同义类型定义〉对应于抽象语法中的同义类型定义。

带有关键字NEWTYPE的一个〈同义类型定义〉可以从包含CONSTANTS〈范围条件〉的一个〈部分类型定义〉中区分出来。这样的一种〈同义类型定义〉是以某一个名字(例如anon)引入一个〈部分类型定义〉的简化形式,在此〈部分类型定义〉之后跟随一个〈同义类型定义〉,它带有以此名字的类别为基础的关键字SYNTYPE,即:

```
NEWTYPE X/*细节*/
        CONSTANTS/*常数表*/
ENDNEWTYPEx;
```

等于〈部分类型定义〉

```
NEWTYPE  anon/*细节*/
ENDNEWTYPE anon;
```

后面跟着一个〈同义类型定义〉

```
SYNTYPE  X=anon
        CONSTANTS/*常数表*/
ENDSYNTYPE X;
```

当在一个运算符的〈变元表〉中将一个〈同义类型标识符〉用作为一个〈变元〉时,变元表中此变元的类别是此同义类型的父辈类别标识符。

当用〈同义类型标识符〉作为一个运算符的结果时,此结果的类别为此同义类型的父辈类别标识符。

当把〈同义类型标识符〉用作为一个名字的限定符时,此限定符为该同义类型的父辈类别标识符。

在一个〈同义类型定义〉结尾之处,在关键字ENDSYNTYPE或ENDNEWTYPE之后给出的任选的〈同义类型名〉必须分别与在SYNTYPE或NEWTYPE后规定的〈同义类型名〉相同。

如果使用了关键字SYNTYPE,并省略了〈范围条件〉,则此类别的所有值都在范围条件之内,这样,此〈同义类型标识符〉与此类别标识符有完全相同的语义且范围条件总是对的。

语义

同义类型定义规定了一个同义类型,它引用一个类别标识符和范围条件。规定一个同义类型标识符是与规定此同义类型之父辈类别标识符是一样的,但下面的情况除外:

a) 对一个声明为同义类型的变量的赋值语句(参见§5.5.3节),

b) 一个信号的输出,如果规定此信号的一个类别是一个同义类型(参见§2.7.4节),

c) 调用一个过程,当对此过程的IN参数变量规定的一个类别是一个同义类型时(参见§2.4.5节),

d) 创建一个进程,当为此进程参量规定的一个类别是一个同义类型时(参见§2.7.2和§2.4.4节),

e) 一个信号的输入,以及关联于此输入的一个变量所具有的一个类别是同义类型(参见§2.6.4节),

f） 应用在一个运算符的一个表达式中，它有一个同义类型被定义为一个变元类别，或定义为一个结果类别（参见§5.4.2.2和§5.5.2.4节），

g） 关于定时器的一个置定时或复位语句，而在定时器定义中有一个类别是一个同义类型（参见§2.8节），

h） 一个进口定义（参见§4.13节）。

例如，一个〈同义类型定义〉带有关键字SYNTYPE和“=〈同义类型标识符〉”，等效于用该〈同义类型标识符〉之〈同义类型定义〉的〈父辈类别标识符〉来代替其〈父辈类别标识符〉。即：

```
SYNTYPE s2=n1 CONSTANTS a1:a3; ENDSYNTYPE s2;
SYNTYPE s3=s2 CONSTANTS a1:a2; ENDSYNTYPE s3;
```

等效于

```
SYNTYPE s2=n1 CONSTANTS a1:a3; ENDSYNTYPE s2;
SYNTYPE s3=n1 CONSTANTS a1:a2; ENDSYNTYPE s3;
```

当一个同义类型借助于〈同义类型标识符〉来规定时，则这两个同义类型必须不相互定义。

由一个同义类型定义规定的一个同义类型有一个标识符，它是由此同义类型名引入的名字，而此同义类型名用包围它的作用域单位的标识符来限定。

一个同义类型有一个类别，它是由在此同义类型定义中给出的父辈类别标识符所标识的类别。

一个同义类型有一个范围，它是由此同义类型定义中的那些常数所规定的值的集合。

5.4.1.9.1 范围条件

抽象文法

范围条件	::	Or运算符标识符
		条件项 **set**
条件项	=	开范围|闭范围
开范围	::	运算符标识符
		基本表达式
闭范围	::	And运算符标识符
		开范围
		开范围
Or运算符标识符	=	标识符
And运算符标识符	=	标识符

具体正文文法

〈范围条件〉 ::=

{〈闭范围〉|〈开范围〉}{,{〈闭范围〉|〈开范围〉}}*

〈闭范围〉 ::=

〈常数〉:〈常数〉

〈开范围〉 ::=

〈常数〉

|{=|/=|<|>|<=|>=}〈常数〉

〈常数〉 ::=

〈基本表达式〉

如果符号" <" (" <=" 、" >" 、" >=" 分别地) 与一个〈运算符标记〉

P, P -> Boolean

同时定义，这里P是该同义类型的类别，则它必须仅用在该〈范围条件〉的具体语法之中。这些符号代表运算符标识符。

如果符号“<=”与一个〈运算符标记〉

P, P -> Boolean

一起定义，这里P是该同义类型的类别，则只能采用一个〈闭范围〉。

一个〈范围条件〉中的〈常数〉的类别必须与该同义类型的类别相同。

语义

范围条件确定了范围检验。当一个同义类型对其类别有附加的语义时，就要用范围检验(参见§5.4.1.9节以及同义类型具有不同语义的情况—参见§5.5.3、§2.6.4、§2.7.2、§2.5.4、§5.4.2.2和§5.5.4节)。范围检验也用来确定一个判定的解释（参见§2.7.5节)。

范围检验是从此范围条件构成的运算符的应用。此运算符的这种应用必须等于True，否则，此系统的未来的行为是不确定的。范围检验可以如下推出：

a) 在〈范围条件〉中的每一个元素（〈开范围〉或〈闭范围〉）有一个对应的开范围或闭范围在条件项中。

b) 形式为〈常数〉的〈开范围〉等效于形式为=〈常数〉的一个〈开范围〉。

c) 对于一个给定项A、则

i) 一个形式=〈常数〉、/=〈常数〉、<〈常数〉、<=〈常数〉、>〈常数〉和>=〈常数〉的〈开范围〉，分别在形式A=〈常数〉，A/=〈常数〉、A<〈常数〉、A<=〈常数〉、A>〈常数〉或A>=〈常数〉的范围检验中具有子项。

ii) 一个形式为“〈第一个常数〉:〈第三个常数〉”的〈闭范围〉在形式为“〈第一个常数〉<=A AND A<=〈第二个常数〉”的范围检验中具有一个子项，这里AND对应于布尔AND运算符，并对应于抽象语法中的AND运算符标识符。

d) 有一个Or运算符标识符分布地作用于条件项set中的所有元素，(它就是所有元素的布尔逻辑的并（OR)）范围检验中要检验的项是由〈范围条件〉导出的所有子项的布尔逻辑的并（OR）构成的项。

如果规定一个同义类型时没有给出〈范围条件〉，则范围检验为True。

5.4.1.10 结构类别

具体正文文法

〈结构定义〉 ::=
STRUCT〈字段表〉[〈结束〉][ADDING]
〈字段表〉 ::=
〈字段〉{〈结束〉〈字段〉}*
〈字段〉 ::=
〈字段名〉{,〈字段名〉}* 〈字段类别〉
〈字段类别〉 ::=
〈类别〉

结构类别的每一个〈字段名〉必须不同于同一个〈结构定义〉中的每一个其它〈字段名〉。

语义

结构定义规定了结构类别，其值由具有（可以是不同的）类别的一些字段值组成。

值表的长度由结构定义来确定，并且，一个值的类别由其在值表中的位置来决定。

模型

一个结构定义是下面定义的派生语法：

a) 用来创建结构值的运算符 Make!，以及

b) 一些运算符用于修改结构值以及用于从结构值中提取字段值。

用于修改一个字段的隐含运算符的名字是用“Modify!”与该字段拼接起来构成的。

用于提取一个字段的隐含运算符的名字是用“Extract!”与该字段拼接起来构成的。

Make!运算符的〈变元素〉就是出现在该字段表中的〈字段类别〉表，其次序与它们出现时的次序相同。Make!运算符的〈结果〉就是该结构的类别标识符。

字段修改运算符的〈变元表〉就是该结构的类别标识符，其后跟有那个字段的〈字段类别〉，字段修改运算符的〈结果〉就是该结构的类别标识符。

关于字段提取运算符的〈变元表〉是该结构的类别标识符。字段提取运算符的〈结果〉就是那个字段的〈字段类别〉。

对每一个字段，有一个隐含的等式，它规定将结构的一个字段修改成为某个值，等效于把那个值赋给该字段以构成一个结构值。

对每一个字段，有一个隐含的等式，它规定提取一个结构的值的一个字段时，将返回原先构成该结构值时与那个字段关联的值。

例如

```
NEWTYPE s STRUCT
        b  Boolean;
```

```
        i   Integer;
        c   Character;
    ENDNEWTYPE     s;
```

隐含有

```
    NEWTYPE s
    OPERATORS
        Make!       :Boolean, Integer, Character     -> s;
        bModify!    :s, Boolean                      -> s;
        iModify!    :s, Integer                      -> s;
        cModify!    :s, Character                    -> s;
        bExtract!   :s                               -> Boolean;
        iExtract!   :s                               -> Integer;
        cExtract!   :s                               -> Character;
    AXIOMS
        bModify!    (Make!(x,y,z),b)                 ==Make!(b,y,z);
        iModify     (Make!(x,y,z),i)                 ==Make!(x,i,z);
        cModify!    (Make!(x,y,z),c)                 ==Make!(x,y,c);
        bExtract!   (Make!(x,y,z))                   ==x;
        iExtract!   (Make!(x,y,z))                   ==y;
        cExtract!   (Make!(x,y,z))                   ==z;
    ENDNEWTYPEs;
```

5.4.1.11 继承

具体正文文法

```
〈继承规则〉    ::=
    INHERITS〈父辈类别〉[〈字面值再命令〉]
          [ [OPERATORS]{ALL| (〈继承表〉)} [〈结束〉]] [ADDING]
〈父辈类别〉    ::=
                〈类别〉
〈继承表〉      ::=
                〈继承运算符〉{,〈继承运算符〉} *
〈继承运算符〉    ::=
                [〈运算符名〉=]〈继承运算符名〉
〈继承运算符名〉    ::=
                〈父辈类别运算符名〉
〈字面值再命名〉    ::=
                LITERALS〈字面值再命名表〉〈结束〉
〈字面值再命名表〉    ::=
                〈字面值再命名偶对〉{,〈字面值再命名偶对〉} *
〈字面值再命名偶对〉    ::=
                〈字面值再命名标记〉=〈父辈字面值再命名标记〉
```

〈字面值再命名标记〉 ::=
〈字面值 运算符名〉
| 〈字符串字面值〉

不允许类别通过继承，循环地继承自身。

〈字面值再命名表〉中的所有〈字面值再命名标记〉必须是不同的。〈字面值再命名表〉中的所有〈父辈字面值再命名标记〉必须是不同的。

〈继承表〉中的所有〈继承运算符名〉必须是不同的。〈继承表〉中所有的〈运算符名〉必须是不同的。

在〈继承表〉中规定的〈继承运算符名〉必须是在定义〈父辈类别〉的〈部分类型定义〉中所定义的〈父辈类别〉的一个可见运算符。如果定义一个运算符名带有一个〈感叹号〉，则此运算符名在这一点是不可见的。

当〈父辈类别〉的若干个运算符，与〈继承运算符名〉具有同样的名字时，则所有这些运算符都要被继承。

语义

通过用 NEWTYPE 与一个继承规则相结合，可以从一个类别得出另一个类别。用继承规则定义的类别是从父辈类别分离出来的。

如果父辈类别定义了字面值，则这些字面值名字将作为此类别的字面值的名字继承下来，除非对那个字面值进行了再命名。如果在一个字面值再命名偶对中，父辈字面值名作为第二个名字出现，这就是在进行字面值再命名，在这种情况下，此字面值被再命名为那个偶对中的第一个名字。

除“=”和“/=”外，对父辈类别的每一个运算符都有一个继承运算符，一个父辈类别的运算符是这样的一个运算符，它应符合以下两点：

a) 它由任意的部分类型定义或同义类型定义所定义（已经被定义的除外），这定义了一个类别，在继承点是可见的，并且

b) 它把此父辈类别用作为变元或作为结果。

由 ALL 来规定或由继承表来规定，把一些运算符的名字继承下来。一个继承运算符的名字

a) 与父辈类别运算符名字是相同的，如果规定了 ALL 并且在定义父辈类别的部分类型定义或同义类型定义中，此名字是显式地或隐含地定义为一个运算符名字的话，否则

b) 如果父辈运算符标识符在继承表中给出，并且给出一个运算符名字，后面跟有“=”，那么继承运算符就是再命名成这个名字。否则

c) 如果父辈运算符标识符在继承表中给出，并且没有给出一个运算符名字后面跟有“=”，则继承运算符的名字与父辈类别运算符名字相同，否则

d) 如果没有规定 ALL，并且在继承表中没有提及父辈运算符标识符，则继承运算符被再命名成一个不可见的但唯一的名字。在公理中或表达式之中，不能显式地使用这样的名字。

一个继承运算符的变元类别和结果与父辈类别的对应运算符的变元类别和结果相同，除非该变元类别或结果本身就是父辈类别，在这种情况下，要把它变成被规定的类别。也就是说

在继承运算符中，父辈类别的每一次出现，都要改变成那个新的类别。

通过继承可以从父辈类别的每一个等式推导出一个等式。父辈类别的等式是

a) 包含父辈类别的一个运算符（或字面值）的任何一个等式，

b) 通过部分类型定义或同义类型定义规定的任意一个等式，它规定了在继承点是可见的一个类别。

一个继承等式与父辈类别的对应等式是相同的，除非

a) 父辈类别的任何出现被改变成新类别，且

b) 已经重新命名继承运算符（或字面值）的父辈类别的运算符（或字面值），在继承等式中经历同样的再命名。

在（a）中改变类别的结果是：要改变继承字面值和继承运算符的字面值标识符和运算符标识符，要用新类别的类别标识符来限定它们。

模型

〈继承规则〉的具体语法是与包含该〈继承规则〉的〈部分类型定义〉或〈同义类型定义〉中的〈特性表达式〉的具体语法有关的。

抽象语法中新类别的〈字面值〉的集合对应于〈特性表达式〉中的〈字面值标记〉集合加上继承的字面值的集合。

抽象语法中新类别的〈运算符〉的集合对应于〈特性表达式〉中的〈运算符标记〉集合加上继承的运算符的集合。

抽象语法中新类别的〈等式〉的集合对应于〈特性表达式〉中的〈公理〉加上继承的等式的集合。

举例

```
NEWTYPE bit
    INHERITS Boolean
        LITERALS   1=True, 0=False;
        OPERATORS (" NOT"," AND"," OR")
    ADDING
    OPERATORS
        EXOR: bit, bit -> bit;
    AXIOMS/* 注－这里使用了 NOT 的两种不同写法 */
        EXOR (a, b) == (aAND " NOT" (b)) OR (NOT a AND b);
ENDNEWTYPE bit;
```

5.4.1.12 生成程序

生成程序使得我们可以定义带有参数的正文样板，在考虑数据类型的语义之前，可以通过实例把它扩展。

5.4.1.12.1 生成程序定义

具体正文文法

〈生成程序定义〉 :: =
GENERATOR 〈生成程序名〉(〈生成程序参量表〉)〈生成程序正文〉
ENDGENERATOR [〈生成程序名〉]
〈生成程序正文〉 :: =
[〈生成程序实例集〉]〈特性表达式〉
〈生成程序参量表〉 :: =
〈生成程序参量〉{,〈生成程序参量〉}*
〈生成程序参量〉 :: =
{TYPE|LITERAL|OPERATOR|CONSTANT}
〈生成程序形式名〉{,〈生成程序形式名〉}*
〈生成程序形式名〉 :: =
〈生成程序形式名〉
〈生成程序类别〉 :: =
〈生成程序形式名〉
|〈生成程序名〉

如果一个〈特性表达式〉处于〈生成程序正文〉之中，则在此〈特性表达式〉中必须只用〈生成程序名〉或〈生成程序形式名〉。

在〈生成程序定义〉中，所有同一类（TYPE，LITERAL，OPERATOR 或 CONSTANT）的〈生成程序形式名〉必须是不同的。在同一个〈生成程序定义〉中，LITERAL 类的名字必须与每一个 CONSTANT 类的名字不同。

关键字 GENERATOR 之后的〈生成程序名〉必须与〈生成程序定义〉中所有的类别名字不同，并且也不同于那个〈生成程序定义〉的所有 TYPE〈生成程序参量〉。

仅在〈生成程序类别〉在〈生成程序正文〉中作为一个〈扩展类别〉（参见§5.2.2节）出现时，它才是合法的，并且，其名字或是那个〈生成程序定义〉的〈生成程序名〉、或是由那个定义确定的〈生成程序形式名〉。

如果〈生成程序类别〉是一个〈生成程序形式名〉，则它必须是定义为 TYPE 类的一个名字。

关键字 ENDGENERATOR 之后的任选〈生成程序名〉必须与 GENERATOR 后面给定的〈生成程序名〉相同。

一个〈生成程序形式名〉不许在〈限定符〉中使用。一个〈生成程序名〉或〈生成程序形式名〉一定不能够

a) 被限定，或

b) 后面跟一个〈感叹号〉，或

c) 在〈缺省赋值〉中使用。

语义

一个生成程序给出了一段正文，它可以在生成程序实例中使用。

在生成程序正文中的生成程序实例的正文应该在此生成程序正文的定义处展开。

每一个生成程序参量属于一类（TYPE，LITERAL，OPERATOR 或 CONSTANT），它们分别用关键字 TYPE，LITERAL，OPERATOR 或 CONSTANT 来规定。

模型

如果生成程序被实例化，则由该生成程序定义所确定的正文仅与抽象语法有关。在定义处生成程序定义没有对应的抽象语法。

举例

```
GENERATOR bag(TYPE item)
LITERALS empty;
OPERATORS
    put         :item,bag  ->bag;
    count       :item,bag  ->Integer;
    take        :item,bag  ->bag;
AXIOMS
    take(i,put(i,b))     ==b;
    take(i,empty)        ==ERROR!;
    count(i,empty)       ==0;
    count(i,put(j,b))    ==count(i,b)+IF i=j THEN 1 ELSE 0 FI;
    put(i,put(j,b))      ==put (j,put(i,b));
ENDGENERATOR bag;
```

注 — 形式定义（见附件 F.2）不允许在限定符中使用〈生成程序形式名〉。在附件 F.2发行后，本建议就这一题目进行了修正。因此，关于这个题目，附件 F.2是无效的。

5.4.1.12.2　生成程序实例

具体正文文法

〈生成程序实例集〉 :: =
　　{〈生成程序实例〉[〈结束〉][ADDING]}$^{+}$
〈生成程序实例〉 :: =
　　〈生成程序标识符〉(〈生成程序实参表〉)
〈生成程序实参表〉 :: =
　　〈生成程序实参〉{,〈生成程序实参〉} *
〈生成程序实参〉 :: =
　　〈扩展类别〉
　　| 〈字面值标记〉
　　| 〈运算符名〉
　　| 〈基本项〉

如果〈生成程序参量〉是 TYPE 类，则对应的〈生成程序实参〉必须是〈扩展类别〉。

如果〈生成程序参量〉是 LITERAL 类，则对应的〈生成程序实参〉必须是一个〈字面值标记〉。

如果一个〈字面值标记〉是一个〈名字类字面值〉，它可以用作为〈生成程序实参〉，当且仅当对应的〈生成程序形式名〉不在〈公理〉中出现、或不在〈生成程序正文〉中〈特性表达式〉的〈字面值映象〉中出现。

如果〈生成程序参量〉是 OPERATOR 类，则对应的〈生成程序实参〉必须是一个〈运算符名〉。

如果〈生成程序参量〉是 CONSTANT 类，则对应的〈生成程序实参〉必须是一个〈基本项〉。

如果〈生成程序实参〉是一个〈生成程序形式名〉，则〈生成程序形式名〉的类必须与〈生成程序实参〉的类相同。

语义

在扩展特性或在生成程序正文中可以用一个生成程序实例来表示由该生成程序标识符所标识的正文的实例。从生成程序正文可以构成包括字面值、运算符和公理的实例正文，构成的方法是：

a) 用生成程序实在参量取代生成程序参量，还

b) 用下述名字取代生成程序的名字：

i) 如果此生成程序实例是处于一部分类型定义或同义类型定义中，则由该部分类型定义或同义类型定义所规定的类别的标识来取代。否则

ii) 在生成程序实例是处于一个生成程序中的情况下，就由那个生成程序的名字来取代。

字面值的实例正文是从生成程序正文的特性表达式中的字面值例示得到的正文，要略去关键字 LITERALS。

运算符的实例正文是从生成程序正文的特性表达式中的运算符例示得到的正文，要略去关键字 OPERATORS。

公理的实例正文是从生成程序正文的特性表达式中的公理例示得到的正文，要略去关键字 AXIOMS。

当在生成程序实例集合的表中有不止一个生成程序实例时，多个字面值（运算符和公理）的实例正文是这样构成的，即将所有生成程序的字面值（运算符、公理）的实例正文以它们在表中出现的次序拼接起来构成。

多个字面值的实例正文是多个字面值组成的一个表，这些字面值是属于包围它的部分类型定义、同义类型定义或生成程序定义的特性表达式的，而这些定义出现在特性表达式中显式地声明字面值表之前。这就是说、如果已经规定了次序，由生成程序实例集合所定义的那些字面值将按照它们被例示的次序，并安排在任何其它字面值之前。

运算符和公理的实例正文被分别加到包围它的部分类型定义、同义类型定义或生成程序定义的运算符表和公理表中。

当把实例正文加入到一个特性表达式中时，如果有必要，应该增加关键字 LITERALS、OPERATORS 和 AXIOMS，以产生正确的具体文法。

模型

在一个生成程序实例之后，对应于此实例的抽象语法就确定了，它们的关系由在实例处的实例正文来确定。

举例

```
NEWTYPE boolbag bag (Boolean)
    ADING
    OPERATORS
        yesvote: boolbag-> Boolean;
    AXIOMS
        yesvote (b) ==count (True, b) > count (False, b);
ENDNEWTYPE boolbag;
```

5.4.1.13 同义词

一个同义词给一个基本表达式起一个名字，它表示一种类别的一个值。

具体正文文法

〈同义词定义〉 :: =
SYNONYM 〈同义词名〉[〈类别〉] = 〈基本表达式〉
| 〈外部同义词定义〉

以上定义中可选择的〈外部同义词定义〉在§4.3.1节中介绍。

如果〈基本表达式〉的类别不能唯一地确定，则在〈同义词定义〉中必须规定一个类别。

用〈类别〉标识的类别必须是〈基本表达式〉可以具有的类别。

此〈基本表达式〉不允许引用由此〈同义词定义〉规定的同义词，无论是直接地引用或间接地（通过另一个同义词）引用都不允许。

语义

同义词所表示的值由出现该同义词定义的上下文来确定。

如果基本表达式的类别不能在此同义词的上下文中唯一地确定，则此类别由〈类别〉给出。

一个同义词有一个值，它就是在此同义词定义中的基本项的值。

一个同义词有一个类别，此类别就是此同义词定义中基本项的类别。

模型

如§5.4.2.2节所定义，具体语法中的〈基本表达式〉表示抽象语法中的一个基本项。

如果规定了一个〈类别〉，则此〈基本表达式〉的结果就具有这个类别。〈基本表达式〉表示抽象语法中的一个基本项，此抽象语法有一个运算符标识符，其名字和变元类别与具体语法给出的一样，并且，其结果类别等于在具体语法中规定的类别。

5.4.1.14　名字类字面值

名字类字面值是一个简化形式，用来书写由一个正规表达式定义的字面值名的集合(可以是无限集)。

具体正文文法

〈名字类字面值〉　::　=
　　NAMECLASS〈正规表达式〉
〈正规表达式〉　::　=
　　〈部分正规表达式〉
　　{ [OR]〈部分正规表达式〉} *
〈部分正规表达式〉　::　=
　　〈正规元素〉[〈自然数字面值名〉 | + | *]
〈正规元素〉　::　=
　　(〈正规表达式〉)
　| 〈字符串字面值〉
　| 〈正规区间〉
〈正规区间〉　::　=
　　〈字符串字面值〉 : 〈字符串字面值〉

由〈名字类字面值〉构成的名字必须满足字面值的一般静态条件（参见§5.2.2节），并满足与名字有关的词法规则（参见§2.2.1节）或者满足对〈字符串字面值〉的具体语法（参见§5.4.1.2节）。

在一〈规则区间〉中的两个〈字符串字面值〉的长度必须都是1，它们还必须是由 character 类别（参见§5.6.2节）定义的字面值。

语义

名字类字面值是规定字面值标记的一种可选用的方法。

模型

一个名字类字面值等价于名字的一个集合，规定这个名字集合应满足由〈正规表达式〉规定的语法。在抽象语法中，此名字类字面值等价于这个名字集合。

如果一个〈正规表达式〉是不带有 OR 的一连串〈部分正规表达式〉，则它规定了：这些名字的构成包括第一个〈部分正规表达式〉定义的字符，后面跟有由第二个〈部分正规表达式〉定义的字符。

当在两个〈部分正规表达式〉之间规定了一个 OR 时，则这些名字从第一个〈部分正规表达式〉构成，或是从第二个〈部分正规表达式〉构成。注意：与简单的顺序比较 OR 把项结合得更紧密，所以

NAMECLASS 'A'　'0'　OR　'1'　'2' ;

等价于

NAMECLASS 'A' ('0' OR '1') '2' ;

这个名字类定义了两个字面值 A02，A12。

如果一个〈正规元素〉后跟有〈自然数字面值名〉，则〈部分正规表达式〉等价于被重复的〈正规元素〉，重复的次数由〈自然数字面值名〉规定。

例如

NAMECLASS 'A' ('A' OR 'B') 2

定义了名字 AAA、AAB、ABA 和 ABB。

如果〈正规元素〉后跟有一个“*”号，则〈部分正规表达式〉等价于被重复0次或多次的〈正规元素〉。

例如

NAMECLASS 'A' ('A' OR 'B')*

定义了名字 A，AA，AB，AAA，AAB，ABA，ABB，AAAA，…等等。

如果〈正规元素〉后跟有一个″ +″ 号，则〈部分正规表达式〉等价于被重复一次或多次的〈正规元素〉。

例如

NAMECLASS 'A' ('A' OR 'B') +

定义了名字 AA，AB，AAA，AAB，ABA，ABB，AAA，…等等。

如果一个〈正规元素〉是一个用括号括起来的〈正规表达式〉，则它定义了由该〈正规表达式〉确定的字符序列。

如果一个〈正规元素〉是一个〈字符串字面值〉，则它定义了在该字符串字面值（略去引号）中给出的字符序列。

如果一个〈正规元素〉是一个〈正规区间〉，则它定义了由〈正规区间〉规定的所有字符为可选择字符序列。根据字符类别的定义（参见§5.6.2节），由〈正规区间〉定义的字符是大于或等于第一个字符，小于或等于第二个字符的所有字符。例如

'a' : 'f'

决定了选择对象是 'a' 或 'b' 或 'c' 或 'd' 或 'e' 或 'f'。

如果所定义名字的次序是重要的话（例如，如果规定了 ORDERING），则认为规定了这些名字的顺序，使得它们根据字符串类别的排序按照字母顺序排序。如果两个名字开头的一些字符相同，但具有不同的长度，则较短的名字被认为是先定义的。

5.4.1.15 字面值映象

字面值映象是用来规定从字面值到值的映象的简化形式。

具体正文文法

〈字面值映象〉::=

MAP〈字面值等式〉{〈结束〉〈字面值等式〉} * [〈结束〉]

〈字面值等式〉::=

〈字面值量化条件〉

(〈字面值公理〉{〈结束〉〈字面值公理〉} * [〈结束〉])

〈字面值公理〉::=

〈等式〉

| 〈字面值等式〉

〈字面值量化条件〉::=

FOR ALL〈值名〉{,〈值名〉} * IN〈扩展类别〉LITERALS

〈拼字项〉::=

SPELLING (〈值标识符〉)

〈字面值映象〉和〈拼字项〉规则不是数据核的一部分，但分别出现在§5.2.1节和§5.2.3节中的〈特性表达式〉规则和〈项〉规则之中。

语义

字面值映象是用于定义其范围遍及一个类别的所有字面值的多个(可能无限个)公理的简化形式。字面值映象允许一个类别的字面值映象成此类别的值。当此类别含有许多(或无限)个值时，字面值映象是唯一的实际方法用来规定每一个字面值所对应的值。

在字面值映象中采用拼字项机制来引用包含字面值的拼字的字符串。这个机制允许用字符串运算符来定义字面值映象。

模型

〈字面值映象〉是一组〈公理〉的简化形式。这组〈公理〉是从〈字面值映象〉中的〈字面值等式〉推导出来的。在此推导过程中用到的〈等式〉是在〈字面值公理〉规则中的所有〈等式〉。在每一个〈等式〉中，由〈字面值量化条件〉中的〈值名〉所规定的〈值标识符〉被取代。在每一个导出的〈等式〉中，同一个〈值标识符〉的每一次出现要由相同的〈字面值运算符标识符〉所取代，它具有此〈字面值量化条件〉的〈类别〉。导出的〈公理〉组含有所有可以以这种方式导出的〈等式〉。

在同一〈部分类型定义〉中，导出的关于〈字面值等式〉的〈公理〉被加到在关键字 AXIOMS 之后和在关键字 MAP 之前定义的〈公理〉(若有的话)之中。

例如

```
NEWTYPE abc LITERALS 'A', b, 'c';
    OPERATORS
        " >" :abc, abc -> Boolean;
        " +" :abc, abc -> Boolean;
    MAP FOR ALL x, y IN abc LITERALS
        (x<y=>y+x);
    ENDNEWTYPE abc;
```

是从下面的具体语法导出的具体语法。

```
NEWTYPE abc LITERALS 'A', b, 'c';
    OPERATORS
        " <": abc, abc -> Boolean;
        " +": abc, abc -> Boolean;
AXIOMS
```

```
'A'   <  'A'    =>   'A'    +  'A';
'A'   <  b      =>   b      +  'A';
'A'   <  'c'    =>   'c'    +  'A';
b     <  'A'    =>   'A'    +  b;
b     <  b      =>   b      +  b;
b     <  'c'    =>   'c'    +  b;
'c'   <  'A'    =>   'A'    +  'c';
'c'   <  b      =>   b      +  'c';
'c'   <  'c'    =>   'c'    +  'c';
ENDNEWTYPE abc;
```

如果〈字面值量化条件〉包含一个或多个〈拼字项〉，则可用字符串字面值替换〈拼字项〉(参见§5.6.3节)。

如果一个〈拼字项〉的〈字面值 运算符标识符〉的〈字面值标记〉是一个〈字面值 运算符名〉(参见§5.2.2节)，则该〈拼字项〉是从〈字面值 运算符标识符〉导出的一个大写体字符串的简化形式。此字符串含有该〈字面值 运算符标识符〉的〈字面值 运算符名〉的大写体拼字。

如果一个〈拼字项〉之〈字面值 运算符标识符〉的〈字面值标记〉是一个〈字符串字面值〉(参见§5.2.2节和§5.4.1.2节)。则该〈拼字项〉是从此〈字符串字面值〉导出的字符串的简化形式。此字符串含有该〈字符串字面值〉的拼字。

在含有此〈拼字项〉的〈字面值等式〉被加上扩展之后，此字符串被用来取代〈值标识符〉。

例如

```
NEWTYPE abc LITERALS 'A', Bb, 'c';
    OPERATORS
        " <": abc, abc->Boolean;
    MAPFOR ALL x, y IN abc LITERALS
        SPELLING (x) <SPELLING (y) => x<y;
    ENDNEWTYPE abc;
```

是从下面的具体语法导出的具体语法。

```
NEWTYPE abc LITERALS 'A', Bb, 'c';
    OPERATORS
        " <": abc, abc-> Boolean;
    AXIOMS /* 注意， 'A'， Bb， 'c' 具有局部类别 abc */
        /* ''A''', 'BB' 和 ''C''' 这些字面值容易混淆，应该用字符
           串标识符来描述。—为了简短，这里就省略了 */
    ''A'''   <  ''A'''   => 'A'   <  'A';
    ''A'''   <  'BB'     => 'A'   <Bb;
    ''A'''   <  ''c'''   => 'A'   <  'c';
    'BB'     <  ''A'''   => Bb    <  'A';
    'BB'     <  'BB'     => Bb    <Bb;
    'BB'     <  ''c'''   => Bb    <  'c';
    ''c'''   <  ''A'''   => 'c'   <  'A';
    ''c'''   <  'BB'     => 'c'   <Bb;
```

ENDNEWTYPE abc;

〈字面值公理〉中的每一个〈非量化等式〉必须含有一个〈拼字项〉或一个〈字面值运算符标识符〉。

〈拼字项〉必须位于〈字面值映象〉之中。

〈拼字项〉中的〈值标识符〉必须是由一个〈字面值量化〉所定义的〈值标识符〉。

5.4.2 数据的使用

下面规定了在表达式中是如何解释数据类型、类别、字面值、运算符和同义词的。

5.4.2.1 表达式

表达式是字面值、运算符、变量访问、条件表达式以及命令运算符。

抽象文法

表达式 = 基本表达式 |
主动表达式

一个表达式如果含有一个主动初始数（参见§5.5节），则它是一个主动表达式。

不含有主动初始数的表达式是基本表达式。

具体正文文法

为使描述简单，对于基本表达式与主动表达式没有区分不同的具体语法。关于〈表达式〉的具体语法在下面的§5.4.2.2节中给出。

语义

一个表达式被解释为基本表达式或主动表达式的值。如果此值是一个错误的话，则系统未来的行为是不确定的。

表达式的类别就是基本表达式或主动表达式的类别。

5.4.2.2 基本表达式

抽象文法

基本表达式 :: 基本项

关于基本项的静态条件也适用于基本表达式。

具体正文文法

〈基本表达式〉::=
〈基本表达式〉

〈表达式〉::=

〈运算数0〉
| 〈子表达式〉 =＞〈运算数0〉
〈子表达式〉::=
〈表达式〉
〈运算数0〉::=
〈运算数1〉
| 〈子运算数0〉{OR|XOR}〈运算数1〉
〈子运算数0〉::=
〈运算数0〉
〈运算数1〉::=
〈运算数2〉
| 〈子运算数1〉 AND 〈运算数2〉
〈子运算数1〉::=
〈运算数1〉
〈运算数2〉::=
〈运算数3〉
| 〈子运算数2〉{=|/=|>|>=|<|<=|IN}〈运算数3〉
〈子运算数2〉::=
〈运算数2〉
〈运算数3〉::=
〈运算数4〉
| 〈子运算数3〉{+|-|〃}〈运算数4〉
〈子运算数3〉::=
〈运算数3〉
〈运算数4〉::=
〈运算数5〉
| 〈子运算数4〉{*|/|MOD|REM}〈运算数5〉
〈子运算数4〉::=
〈运算数4〉
〈运算数5〉::=
[-|NOT]〈初始数〉
〈初始数〉::=
〈基本初始数〉
| 〈主动初始数〉
| 〈扩展初始数〉
〈基本初始数〉::=
〈字面值标识符〉
| 〈运算符标识符〉(〈基本表达式表〉)

| (〈基本表达式〉)
| 〈条件基本表达式〉

〈扩展初始数〉::=
〈同义词〉
| 〈标引初始数〉
| 〈字段初始数〉
| 〈结构初始数〉

〈基本表达式表〉::=
〈基本表达式〉{,〈基本表达式〉} *

〈运算符标识符〉::=
〈运算符标识符〉
| [〈限定符〉]〈引用文运算符〉

一个不包含任何〈主动初始数〉的〈表达式〉在抽象语法中表示一个基本表达式。一个〈基本表达式〉必须不含有〈主动初始数〉。

如果〈表达式〉是一个带有〈运算符标识符〉的〈基本初始数〉，并且，〈运算符标记〉的一个〈变元类别〉是一个〈同义类型〉的话，则对于那个同义类型（在§5.4.1.9.1节中规定的）的范围检验应作用于对应变元值。范围检验的值必须是 True。

如果〈表达式〉是一个带有〈运算符标识符〉的〈基本初始数〉，而且该〈运算符标记〉的〈结果类别〉是一个〈同义类型〉，则对于那个同义类型的范围检验（在§5.4.1.9.1节中规定的）应作用于该结果值。范围检验的值必须是 True。

如果一个〈表达式〉含有一个〈扩展初始数〉（就是一个〈同义词〉、〈标引初始数〉、〈字段初始数〉或〈结构初始数〉），这个〈扩展初始数〉将在考虑与抽象语法的关系之前，在具体语法层次被取代，如分别在§5.4.2.3、§5.4.2.4、§5.4.2.5以及§5.4.2.6节中所定义的那样。

一个〈引用文运算符〉之前的任选的〈限定符〉与抽象语法的关系，和一个〈运算符标识符〉的〈限定符〉与抽象语法的关系相同（参见§5.2.2节）。

语义

一个基本表达式被解释为由语法上等效于此基本表达式的基本项所表示的值。

通常，没有必要或理由在基本项和基本项的值之间进行区分。例如，关于单位整数值的基本项可以写成“1”。通常可以有多个基本项表示同一个值，例如，整数基本项“0+1”、“3-2”和“(7+5)/12”，一般考虑将基本项的一个简单形式（在这个情况是“1”）用来表示此值。

一个基本表达式具有一个类别，此类别就是等效基本项的类别。

一个基本表达式具有一个值，此值就是等效基本项的值。

5.4.2.3 同义词

具体正文文法

〈同义词〉::=
〈同义词标识符〉
| 〈外部同义词〉

可选择的〈外部同义词〉已在§4.3.1节中描述。

语义

同义词 synonym 是一个简化写法，用来表示在其它地方定义的某个表达式。

模型

一个〈同义词〉代表由该〈同义词标识符〉所标识的〈同义词定义〉所定义的〈基本表达式〉。根据〈同义词定义〉的上下文，在〈基本表达式〉中用到的一个〈标识符〉代表抽象语法中的一个标识符。

5.4.2.4 标引初始数

标引初始数是一个简化语法符号，它可以用来表示一个“数组”值的“标引”。然而，除特殊的语法形式外，标引初始数没有特殊的特性，并表示以初始数为参量的一个运算符。

具体正文文法

〈标引初始数〉::=
〈初始数〉(〈表达式表〉)

语义

标引表达式表示 Extract! 运算符的应用。

模型

一个〈初始数〉后面跟有一个用括号括起来的〈表达式表〉是下列具体语法的派生具体语法。

Extract!(〈初始数〉,〈表达式表〉)并且这也是一个合法表达式，即使在有关表达式的具体语法中 Extract! 也是不允许作为一个运算符名字的。根据§5.4.2.2节，其抽象语法要由这个具体表达式确定。

5.4.2.5 字段初始数

字段初始数是一个简化语法符号，它可以用来表示从“结构”中选择“字段”。然而，除了特殊语法形式以外，字段初始数没有特别的特性，它表示以初始数作为参量的一个运算符。

具体正文文法

〈字段初始数〉::=
〈初始数〉〈字段选择〉

〈字段选择〉::=

! 〈字段名〉

| (〈字段名〉{,〈字段名〉} *)

字段名必须是一个具有此初始数的类别的字段名。

语义

一个字段初始数表示结构类别的一个字段抽取运算符的应用。

模型

形式

〈初始数〉(〈字段名〉)

是下面形式的派生语法。

〈初始数〉!〈字段名〉

另一个形式

〈初始数〉(〈第一字段名〉{,〈字段名〉} *)

是以下形式的派生语法。

〈初始数〉!〈第一字段名〉{!〈字段名〉} *

这里保持了字段名的次序。

形式

〈初始数〉!〈字段名〉

是以下形式的派生语法。

〈字段抽取运算符名〉(〈初始数〉)

这里字段抽取运算符名是从字段名及" Extract!" 按此次序拼接起来构成的。例如

s! f1

是从

f1Extract! (s)

派生的语法，并且，即使在有关表达式的具体语法中 f1Extract! 也是不允许用作为一个运算符名字的，但这也被认为是一个合法的表达式。根据 § 5.4.2.2 节，其抽象语法要从这个具体表达式来确定。

当规定了一个运算符是针对某一类别的，使得当 "name" 与 S 的类别的合法字段名相同时，

Extract! (s, name)

是一个合法项的情况下，那么一个初始数

s (name)

是对于

Extract! (s, name)

的派生具体语法，并且，字段的选择必须写成

s! name

5.4.2.6 结构初始数

具体正文文法

〈结构初始数〉::=
[〈限定符〉](.〈表达式表〉.)

语义

结构初始数表示一个结构类别的值，它由此结构的每一个字段的表达式构成。

形式

(.〈表达式表〉.)

是下面形式的派生语法。

Make!(〈表达式表〉)

这被认为是一个合法的基本表达式，虽然 Make! 在有关基本表达式的具体语法中不允许用作为一个运算符名字。根据§5.4.2.2节，其抽象语法要从这个具体基本表达式来确定。

5.4.2.7 条件基本表达式

具体正文文法

〈条件基本表达式〉::=
IF〈布尔基本表达式〉
THEN〈结果基本表达式〉
ELSE〈替换基本表达式〉
FI

〈结果基本表达式〉::=
〈基本表达式〉

〈替换基本表达式〉::=
〈基本表达式〉

〈条件基本表达式〉表示在抽象语法中的一个基本表达式。如果此〈布尔基本表达式〉代表 True，则此基本表达式由〈结果基本表达式〉来代表，否则，它由〈替换基本表达式〉来代表。

〈结果基本表达式〉的类别必须与〈替换基本表达式〉的类别相同。

语义

条件基本表达式是一个基本初始数，它或被解释为结果基本表达式，或被解释为替换基本表达式。

如果〈布尔基本表达式〉具有值 True，则不去解释〈替换基本表达式〉。如果〈布尔基本表达式〉具有值 False，则不解释〈结果基本表达式〉。如果被解释的〈基本表达式〉具有一个错误值，则系统未来的行为是不确定的。

一个条件基本表达式的类别就是该结果基本表达式的类别（而且也是替换基本表达式的类别）。

5.5 带变量数据的使用

本节规定在进程和过程中所声明的数据和变量的使用规则，以及规定从基础的系统获取值的命令运算符的使用规则。

一个变量有一个类别以及一个关联的值，此值属于这个类别。通过对一个变量赋于一个新的值，可以改变关联于此变量的值。在一个表达式中可以通过访问一个变量，使用关联于此变量的值。

任何一个表达式，如果含有一个变量，就被认为是“主动的”，这是因为通过解释表达式而得到的值可以根据最后赋给该变量的值而改变。

5.5.1 变量和数据定义

具体正文文法

〈数据定义〉::=
{ 〈部分类型定义〉
| 〈同义类型定义〉
| 〈生成程序定义〉
| 〈同义词定义〉} 〈结束〉

如果数据定义是分别在§5.2.1节和§5.4.1.9节中定义的一个〈部分类型定义〉或〈同义类型定义〉，则它构成数据类型定义的一部分。〈生成程序定义〉规则和〈同义词定义〉规则分别在§5.4.1.12节和§5.4.1.13节中定义。

关于引入进程变量和关于过程参数变量的语法分别在§2.5.1.1节和§2.3.4节中给出。在过程中定义的变量不能够被透露。

语义

一个数据定义既可用于定义一个数据类型的一部分，也可用于定义一个表达式的同义词，这在§5.2.1节，§5.4.1.9节或§5.4.1.13节中有进一步的规定。

当一个变量被创建时，它含有一个称为“未定义”的特殊值，此值不同于那个变量的类别的任何其它的值。

5.5.2 访问变量

下面规定了如何解释一个含有变量的表达式。

5.5.2.1 主动表达式

抽象文法

主动表达式 = 变量访问|
条件表达式|
运算符应用|
命令运算符

具体正文文法

〈主动表达式〉::=

〈主动表达式〉

〈主动初始数〉::=

〈变量访问〉

| 〈运算符应用〉

| 〈条件表达式〉

| 〈命令运算符〉

| (〈主动表达式〉)

| 〈主动扩展初始数〉

〈主动扩展初始数〉::=

〈主动扩展初始数〉

〈表达式表〉::=

〈表达式〉{,〈表达式〉}*

为简明起见，关于〈主动表达式〉的具体语法在§5.4.2.2节中作为〈表达式〉给出。如果一个〈表达式〉含有一个〈主动初始数〉，这个〈表达式〉就是一个〈主动表达式〉。

同样为简明起见，关于〈主动扩展初始数〉的具体语法在§5.4.2.2节中作为〈扩展初始数〉给出。如果〈扩展初始数〉含有一个〈主动初始数〉，则它就是一个〈主动扩展初始数〉。如同在§5.4.2.3、§5.4.2.4、§5.4.2.5和§5.4.2.6节中所规定的，在考虑与抽象语法的关系之前，对于一个〈扩展初始数〉应在具体语法层次上进行替换。

语义

主动表达式是一个其值取决于系统的当前状态的表达式。

主动表达式的类别就是其等效基本项的类别。

一个主动表达式具有一个值，此值等效于解释此主动表达式时的基本项。

模型

每次解释主动表达式时，通过寻找等效于此主动表达式的基本项来确定此主动表达式的值。这个基本项从一个基本表达式确定，这个基本表达式的构成方法是：把该主动表达式中的每一个主动初始数用等于那个主动初始数的值的基本项来取代。此主动表达式的值与该基本表达式的值相同。

主动表达式中，每一个运算符的解释顺序根据§5.4.2.2节中给出的具体语法来确定，或者在有二义的情况下，按照从左至右的顺序来进行解释。在一个主动表达式表或表达式表中，对表的每一个元素要按照从左至右的顺序来解释。

5.5.2.2 变量访问

抽象文法

变量访问 = 变量标识符

具体正文文法

〈变量访问〉::=

〈变量标识符〉

语义

对变量访问的解释就是给出与所标识的这个变量关联的值。

变量访问所具有的类别就是由此变量访问所标识的变量的类别。

变量访问具有一个值，这个值就是最后关联于此变量的值，或者，如果这个值是“未定义”，则出现一个错误。如果变量访问的值是一个错误，则系统未来的行为是不确定的。

5.5.2.3 条件表达式

条件表达式是这样一种表达式，它或被解释为结果表达式、或被解释为替换表达式。

抽象文法

条件表达式 :: 布尔表达式
结果表达式
替换表达式
布尔表达式 = 表达式
结果表达式 = 表达式
替换表达式 = 表达式

结果表达式的类别必须与替换表达式的类别相同。

具体正文文法

〈条件表达式〉::=
IF〈布尔主动表达式〉
THEN〈结果表达式〉
ELSE〈替换表达式〉
FI
| IF〈布尔表达式〉
THEN〈主动结果表达式〉
ELSE〈替换表达式〉
FI
| IF〈布尔表达式〉
THEN〈结果表达式〉
ELSE〈主动选择表达式〉
FI
〈结果表达式〉::=
〈表达式〉
〈替换表达式〉::=
〈表达式〉

〈条件表达式〉与〈条件基本表达式〉有区别是因为在〈条件表达式〉中包含有〈主动表达式〉。

语义

条件表达式被解释为：条件的解释后面或是跟着结果表达式的解释，或是跟着替换表达式的解释。仅当条件具有值 True 时，才去解释结果表达式，这样，如果条件具有值 False，则只有在替换表达式是错误时，系统未来的行为才是不确定的。类似地，仅当条件具有值 False 时，才去解释替换表达式，如果条件具有值 True，则只有在结果表达式是错误时，系统未来的行为才是不确定的。

条件表达式的类别，与结果表达式和替换表达式的类别相同。

条件表达式具有一个值，如果条件为 True，此值是结果表达式的值；如果条件为 False 则此值是替换表达式的值。

5.5.2.4　运算符应用

运算符应用是一个运算符的应用，其中有一个或多个实在变元是主动表达式。

抽象文法

运算符应用　　::　　运算符标识符
　　表达式+

如果运算符标记的一个变元类别是同义类型，并且表达式表中的对应表达式是基本表达式，则把在 § 5.4.1.9.1 节中规定的范围检验应用于表达式之值时，必须得到 True。

具体正文文法

〈运算符应用〉::=
　　〈运算符标识符〉(〈主动表达式表〉)

〈主动表达式表〉::=
　　〈主动表达式〉[,〈表达式表〉]
　| 〈基本表达式〉,〈主动表达式表〉

〈运算符应用〉区别于语法上类似的〈基本表达式〉之处在于：在括起来的〈表达式〉表中，有一个〈表达式〉是一个〈主动表达式〉。如果所有括起来的〈表达式〉都是〈基本表达式〉，则此结构表示一个基本表达式，见 § 5.4.2.2 节中的定义。

语义

运算符应用是一个主动表达式，它具有等效于此运算符应用的基本项的值。该等效的基本项象在 § 5.5.2.1 节中那样来确定。

在解释运算符应用的那些表达式时，应按照解释此运算符之前所给出的次序进行解释。

如果运算符标记的一个变元类别是一个同义类型，并且在主动表达式表中对应的表达式是一个主动表达式，则应该对此表达式的值进行在 § 5.4.1.9.1 节中规定的范围检验。如果在解释时，范围检验的值为 False，则系统出错，而系统的未来行为是不确定的。

如果运算符标记的结果类别是一个同义类型，则要按照§5.4.1.9.1节中规定的范围检验来检验此运算符应用的值。如果在解释时范围检验为False，则系统出错，并且系统未来的行为是不确定的。

5.5.3 赋值语句

抽象文法

赋值语句 :: 变量标识符
表达式

变量标识符的类别与表达式的类别必须相同。

如果声明此变量是一个同义类型，并且此表达式是一个基本表达式，则对此表达式按照§5.4.1.9.1节中的规定进行范围检验，必须得出True。

具体正文文法

〈赋值语句〉::=
〈变量〉：=〈表达式〉
〈变量〉::=
〈变量标识符〉
| 〈标引变量〉
| 〈字段变量〉

如果变量是一个〈变量标识符〉，则具体语法中的〈表达式〉表示抽象语法中的〈表达式〉。其它形式的〈变量〉，即〈标引变量〉和〈字段变量〉都是派生语法，并且，抽象语法中的〈表达式〉要从下面的§5.5.3.1和§5.5.3.2节中定义的等效具体语法中得出。

语义

赋值语句被解释为：它建立起一种联系，把在此赋值语句中标识的变量与此赋值语句中的表达式的值联系起来。原先与变量联系的值将失去。

如果声明变量是一个同义类型，并且表达式是主动表达式，则对此表达式按照§5.4.1.9.1节中的规定进行范围检验。如果这个范围检验等于False，则赋值出错，而系统未来的行为不确定。

5.5.3.1 标引变量

标引变量是一种简化语法表示，它可以用来表示“数组”元素的标引。然而，除了特殊语法形式外，一个标引主动初始数没有特殊的特性，并且，表示以主动初始数作为参量的一个运算符。

具体正文文法

〈标引变量〉::=
〈变量〉(〈表达式表〉)

对一个名为Modify！的运算符必须要有一个合适的定义。

语义

标引变量表示把一个值赋给一个变量。这要通过使用 Modify! 运算符去访问一个变量以及作用于在标引变量中给出的表达式。

模型

此具体语法形式

〈变量〉(〈表达式表〉) ::= 〈表达式〉

是下面语法形式

〈变量〉 ::=Modify! (〈变量〉,〈表达式表〉,〈表达式〉) 的派生具体语法。这里同一个〈变量〉要重复写出。此正文被认为是一个合法的赋值语句，即使在有关表达式的具体语法中 Modify! 不允许作为一个运算符名。这个〈赋值语句〉的抽象语法要根据上面 § 5.5.3 节确定。

在应用进口的模型之前必须先应用标引变量的模型（参见 § 4.13 节）。

5.5.3.2 字段变量

字段变量是一个简化符号，用来把一个值赋给一个变量，使得所改变的仅仅是那个变量的某一个字段中的值。

具体正文文法

〈字段变量〉::=

〈变量〉〈字段选择〉

对于一个名为 Modify! 的运算符，必须有一个适当的定义。通常，这个定义隐含在一个结构化的类别定义中。

语义

字段变量表示把一个值赋给一个字段，通过应用一个字段修改运算符来进行赋值。

模型

括起来的字段选择，是用!〈字段名〉来进行字段选择的派生语法。见 § 5.4.2.5 节中的定义。

此具体语法形式

〈变量〉!〈字段名〉 ::= 〈表达式〉

是下面语法形式的派生具体语法。

〈变量〉 ::= 〈字段 修改 运算符名〉(〈变量〉,〈表达式〉)

a) 同一个〈变量〉要重复写出，并且

b) 〈字段 修改 运算符名〉是把字段名与“Modify!”拼接起来构成的，并且

c) 即使此〈字段 修改 运算符名〉在有关表达式的具体语法中不允许作为一个运算符名字，这个语法形式仍被认为是一个合法的赋值语句。

如果在字段选择中有多于一个〈字段名〉，则可以模仿上述方法，从右至左依次扩展每一个!〈字段名〉，并将〈字段变量〉的剩余部分看成是一个〈变量〉。例如

```
var! fielda! fieldb: =expression;
/*变量! 字段a! 字段b: =表达式; */
```

第一次扩展为

```
var! fielda: =fieldbModify! (var! fielda, expression);
```

然后扩展为

```
var: =fieldaModify! (var, fieldbModify! (var! fielda, expression));
```

根据这一方法构成的〈赋值语句〉，其抽象语法根据上面的§5.5.3节确定。

5.5.3.3 缺省赋值

缺省赋值是一种简化形式，用来把同样的值赋给某个特定类别的所有变量，在这些变量被创建后，立即对它们赋值。

具体正文文法

〈缺省赋值〉::=

DEFAULT〈基本表达式〉[〈结束〉]

一个〈部分类型定义〉或〈同义类型定义〉只能含有不超过一个〈缺省赋值〉。(这防止了在生成程序实例中出现多重赋值)。

语义

可以任选地(即可选用也可不选用)把缺省赋值加入到一个类别的特性表达式中。缺省赋值规定：任何(在部分类型定义或同义类型定义中引入的)变量被声明具有该类别，就立即被赋予该基本表达式的值。

如果没有缺省赋值，则当一个变量被声明时，它的值就是未定义的。

当声明一个变量时，可以赋予它另外一个值，办法是在变量声明中包括一个显式的赋值语句。

缺省赋值是不被继承的。

模型

具体语法形式

DEFAULT〈基本表达式〉

用在引入类别s的特性表达式中，将意味着把此〈基本表达式〉的值赋给一个变量。这个赋值句在声明此变量之后立即被解释，也就是解释应在同一个进程或过程中的任何显式规定的动作被解释之前进行。例如，如果

```
DEFAULT 2 * dnumber
```

为类别s给定的缺省赋值句，并且在具体语法中有一个声明

```
DCL v s;
```

则有一个隐含赋值句

```
v: =2 * dnumber;
```

如果在该声明中还有一个〈初始值〉，则要在〈缺省赋值〉中的〈基本表达式〉赋给该变量之后，才把此〈初始值〉赋给该变量。

所隐含的赋值语句具有一个〈赋值语句〉与抽象语法的普通关系（参见§5.5.3）。

如果为一个〈数据定义〉规定了〈缺省赋值〉，则其〈类别〉（代表一个同义类型或类别）具有一个缺省值（即用于缺省赋值的值），此值就是〈缺省赋值〉的〈基本表达式〉的值。如果在〈同义类型定义〉中没有给出〈缺省赋值〉，则当（在同义类型定义中给出的）父辈类别标识符（用于标识一个同义类型或类别）有一个缺省值时，该同义类型也将具有一个缺省值。

对于〈同义类型定义〉，当且仅当在§5.4.1.9.1节中规定的范围检验对于该缺省值得出True时，才对该缺省赋值进行解释。这就是说，对同义类型的每一个变量，有一个隐含的判定，其形式为

```
DECISION〈范围检验〉
 (True) : 〈缺省赋值〉
ELSE : ENDDECISION。
```

5.5.4 命令运算符

命令运算符从基础系统状态获得值。

抽象文法

```
命令运算符      =    Now 表达式 |
                     PId 表达式 |
                     视见表达式 |
                     定时器活跃表达式
```

具体正文文法

```
〈命令运算符〉::=
          〈NOW 表达式〉
        | 〈进口表达式〉
        | 〈PId 表达式〉
        | 〈视见表达式〉
        | 〈定时器活跃表达式〉
```

可选择的〈进口表达式〉在§4.13节中定义。

命令运算符是一些表达式，或是用来检验定时器是否活跃，或者用于访问系统时钟，或者访问某个进程的PId值，或是访问进口变量。

5.5.4.1 NOW

抽象文法

```
NOW 表达式      ::      ()
```

具体正文文法

〈NOW 表达式〉::=
NOW

语义

NOW 表达式是一个访问系统时钟变量，以确定绝对系统时间的表达式。

NOW 表达式询问系统时钟的当前值，这个时钟给出时刻值。计时的起点和时间的单位是依赖于系统的。在同一跃迁中，究竟 NOW 的两次出现是否要给出相同的时刻值，这依赖于系统设计。

NOW 表达式具有时间类别。

5.5.4.2 IMPORT 表达式

具体正文文法

进口表达式的具体语法在§4.3 节中定义。

语义

除在§4.13 节中定义的语义外，进口表达式被解释为一个变量访问(参见§5.5.2.2 节)，访问该进口表达式的隐式变量。

模型

进口表达式具有关于值的进口的隐含的语法，这在§4.13 节中已有规定。并且它还具有一个隐含的变量访问，访问在出现〈进口表达式〉的上下文中的进口的隐式变量。

5.5.4.3 PId 表达式

抽象文法

PId 表达式	=	自身表达式| 父辈表达式| 子孙表达式| 发送者表达式
自身表达式	::	()
父辈表达式	::	()
子孙表达式	::	()
发送者表达式	::	()

具体正文文法

〈PId 表达式〉::=
SELF
PARENT

| OFFSPRING
| SENDER

语义

PId（进程标识符）表达式访问在§2.4.4节中定义的一个隐含的进程变量。进程变量表达式被解释为最后关联于对应的隐含变量的值。

PId表达式具有一个PId类别。

PId表达式有一个值，此值是最后关联于对应变量的值，见§2.4.4节中的定义。

5.5.4.4 视见表达式

视见表达式允许一个进程获得在同一个功能块中的另一个进程的变量的值，就仿佛此变量是本地定义的。进行视见的进程不能修改关联于该变量的值。

抽象文法

视见表达式 :: 变量标识符
表达式

此表达式必须是一个PId表达式。

此变量标识符必须是由此表达式所标识的进程中的一个变量的标识符。

具体正文文法

〈视见表达式〉::=
VIEW（〈变量标识符〉,〈PId表达式〉）

在含有此〈视见表达式〉的进程中，必须在一个〈视见定义〉中规定此〈变量标识符〉是可视见的。〈变量标识符〉中的〈限定符〉仅在下述情况才可以省略掉，即在包围它的〈进程定义〉的〈视见定义〉中，没有其它变量具有同样的〈名字〉部分。

语义

解释视见表达式的方式与解释变量访问的方式（参见§5.5.2.2节）相同。此访问的变量是由该PId表达式标识的进程中的变量。（参见§5.5.4.3节）。

视见表达式具有一个值和一个类别，它们是所访问变量的值和类别。

视见表达式是在一个功能块中的一个进程中进行解释的，此视见表达式中的PId表达式所标识的必须是在同一功能块中的进程，否则，视见表达式出错，并且，系统未来的行为是不确定的。此PId表达式所标识的进程类型必须与对应视见定义中的进程标识符的进程类型相同。

5.5.4.5　定时器活跃表达式

抽象文法

定时器活跃表达式　::　定时器标识符
表达式 *

定时器活跃表达式中的表达式 * 的类别必须通过位置对应于类别引用标识符 *，后者直接跟在由该定时器标识符所标识的定时器名字（§2.8）的后面。

具体正文文法

〈定时器活跃表达式〉::=
ACTIVE（〈定时器标识符〉[（〈表达式表〉)])

〈表达式表〉的定义见§5.5.2.1节。

语义

〈定时器活跃表达式〉是一个具有布尔类别的表达式。如果所指明的定时器是活跃的话，它将具有值 True，这里所指明的定时器就是由此〈定时器标识符〉标识的，并且以〈表达式〉（若有的话）的值来置定时的定时器。否则，〈定时器活跃表达式〉具有值 False。表中的表达式按照给出的次序进行解释。

5.6 预定义数据

本节定义在系统层次隐含地定义的数据类别，和数据生成程序。

注意，§5.4.1.1节定义了特种运算符（中缀和一元）的语法和优先权，但这些运算符（除REM和MOD之外）的语义由本节的数据定义来确定。

5.6.1 布尔类别

5.6.1.1 定义

```
NEWTYPE      Boolean
     LITERALS True, False;
     OPERATORS

       " NOT"        : Boolean              -> Boolean;
                                                    /*
       " ="          :Boolean, Boolean      -> Boolean; " =" 和" /=" 运算符是
       " /="         :Boolean, Boolean      -> Boolean; 隐含的。参见§5.4.1.4节
                                                    */
       " AND"        :Boolean, Boolean      -> Boolean;
       " OR"         :Boolean, Boolean      -> Boolean;
       " XOR"        :Boolean, Boolean      -> Boolean;
       " =>"         :Boolean, Boolean      -> Boolean;
   AXIOMS
       " NOT" (True)                  ==False;
       " NOT" (False)                 ==True;

       " =" (True, True)              ==True;
       " =" (True, False)             ==False;
       " =" (False, True)             ==False;
       " =" (False, False)            ==True;

       " /=" (True, True)             ==False;
       " /=" (True, False)            ==True;
       " /=" (False, True)            ==True;
       " /=" (False, False)           ==False;

       " AND" (True, True)            ==True;
       " AND" (True, False)           ==False;
       " AND" (False, True)           ==False;
       " AND" (False, False)          ==False;

       " OR" (True, True)             ==True;
       " OR" (True, False)            ==True;
       " OR" (False, True)            ==True;
       " OR" (False, False)           ==False;

       " XOR" (True, True)            ==False;
       " XOR" (True, False)           ==True;
       " XOR" (False, True)           ==True;
       " XOR" (False, False)          ==False;

       " =>" (True, True)             ==True;
       " =>" (True, False)            ==False;
```

```
    " =>" (False, True)          ==True;
    " =>" (False, False)         ==True;
ENDNEWTYPE Boolean;
```

5.6.1.2 应用

布尔类别用来表示 True 和 False 值。它常常用来作为比较的结果。

在 SDL 中很多数据的简化形式要用到布尔类别，例如，不带有“==”符号的公理以及隐含的相等性运算符“=”和“/=”。

5.6.2 字符类别

5.6.2.1 定义

```
NEWTYPE Character
    LTTERALS
      NUL,   SOH  STX  ETX,   EOT,   ENQ,   ACK,   BEL,
      BS,    HT,   LF,  VT,    FF,    CR,    SO,    SI,
      DLE,   DC1,  DC2, DC3,   DC4,   NAK,   SYN,   ETB,
      CAN,   EM,   SUB, ESC,   IS4,   IS3,   IS2,   IS1,
        ' ',    '!',  '"',  '#',   '$',   '%',   '&',   ''',
        '(',    ')',  '*',  '+',   ',',   '-',   '.',   '/',
        '0',    '1',  '2',  '3',   '4',   '5',   '6',   '7',
        '8',    '9',  ':',  ';',   '<',   '=',   '>',   '?',
        '@',    'A',  'B',  'C',   'D',   'E',   'F',   'G',
        'H',    'I',  'J',  'K',   'L',   'M',   'N',   'O',
        'P',    'Q',  'R',  'S',   'T',   'U',   'V',   'W',
        'X',    'Y',  'Z',  '[',   '\',   ']',   '^',   '_',
        '`',    'a',  'b',  'c',   'd',   'e',   'f',   'g',
        'h',    'i',  'j',  'k',   'l',   'm',   'n',   'o',
        'p',    'q',  'r',  's',   't',   'u',   'v',   'w',
        'x',    'y',  'z',  '{',   '|',   '}',   '~',   DEL;
/*''' 是一个撇号，' '是一个空格，
     '_' 是一个上划线或代字号*/
OPERATORS
                                                    /*
    " ="     :Character, Character -> Boolean;      运算符标记“=”和“/=”是隐含的
    " /="    :Character, Character -> Boolean;      —参见§5.4.1.4
                                                    */
    " <"     :Character, Character -> Boolean;
    " <="    :Character, Character -> Boolean;
    " >"     :Character, Character -> Boolean;
    " >="    :Character, Character -> Boolean;
AXIOMS
/*下面规定了两相邻字符字面值的“小于”关系*/
    NUL  <SOH    ==True;  SOH    <STX    ==True;
    STX  <ETX    ==True;  ETX    <EOT    ==True;
    EOT  <ENQ    ==True;  ENQ    <ACK    ==True;
    ACK  <BEL    ==True;  BEL    <BS     ==True;
    BS   <HT     ==True;  HT     <LF     ==True;
    LF   <VT     ==True;  VT     <FF     ==True;
    FF   <CR     ==True;  CR     <SO     ==True;
    SO   <SI     ==True;  SI     <DLE    ==True;
    DLE  <DC1    ==True;  DC1    <DC2    ==True;
```

DC2 <DC3 ==True; DC3 <DC4 ==True;
DC4 <NAK ==True; NAK <SYN ==True;
SYN <ETB ==True; ETB <CAN ==True;
CAN <EM ==True; EM <SUB ==True;
SUB <ESC ==True; ESC <IS4 ==True;
IS4 <IS3 ==True; IS3 <IS2 ==True;
IS2 <IS1 ==True; IS1 <" ==True;
' ' < '!' ==True; '!' < '"' ==True;
'"' < '#' ==True; '#' < '$' ==True;
'$' < '%' ==True; '%' < '&' ==True;
'&' < ''' ==True; ''' < '(' ==True;
'(' < ')' ==True; ')' < '*' ==True;
'*' < '+' ==True; '+' < ',' ==True;
',' < '-' ==True; '-' < '.' ==True;
'.' < '/' ==True; '/' < '0' ==True;
'0' < '1' ==True; '1' < '2' ==True;
'2' < '3' ==True; '3' < '4' ==True;
'4' < '5' ==True; '5' < '6' ==True;
'6' < '7' ==True; '7' < '8' ==True;
'8' < '9' ==True; '9' < ':' ==True;
':' < ';' ==True; ';' < '<' ==True;
'<' < '=' ==True; '=' < '>' ==True;
'>' < '?' ==True; '?' < '@' ==True;
'@' < 'A' ==True; 'A' < 'B' ==True;
'B' < 'C' ==True; 'C' < 'D' ==True;
'D' < 'E' ==True; 'E' < 'F' ==True;
'F' < 'G' ==True; 'G' < 'H' ==True;
'H' < 'I' ==True; 'I' < 'J' ==True;
'J' < 'K' ==True; 'K' < 'L' ==True;
'L' < 'M' ==True; 'M' < 'N' ==True;
'N' < 'O' ==True; 'O' < 'P' ==True;
'P' < 'Q' ==True; 'Q' < 'R' ==True;
'R' < 'S' ==True; 'S' < 'T' ==True;
'T' < 'U' ==True; 'U' < 'V' ==True;
'V' < 'W' ==True; 'W' < 'X' ==True;
'X' < 'Y' ==True; 'Y' < 'Z' ==True;
'Z' < '[' ==True; '[' < '\' ==True;
'\' < ']' ==True; ']' < '^' ==True;
'^' < '_' ==True; '_' < '`' ==True;
'`' < 'a' ==True; 'a' < 'b' ==True;
'b' < 'c' ==True; 'c' < 'd' ==True;
'd' < 'e' ==True; 'e' < 'f' ==True;
'f' < 'g' ==True; 'g' < 'h' ==True;
'h' < 'i' ==True; 'i' < 'j' ==True;
'j' < 'k' ==True; 'k' < 'l' ==True;
'l' < 'm' ==True; 'm' < 'n' ==True;
'n' < 'o' ==True; 'o' < 'p' ==True;
'p' < 'q' ==True; 'q' < 'r' ==True;
'r' < 's' ==True; 's' < 't' ==True;
't' < 'u' ==True; 'u' < 'v' ==True;
'v' < 'w' ==True; 'w' < 'x' ==True;
'x' < 'y' ==True; 'y' < 'z' ==True;
'z' < '{' ==True; '{' < '|' ==True;
'|' < '}' ==True; '}' < '~' ==True;

```
        \_/      <DEL==True;
    FOR ALL a, b, c IN Character (
        a<a                 ==False;
        a<b AND b<c=> a<c       ==True;
        a<b                 ==b>a;
        a<b OR a>b          ==a/=b;
        a<b=> NOT (b<a);
        NOT (a/=b)          ==a=b;
        a<b OR a=b          ==a<=b;
        a>b OR a=b          ==a>=b;)
ENDNEWTYPE Character;
```

5.6.2.2 应用

字符类别定义了长度为1的字符串，这些字符就是国际5号字母表中的字符。根据字母表的国际参考版本，或者用串，或者用简写符号来定义这些字符。在不同的国家使用此字母表时，其印刷表示法可以不同。

对于字符，定义了128个不同的字面值和值。值的次序以及相等和不相等关系都已定义。

5.6.3 串生成程序

5.6.3.1 定义

```
GENERATOR String(TYPE Itemsort,LITERAL Emptystring)/*串是从1开始"标引"的*/
    LITERALS Emptystring;
    OPERATORS
        MkString : Itemsort -> String;                    /*从一个项形成一个串*/
        Length   : String   -> Integer;                   /*串的长度*/
        First    : String:  -> Itemsort;                  /*串的第一项*/
        Last     : String:  -> Itemsort;                  /*串的最后一项*/
        " //":   : String, String       -> String;        /*拼接 */
        Extract! : String, Integer      -> Itemsort;      /*从串中抽取一个项*/
        Modify!  : String, Integer, Itemsort  -> String;  /*修改串的值*/
        SubString:String, Integer, Integer    -> String;  /*从串中取一子串*/
                /*substring (s, i, j) 给出一个长度为j、从第i个元素开始的串*/
    AXIOMS
    FOR ALL item, itemi, itemj, iteml, item2 IN Itemsort (
    FOR ALL s, s1, s2, s3 IN String (
    FOR ALL i, j IN Integer (
        type String Length (Emptystring)              ==0;
        type String Length (MkString (item))          ==1;
        type String Extract! (MkString (item), 1)     ==item;
        First (s)                                     ==Extract! (s, 1);
        Last (s)                                      ==Extract! (s, Length (s));
        Length (s1//s2)                               ==Length (s1) +Length (s2);
        Length (Modify! (s, i, item))                 ==Length (s);
         (s1//s2) //s3                                ==s1// (s2//s3);
        Emptystring//s                                ==s;
        s//Emptystring                                ==s;
        Emptystring= (MkString (item) //s2) ==False;
        (MkString(item1)//s1)=(MkString(item2)//s2)==(item1=item2)AND(s1=s2);
```

```
        i>0 AND i<=Length(s)==True==>
                    Extract!(Modify!(s,i,item),i)==item;
        i/=j AND i>0 AND i<=Length(s) AND j>0 AND j<=Length(s)==True==>
                    Extract!(Modify!(s,i,item),j)==Extract!(s,j);
        i<=0 OR i>Length(s)==True==>Extract!(s,i)==ERROR!;

        i/=j==True==>
            Modify!(Modify!(s,i,itemi),j,itemj)==Modify!(Modify!(s,j,itemj),i,itemi);
        Modify!(Modify!(s,i,item1),i,item2)==Modify!(s,i,item2);
        i<=0 OR i>Length(s)==True==>Modify!(s,i,item)==ERROR!;
        i<=Length(s1)==Ture==>
                    Extract!(s1//s2,i)==Extract!(s1,i);
        i>Length(s1)==True==>
                    Extract!(s1//s2,i)==Extract!(s2,i-Length(s1));

        i>0 AND i<=Length(s)==True==>SubString(s,i,0)==Emptystring;
        i>0 AND i<=Length(s)==True==>SubString(s,i,1)==
                    MkString(Extract!(s,i));
        i>0 AND i<=Length(s)AND i-1+j<=Length(s)AND j>1==True==>
                    SubString(s,i,j)==SubString(s,i,1)//SubString(s,i+1,j-1);
        i<0 OR i>Length(s)OR j<=0 OR i+j>Length(s)==True==>
                    SubString(s,i,j)==ERROR!;

        i>0 AND i<=Length(s)==True==>
            Modify!(s,i,item) ==
                Substring(s,1,i-1)//MkString(item)//Substring(s,i+1,Length(s)-i);)));
ENDGENERATOR String;
```

5.6.3.2 应用

串生成程序可以用来定义一个类别，它使得我们可以构成具有任意项类别的串。最普通的用法就是生成下面所定义的字符串。

为了访问串的值和把值赋给串，典型地是采用抽取运算符 Extract!和修改运算符 Modify!，并采用在§5.4.2.4节和§5.5.3.1节中定义的简化符号。

5.6.4 字符串类别

5.6.4.1 定义

```
NEWTYPE Charstring String(Character,'')
    ADDING LITERALS NAMECLASS""((' ':'&')OR """ OR ('(':'~'))+"";
    /*用任意字符(从一个空格' '到一个上划线'—')组成的具有任意长度的字符串*/
        /* 'ABC' == 'AB' // 'C'
        形式的等式是隐含的——参见§5.4.1.2节*/
    MAP FOR ALL c IN Character LITERALS(
        FOR ALL charstr IN Charstring LITERALS(
            Spelling(charstr)==Spelling(c)=>charstr==Mkstring(c);
        ) );/*串'A'是从字符'A'构成等等。
ENDNEWTYPE Charstring;
```

5.6.4.2　应用

字符串类别定义了用字符组成的串。一个字符串字面值可以含有印刷字符和空格。采用Mkstring可以把非印刷字符作为一个串来使用，例如Mkstring(DEL)。

/*举例*/SYNONYM newline-prompt Charstring=Mkstring(CR)//Mkstring(LF)//'$>';

5.6.5　整数类别

5.6.5.1　定义

```
NEWTYPE Integer
        LITERALS NAMECLASS( '0' : '9' ) * ( '0' : '9' );
            /*在一个数字(0～9中之一)前面可有任意个数字*/
        OPERATORS
            "-"     :Integer                -)Integer;
            "+"     :Integer,Integer        -)Integer;
            "-"     :Integer,Integer        -)Integer;
            "*"     :Integer,Integer        -)Integer;
            "/"     :Integer,Integer        -)Integer;
                                                /*
            "="     :Integer,Integer        -)Boolean;      运算符标记"="和"/="是
            "/="    :Integer,Integer        -)Boolean;      隐含的——参见§5.4.1.4节
                                                  */
            "<"     :Integer,Integer        -)Boolean;
            ">"     :Integer,Integer        -)Boolean;
            "<="    :Integer,Integer        -)Boolean;
            ">="    :Integer,Integer        -)Boolean;
            Float   :Integer                -)Real; /*公理在NEWTYPE Real定义中*/
            Fix     :Real                   -)Integer;/*公理在NEWTYPE Real定义中*/
AXIOMS
FOR ALL a,b,c IN Integer(
            /*求反*/
            0-a                             ==-a;
            /*加法*/
            0+a                             ==a;
            a+b                             ==b+a;
            a+(b+c)                         ==(a+b)+c;
            /*减法*/
            a-a                             ==0;
            (a-b)-c                         ==a-(b+c);
            (a-b)+c                         ==(a+c)-b;
            a-(b-c)                         ==(a+c)-b;
            /*乘法*/
            a*0                             ==0;
            a*1                             ==a;
            a*b                             ==b*a;
            a*(b*c)                         ==(a*b)*c;
            a*(b+c)                         ==a*b+a*c;
            a*(b-c)                         ==a*b-a*c;
            /*排序*/
            a+1>a                           ==True;
            a-1<a                           ==True;
            /*相等性*/
```

```
        (a>b) OR (b>a)==NOT (a=b);
        /*正规的排序公理*/
        "<"(a,a)                    ==False;
        "<"(a,b)                    ==">"(b,a);
        "<="(a,b)                   =="OR"("<"(a,b),"="(a,b));
        ">="(a,b)                   =="OR"(">"(a,b),"="(a,b));
        "<"(a,b)==True              ==>NOT("<"(b,a))==True;
        "<"(a,b)AND"<"(b,c)==True==>"<"(a,c)==True;
        /*除法*/
        a/0                         ==ERROR!;
        a>=0 AND b>a==True                              ==>a/b==0;
        a>=0 AND b<=a AND b>0==True                     ==>a/b==1+(a-b)/b;
        a>=0 AND b<0==True                              ==>a/b==-(a/(-b));
        a<0 AND b<0==True                               ==>a/b==(-a)/(-b);
        a<0 AND b>0==True                               ==>a/b==-((-a)/b);
        /*字面值2到9*/
        TYPE Integer 2==1+1;TYPE Integer 3==2+1;
        TYPE Integer 4==3+1;TYPE Integer 5==4+1;
        TYPE Integer 6==5+1;TYPE Integer 7==6+1;
        TYPE Integer 8==7+1;TYPE Integer 9==8+1;
    MAP /*0至9以外的字面值*/
        FOR ALL a,b,c IN Integer LITERALS
        (Spelling(a)==Spelling(b)//Spelling(c),Length(Spelling(c))==1==>
                a==b*(9+1)+c;
        );
ENDNEWTYPE Integer;
```

5.6.5.2 应用

整数类别用于带有小数点符号的算术整数。

5.6.6 自然数同义类型

5.6.6.1 定义

```
SYNTYPE Natural=Integer CONSTANTS>=0 ENDSYNTYPE Natural;
```

5.6.6.2 应用

当仅需要正的整数时，就使用自然数同义类型。所有的运算符将是整数运算符，但是当一个值用作为一个参量或被赋值时，要对此值进行检验。一个负的值将是一个错误。

5.6.7 实数类别

5.6.7.1 定义

```
NEWTYPE Real
        LITERALS NAMECLASS (('0':'9')*('0':'9'))
        OR (('0':'9')*'.'('0':'9')+);
        OPERATORS
            "-"   :Real              ->Real;
            "+"   :Real,Real         ->Real;
            "-"   :Real,Real         ->Real;
            "*"   :Real,Real         ->Real;
            "/"   :Real,Real         ->Real;
```

```
                                    /*
    "="  :Real,Real      ->Boolean;   运算符标记"="和"/="是
    "/=" :Real,Real      ->Boolean;   隐含的——参见§5.4.1.4节
                                    /*
    "<"  :Real,Real      ->Boolean;
    ">"  :Real,Real      ->Boolean;
    "<=" :Real,Real      ->Boolean;
    ">=" :Real,Real      ->Boolean;
AXIOMS
FOR ALL a,b,c IN Real(
    /*求反*/
    0-a                  ==-a;
    /*加法*/
    0+a                  ==a;
    a+b                  ==b+a;
    a+(b+c)              ==(a+b)+c;
    /*减法*/
    a-a                  ==0;
    (a-b)-c              ==a-(b+c);
    (a-b)+c              ==(a+c)-b;
    a-(b-c)              ==(a+c)-b;
    /*乘法*/
    a*0                  ==0;
    a*1                  ==a;
    a*b                  ==b*a;
    a*(b*c)              ==(a*b)*c;
    a*(b+c)              ==a*b+a*c;
    a*(b-c)              ==a*b-a*c;);
    /*排序*/
    FOR ALL i,j IN Integer(
        Float(i)>Float(j)          ==TYPE Integer">"(i,j);
        Float(j)=0==False          ==>Float(i)/Float(j)>0==Float(i)>0
                          AND Float(j)>0 OR Float(i)<0 AND Float(j)<0;
        Float(i)>0 AND Float (j)>0 AND Float(i)>Float(j)
                                   ==>Float(i)/Float(j)>1==True;);
        FOR ALL a,r,b IN Real(a+r<b+r==a<b;
            r>0==>a*r<b*r==a<b;
            r<0==>a*r<b*r==b<a;);
        /*正规的排序公理*/
        FOR ALLa,b,c,dINReal(
            /*相等性和排序*/
            (a>b)OR(b>a)==NOT(a=b);
            "<"(a,a)               ==False;
            "<"(a,b)               ==">"(b,a);
            "<="(a,b)              =="OR"("<"(a,b),"="(a,b));
            ">="(a,b)              =="OR">"(a,b),"="(a,b));
            "<"(a,b)==True              ==>NOT("<"(b,a))==True;
            "<"(a,b)AND"<"(b,c)==True==>"<"(a,c) ==True;
        /*除法*/
            a/0                    ==ERROR!;
            a=0==False==>a/a ==    1;
            a=0==False==>0/a ==    0;
            b=0==False==>(a/b)*b==a;
```

```
        b=0 OR c=0==False==>(a*b)/(c*b)==a/c;
        b=0 OR d=0==False==>a/b+c/d==(a*d+b*c)/(b*d);
        b=0 OR d=0==False==>a/b-c/d==(a*d-b*c)/(b*d);
        b=0 OR d=0==False==>(a/b)*(c/d)==(a*c)/(b*d);
        b=0 OR d=0==False==>(a/b)/(c/d)==(a*d)/(b*c););
        /*整数与实数间的转换                    */
    FOR ALL a,i,j IN Integer(
    FOR ALL r IN Real(
      Fix(Float(a))                  ==a;
      r-1.0<Float(Fix(r))            ==True;/*Note Fix(1.5)==1,Fix(-0.5)==-1*/
      Float(Fix(r))<=r               ==True;
      Float(TYPE integer"+"(i,j))==      Float(i)+Float(j);));
  MAP
    FOR ALL r,s IN Real LITERALS(
        FOR ALL i,j IN Integer LITERALS(
            Spelling(r)==Spelling(i)==>r==Float(i);
            Spelling(r)==Spelling(i)==>i==Fix(r);
            Spelling(r)==Spelling(i)//Spelling(s),Spelling(s)== '.' //Spelling(j)
                                    ==>r==Float(i)+s;
            Spelling(r)== '.' //Spelling(i),Length(Spelling(i))==1
                                    ==>r==Float(i)/10;
            Spelling(r)== '.' //Spelling(i)//Spelling(j),Length(Spelling(i))==1,
                Spelling(s)== '.' //Spelling(j)
                                    ==>r==(Float(i)+s)/10;
        )    );
ENDNEWTYPE Real;
```

5.6.7.2　应用

实数类别用来表示实数。凡是能够用一个整数除以另一个整数来表示的数都可以用实数类别来表示。不能够以这种方式表示的数(无理数—例如$\sqrt{2}$)不是实数类别的一部分。然而，对于实际工程，通常可以采用足够准确的近似。不采用另外的技术手段，要定义一个数的集合包括所有的无理数是不可能的。

5.6.8　数组生成程序

5.6.8.1　定义

```
GENERATOR Array(TYPE Index,TYPE Itemsort)
        OPERATORS
          Make!     :Itemsort                          ->Array;
          Modify!   :Array,Index,Itemsort              ->Array;
          Extract!  :Array,Index                       ->Itemsort;
        AXIOMS
        FOR ALL item,item1,item2,itemi,itemj IN Itemsort(
        FOR ALL i,j,ipos IN Index(
        FOR ALL a,s IN Array(
            type Array Extract!(Make!(item,i))==item;
            Modify!(Modify!(s,i,item1),i,item2)==Modify!(s,i,item2);
            Extract!(Modify!(a,ipos,item),ipos)==item;
            i=j==False      ==>Extract!(Modify!(a,j,item),i)==Extract!(a,i);
            i=j==False      ==>
                Modify!(Modify!(s,i,itemi),j,itemj)==
                        Modify!(Modify!(s,j,itemj),i,itemi);)));
```

```
        /*相等性*/
        type Array Make!(item1)=Make!(item2)==item1=item2;
    Modify!(a,i,item)=s          ==(Extract!(s,i)=item)AND(a=s);
ENDGENERATOR Array;
```

5.6.8.2 应用

数组生成程序可以用来定义一个类别，它由另一个类别来标引。例如

```
NEWTYPE indexbychar Array(Character,Integer)
ENDNEWTYPE indexbychar;
```

定义了一个含有整数元素的数组，其元素用字符来标引。

数组元素的标引通常采用简化形式 Modify!和 Extract!。其定义在§5.5.3.1和§5.4.2.4节。例如

```
DCL charvalue indexbychar;
.....
TASK charvalue('A'):=charvalue('B')-1;
```

5.6.9 幂集生成程序

5.6.9.1 定义

```
GENERATOR Powerset(TYPE Itemsort)
    LITERALS Empty;
    OPERATORS
        "IN      :Itemsort,    Powerset  ->Boolean;/*是运算符成员*/
        Incl     :Itemsort,    Powerset  ->Powerset;/*把项包含到集合中*/
        Del      :Itemsort,    Powerset  ->Powerset;/*从集合中删去项*/
        "<"      :Powerset,    Powerset  ->Boolean;/*是运算符的正常子集*/
        ">"      :Powerset,    Powerset  ->Boolean;/*是运算符的正常超集*/
        "<="     :Powerset,    Powerset  ->Boolean;/*是运算符的子集*/
        ">="     :Powerset,    Powerset  ->Boolean;/*是运算符的超集*/
        "AND"    :Powerset,    Powerset  ->Powerset;/*两个集合的交集*/
        "OR"     :Powerset,    Powerset  ->Powerset;/*两个集合的并集*/
    AXIOMS
    FOR ALL i,j IN Itemsort(
    FOR ALL p,ps ,a,b,c IN Powerset(
        i IN type Powerset Empty              ==False;
        i IN Incl(i,ps)                       ==True;
        i IN ps                               ==i IN Incl(j,ps);
        type Powerset Del(i,Empty)            ==Empty;
        NOT(i IN ps)                          ==Del(i,ps)=ps;
        Del(i,Incl(i,ps))                     ==ps;
        i=j==False==>Del(i,Incl(j,ps))        ==Incl(j,Del(i,ps));
        Incl(i,Incl(j,p))                     ==Incl(j,Incl(i,p));
        Incl(i,Incl(i,p))                     ==Incl(i,p);
        a<b=>(i IN a=>i IN b)                 ==True;
        i IN(a AND b)                         ==TYPE Boolean"AND"(i IN a,i IN b);
        i IN (a OR b)                         ==TYPE Boolean"OR"(i IN a,i IN b);
        /*相等性*/
        Empty=Incl(i,ps)==False;
        Incl(i,a)=b ==(i IN b)AND(a=Del(i,b));
        /*正规的排序公理*/
        "<"(a,a) ==False;
```

```
        "<"(a,b) ==">"(b,a);
        "<="(a,b)=="OR"("<"(a,b),"="(a,b));
        ">="(a,b)=="OR"(">"(a,b),"="(a,b));
        "<"(a,b)==True ==>NOT("<"(b,a))==True;
        TYPE Boolean"AND"("<"(a,b),"<"(b,c))==True
        ==>"<"(a,c)==True;))
ENDGENERATOR Powerset;
```

5.6.9.2 应用

幂集用来表示数学集合。例如

```
NEWTYPE Boolset Powerset(Boolean)ENDNEWTYPE Boolset;
```

一个变量可以具有此类别 Boolset。此变量可以为空，或包含(True)、(False)或(True,False)。

5.6.10 PId 类别

5.6.10.1 定义

```
NEWTYPE PId
        LITERALS Null;
        OPERATORS       unique!:     PId->PId;
                        /*
                        "="   :Pid,Pid->Boolean;运算符标记"="和"/="
                        "/="  :Pid,Pid->Boolean;是隐含的。见§5.4.1.4
                         */
        AXIOMS
        FOR ALL p,p1,p2 IN PId(
            unique!(p) =Null              ==False;
            unique!(p1)=unique!(p2)        ==p1=p2);
DEFAULT Null;
ENDNEWTYPE PId;
```

5.6.10.2 应用

PId 类别用来对进程进行标识。注意，除值 Null 以外，PId 类别中没有别的字面值。当一个进程被创建时，基础系统使用 unique!运算符来生成一个新的、唯一的值。

5.6.11 持续时间类别

5.6.11.1 定义

```
NEWTYPE Duration INHERITS Real("+","-",">")
ADDING
        OPERATORS
            "*" :Duration,Real->Duration;
            "/" :Duration,Real->Duration;
        AXIOMS/*在下面，每一个 d 必须是上下文中的一个持续时间值*/
        FOR ALL d,z IN Duration(
        FOR ALL r IN Real(
            /*相等性*/
            (d>z)OR(z>d)==NOT(d=z);
            /*持续时间乘以实数*/
            d*0      ==0;
```

```
        0 * r      ==0;
        d * TYPE Real"+"(1,r) ==d+(d * r);
        d * TYPE Real"-"(1,r)  ==d-(d * r);
        d * TYPE Real"-"(r,1)  ==(d * r)-d;
        d * TYPE Real"-"(r)    ==0-(d * r);
        / * 持续时间被实数除 * /
        d/0==ERROR!;
        r=0==False==>d/r==d * TYPE Real"/"(1,r);
        / * 就是说,除以实数与乘以实数的倒数是一样的 * /
        r=0==False==>z * r=d ==(d/r)=z;));
    MAP
        FOR ALL d IN Duration LITERALS(
        FOR ALL r IN Real LITERALS(Spelling(d)==Spelling(r)
        ==>d==1 * r));
ENDNEWTYPE Duration;
```

5.6.11.2 应用

具有持续时间类别的值可用于加到当前时刻,来对定时器设置定时。持续时间类别的字面值是与实数类别的字面值一样的。持续时间的一个单位长度将决定于所设计的系统。

持续时间可以用实数来乘或除。

5.6.12 时间类别

5.6.12.1 定义

```
NEWTYPE Time INHERITS Real OPERATORS("<","<=",">",">=")ADDING
    OPERATORS
      "+" :Time,Duration       ->Time;
      "-" :Time,Duration       ->Time;
      "-" :Time,Time           ->Duration;
    AXIOMS
    FOR ALL t,t1,t2 IN Time(
    FOR ALL d,d1,d2 IN Duration(
      (t1>t2)OR(t2>t1)==NOT(t1=t2);
      t+0                      ==t;
      t-d                      ==t+TYPE Duration"-"(0,d);
      (t+d1)+d2                ==t+TYPE Duration"+"(d1,d2);
      (t+d1)-(t+d2)            ==TYPE Duration"-"(d1,d2);));
    MAP
      FOR ALL d IN Duration LITERALS(
      FOR ALL t IN Time LITERALS(Spelling(d)==Spelling(t)==>t==0+d));
ENDNEWTYPE Time;
```

5.6.12.2 应用

NOW 表达式返回一个具有时间类别的值。把一个持续时间值加到一个时间值上（或从一个时间值减去）可以得到另一个时间值。从一个时间值减去另一个时间值可以得到一个持续时间值。时间值可用于设置定时器的到时时间。

时间的起点决定于系统。一个时间单位就是一个持续时间单位。

附 件 I

（附于建议 Z.100）

非参量化数据类型的形式模型①

I.1 多分类代数

多分类代数 A 是一个2元组〈D，O〉，这里

a) D 是由一些集合组成的集合，并且，D 的元素称为（A 的）**数据载体**；数据载体 dc 的元素称为**数据值**；并且

b) O 是一组全函数，这里每一个函数的域是一个笛卡儿积，即 A 的数据载件和数据载体之一的范围的笛卡儿积。

I.2 数据类型定义的语义

I.2.1 一般概念

I.2.1.1 标记

标记 SIG 是一个元组〈S，OP〉，这里

a) S 是一组**类别标识符**（也称为类别）；并且

b) OP 是一组**运算符**。

运算符包括**运算标识符** OP，（变元）类别表 w，其元素在 S 中，以及（范围）类别 s∈S。这通常写为 op：w→s。如果 w 等于空表，则 op：w→s 称为类别 s 的一个**空数组**运算符或**常数符号**。

I.2.1.2 标记形态

设 SIG_1＝〈S_1，OP_1〉和 SIG_2＝〈S_2，OP_2〉是两个标记，**标记形态** g：$SIG_1 \rightarrow SIG_2$是一对映象

$$g = \langle gs : S_1 \rightarrow S_2,\ gop : OP_1 \rightarrow OP_2 \rangle$$

使得，对于所有的 $e\text{-}opid_1$＝〈$opidf_1$，〈gs（$e\text{-}sidf_1$），…，gs（$e\text{-}sidf_k$）〉，gs（e-res），pos〉∈op_1有

$$gop\ (e\text{-}opid_1) = \langle opidf_2, \langle (e\text{-}sidf_1), \cdots, (e\text{-}sidf_k) \rangle, (e\text{-}res), pos \rangle$$

用于某个运算标识符 $opidf_2$。

---

① CCITT 与 ISO 之间已达成了一致意见，把本附件的正文用作为关于抽象数据类型的基础代数模型的公共形式描述。这一内容除了出现在本建议中（在术语、印刷和编号方面适当地有些改动），也出现在 ISO IS8807之中。本附件的§§I.1、I.2.1.1、I.2.1.2、I.2.1.3、I.2.1.4、I.2.1.5、I.2.1.6、I.3、I.4.1、I.4.2、I.4.3、I.4.4、I.4.5以及 I.4.6分别出现在 IS8807的§§5.2、7.2.2.1、7.3.2.8、7.2.2.2、7.2.2.3、7.2.2.4、7.2.2.5、4.7、7.4.2.1、7.4.2.2、7.4.3、7.4.3和7.4.4中。本附件中的术语类别标识符、运算符、变量标识符、变量、代数规格以及运算，在 IS8807中分别被类别变量、运算变量、值变量、值变量、数据表现以及函数所取代。

I.2.1.3　项

设V为变量的任意集合，设〈S，OP〉为一个标记。类别s∈S的那些**项**是集合TERM（OP，V，S）中的项，此集合带有OP中的运算符和V中的变量。可以通过下面步骤归纳地对此集合作出定义：

a)　每一个变量x:s∈V都在TERM（OP，V，s）中；

b)　对每一个空数组运算符op，如果op∈OP，同时res（op）=s，则op在TERM（OP，V，s）之中；

c)　对于i=1、……、n，如果类别s_i的项t_i在TERM（OP，V，s_i）之中，则对于每一个op∈OP，如果它具有arg（op）=〈s_1，…，s_n〉和res（op）=s，则op（t_1，…，t_n）就在TERM（OP，V，s）之中。

如果项t是TERM（OP，V，s）的一个元素，则s称为t的类别，用sort（t）来表示。

由类别为s∈S的**基本项**组成的集合TERM（OP，s）被定义为集合TERM（OP，{}，S）。

I.2.1.4　等式

对于标记<S，OP>来说类别为s的**等式**是一个三元组〈V，L，R〉，这里

a)　V是一组变量标识符；

b)　L，R∈T（OP，V，s）；且

c)　s∈S

等式e′=〈{}，L′，R′〉是等式e=〈V，L，R〉的一个**基本实例**，对每一个V中的变量v：s，如果可以从L，R中获得L′，R′，则用具有类别s的相同的基本项来取代L，R中那个变量的每一次出现。

对于等式的基本实例〈{}，L，R〉，采用了L=R的写法。

注 — 如果没有引入语义上的复杂性的话，等式〈V，L，R〉也可以写为L=R。

I.2.1.5　条件等式

对于标记〈S，OP〉来说，类别为s的条件等式是一个三元组〈V，Eq，e〉，这里

a)　V是一组变量标识符；

b)　根据〈S，OP〉，Eq是一组等式，其变量是V中的变量；

c)　根据〈S，OP〉，e是一个类别为s的等式，其变量是V中的变量。

I.2.1.6　代数规格

代数规格 SPEC是一个三元组〈S，OP，E〉，这里

a)　〈S，OP〉是一个标记；

b)　根据〈S，OP〉，E是一组条件等式。

I.3 推理系统

推理系统是一个三元组 D=〈A，Ax，I〉，其中：

a) A 是一个集合，其元素称为**断言**；

b) A⊇Ax，Ax 是**公理**的集合，

c) I 是一组**推理规则**。

每一个推理规则 R∈I 具有下面的格式

$$R: \frac{P_1, \cdots, P_n}{Q}$$

这里，P_1，…，P_n，Q∈A。

推理系统 D 中对断言 P 的一个**推导**是由一些断言组成的一个有限序列 s，它满足下面的条件：

a) s 的最后元素是 P，

b) 如果 Q 是 s 的一个元素，则或是 Q∈Ax，或是存在一个规则 R∈I

$$R: \frac{P_1, \cdots, P_n}{Q}$$

其中 s 的元素 P_1，…，P_n 先于 Q。

如果推理系统 D 中存在 P 的一个推导，那么就把这点写成 D|-P。如果 D 是由上下文唯一确定的，则写法可简化为|-P

I.4 代数规格的语义

在§I. 4节中，一组类别 s、一组运算符 OP 以及一组等式 E 的所有事件都涉及一给定的代数规格 SPEC=〈S，OP，E〉，如在§I. 2. 1. 6中所定义的。

为了定义代数规格 SPEC 的语义，使用了与 SPEC 有关联的推理系统。此推理系统的定义在§I. 4. 1—I. 4. 3节中。应用这个推理系统，在§I. 4. 4和§I. 4. 5节中定义了与〈S，OP，E〉有关的基本项的集合和相同类的一种关系。这种关系在§I. 4. 6节中用来定义一种代数，用来表示由〈S，OP，E〉规定的数据类型（参见§I. 1节）。

I.4.1 由等式生成的公理

设 ceq 为一条件等式。由 ceq 生成的公理的集合，表示为 Ax（ceq），其定义如下：

a) 如果 ceq=〈V，Eq，e〉，其中 Eq≠{}，则 Ax（ceq）=｛｝；且

b) 如果 ceq=〈V，{}，e〉，则 Ax（ceq）是类别为 e 的所有基本实例的集合（参见§I. 2. 1. 3节）。

I.4.2 由等式生成的推理规则

设 ceq 为一条件等式。由 ceq 生成的推理规则的集合用 Inf（ceq）来表示，其定义如下：

a) 如果 ceq＝〈V，{}，e〉，则 Inf（ceq）＝{}，并且

b) 如果 ceq＝〈V，{e_1，…，e_n}，e〉，其中 n＞0，则 Inf（ceq）含有

$$\frac{e'_1，\cdots，e'_n}{e'}$$

的所有规则。

这里 e'_1，…，e'_n，e' 分别是 e_1，…，e_n，e 的基本实例，它们是用下述方法得到的：对于 V 中的每一个变量 x，对于那个变量在 e_1，…，e_n，e 中的每一次出现，用具有类别 sort（x）的相同的基本项来取代。

I.4.3 生成的推理系统

由一个代数规格 SPEC＝〈S，OP，E〉生成的**推理系统** D＝〈A，Ax，I〉（参见§I.3节），定义如下：

a) A 是等式 w. r. t. 〈S，OP〉的所有基本实例的集合；并且

b) Ax＝$\bigcup$ {Ax（ceq）|ceq∈E} $\bigcup$ID，其中 ID＝ {t＝t|t 是一个基本项}；且

c) I＝$\bigcup$ {Inf（ceq）|ceq∈E} $\bigcup$SI，

这里 SI 由下面的纲要给出

i) $\dfrac{t_1=t_2}{t_2=t_1}$对于所有的基本项 t_1，t_2；且

ii) $\dfrac{t_1=t_2，\ t_2=t_3}{t_1=t_3}$对于所有的基本项 t_1，t_2，t_3；且

iii) $\dfrac{t_1=t'_1，\ \cdots，\ t_n=t'_n}{op\ (t_1，\ \cdots，\ t_n)\ =op\ (t'_1，\ \cdots，\ t'_n)}$

对于所有的运算符 op：s_1，…，s_n→s∈OP，其中 n＞0，而且类别为 s_i 的 t_i，t'_i 之所有基本项，i＝1，…，n。

I.4.4 由一代数规格生成的相同关系

设 D 为由一代数规格 SPEC＝〈S，OP，E〉生成的推理系统。我们说对于 SPEC，两个基本项 t_1和 t_2是相同的，并用式子 $t_1 \equiv_{SPEC} t_2$来表示，当且仅当 D|-t_1＝t_2。

I.4.5 相同类

一个基本项 t 的 **SPEC-相同类** [t]，是根据代数规格 SPEC，与 t 相同的所有项的集合，即

$$[t] = \{t' \mid t \equiv_{SPEC} t'\}$$

I.4.6 商项代数

一个代数规格 SPEC=〈S，OP，E〉的语义解释是下面的叫做商项代数的多分类代数 Q=〈D_q，O_q〉，这里

a) D_q 是集合 {Q (s) |s∈S}，此处，对每一个 s=S，
Q (s) = { [t] |t 是类别 s 的基本项}；且

b) O_q 是运算的集合 {op′ |op∈OP}，这里 op′ 用下式定义
op′ ([t_1]，…，[t_n]) = [op (t_1，…，t_n)].

附件 A、B、C 和 E
附于建议 Z.100

功能规格和描述语言(SDL)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

附 件 A

(附于建议 Z.100)

SDL 词汇表

建议 Z.100 包含了 SDL 术语的正式定义。编辑 SDL 词汇表的目的是帮助新的 SDL 用户阅读建议 Z.100 和该建议的附件。词汇表给出了词汇的简洁定义和本建议定义该词汇的节号。本词汇表中的定义可以是建议 Z.100 中的正式定义的概要或意译，因此，也可能是不完全的。

一个定义中的术语也可以在本词汇表中找到。如果某一用楷体（英文用斜体）印刷的词组、例如过程标识符 (procedure identifier)、未出现在本词汇表中，则它可能是两个术语的链接，在此例中就是术语过程 (procedure) 后面跟术语标识符 (identifier)。当一个斜体词在本词汇表中找不到时，则该词可能是词汇表中某术语的导出词。例如，exported 是 export 的过去式。

除一个术语是另一个术语的同义词外，在术语的定义后面给出了在建议 Z.100 中使用该术语的主要参考处，这些参考处放在定义之后的方括号 [] 中。例如，〔3、2〕表示主要参考处是 § 3.2 节。

abstract data type　　抽象数据类型

F : type abstrait de données

S : tipo abstracto de datos

抽象数据类型是数据类型 (data type) 的同义词。所有 SDL 数据类型都是抽象数据类型。

abstract grammar　　抽象文法

F : grammaire abstraite

S : gramática abstracta

抽象文法定义 SDL 的语义。抽象文法通过抽象语法和良形式的规则来描述。〔1.2，1.4.1〕

abstract syntax　　抽象语法

F : syntaxe abstraite

S : sintaxis abstracta

与具体语法比较起来，抽象语法是描述一个 SDL 规格的概念性结构的手段。这里的具体语法代表 SDL 的每一种具体语法，即 SDL/GR 和 SDL/PR。〔1.2〕

access　　访问

F : accès

S : acceder

访问是对于变量的一种操作，它给出最后赋予变量的值。如果被访问的变量值具有不确定性则产生一个错误。

action **动作**

F : action

S : acción

一个动作是在一个跃迁串中执行的一个操作，例如，一个任务、输出、判定、创建请求，或一个过程调用。〔2.7〕

active timer **活跃定时器**

F : temporisateur actif

S : temporizador activo

一活跃定时器是在拥有它的过程的输入端口中有一定时器信号的定时器或是按计划在某个未来的时刻产生一定时器信号的定时器。〔2.8.2，5.5.4.5〕

actual parameter **实在参数**

F : paramètre réel

S : paràmetro efectivo

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表达式。要注意的是，在一过程调用的某些情况中，实在参数必须是一个变量（即一种特殊类型的表达式；参见 IN/OUT）。〔2.7.2，2.7.3，4.2.2〕

actual parameter list **实在参数表**

F : liste de paramètres réels

S : lista de paràmetros efectivos

实在参数表是实在参数组成的列表。实在参数通过位置与对应的形式参数表中的相应元素相匹配。

area **区**

F : zone

S : área; zona

一个区是具体图形语法中的一个二维空间。区常常对应于抽象语法中的节点，并且，它通常包含公共正文语法。在互作用图中，各个区可以用信道或信号路由连接起来。在控制流图中，可以用流线把区连接起来。

array **数组**

F : tableau (array)

S : matriz

数组是用来引进数组概念、简化数组定义的预定义生成程序。

assign **赋值**

F : affectation

S : asignar

赋值是作用了一个变量的操作，它将一个值联结到该变量，以取代以前与该变量相连结的值。〔5.5.3〕

assignment statement　　赋值语句

F : instruction d'affectation

S : sentencia de asignación

赋值语句是把一个值赋给一个变量的语句。〔5.5.3〕

association area　　关联区

F : zone d'association

S : área de asociación

一个关联区是在一互作用图中，区之间通过一联结符号形成的一个连接。有五种关联区：信道子结构关联区，输入关联区，优先输入关联区，连续信号结关联和保存关联区。〔2.6.3，3.2.3，4.10.2，4.11〕

axiom　　公理

F : axiome

S : axioma

公理是一种特殊的等式，隐含了它等价于布尔字面值 True。“公理”被用作为“公理和等式”的同义词。〔5.1.3〕

basic SDL　　基本 SDL

F : LDS de base

S : LED básico

基本 SDL 是在建议 Z.100 § 2 中定义的 SDL 的子集。

behaviour　　行为

F : comportement

S : comportamiento

一个系统的行为或功能行为是对多个激励序列的响应序列的集合。〔1.1.3〕

block　　功能块

F : bloc

S : bloque

一个功能块是系统的一部分或父辈功能块的一部分。当自身使用时，功能块是功能块实例的同义词。一个功能块是一作用域单位，并且，它提供了一静态接口。〔2.4.3〕

block area　　功能块区

F : zone de bloc

S : área de bloque

在一个交互作用图中功能块区是一个功能块的定义，或者是对一个功能块的引用。

block definition　　功能块定义

F : définition de bloc

S : definición de bloque

功能块定义是用 SDL/PR 对一个功能块作出的定义。〔2.4.2〕

block diagram　　功能块图

F : diagramme de bloc

S : diagrama de bloque

功能块图是以 SDL/GR 对一个功能块作出的定义。〔2.4.3〕

block substructure　　功能块子结构

F : sous-structure de bloc

S : subestructura de bloque

功能块子结构就是将功能块划分成在较低抽象层次上的子功能块和新的信道。

block substructure definition　　功能块子结构定义

F : définition de sous-structure de bloc

S : definición de subestructura de bloque

功能块子结构定义是用 SDL/PR 来表示一个被划分的功能块的功能块子结构。〔3.2.2〕

block substructure diagram　　功能块子结构图

F : diagramme de sous-structure de bloc

S : diagrama de subestructura de bloque

功能块子结构图是用 SDL/GR 来表示一个被划分功能块的功能块子结构。〔3.2.2〕

block tree diagram　　功能块树图

F : diagramme d'arborescence de bloc

S : diagrama de árbol de bloques

一个功能块树图是一个以 SDL/GR 写出的辅助文件，它通过一个倒立的树图（即父辈功能块在上）表示一个系统被划分成在较低的抽象层次上的多个功能块。〔3.22〕

BNF（Backus-Naur Form）　　巴科斯—诺尔范式

F : forme BNF (Backus-Naur Form)

S : FBN (forma Backus-Naur)

BNF（Backus-Naur Form）是一种形式符号表示法用来表示一种语言的具体正文语法。为了表示具体图形文法，采用了一种 BNF 的扩展形式。〔1.5.2，1.5.3〕

Boolean 布尔型

F : booléen

S : booleano

布尔型是在预定义的部分类型定义中给出定义的一个类别，它有真 True 和假 False 两种值。对于布尔类别来说，预定义运算符有 NOT、AND、OR、XOR 以及蕴含。〔5.6.1〕

channel 信道

F : canal

S : canal

信道是在两功能块之间传递信号的连接物。信道也在功能块与其环境之间传送信号。信道可以是单向的、也可以是双向的。〔2.5.1〕

channel definition 信道定义

F : définition

S : definición de canal

信道定义是用 SDL/PR 写出的一个信道的定义。〔2.5.1〕

channel definition area 信道定义区

F : zone de définition de canal

S : área de definición de canal

信道定义区是用 SDL/GR 给出的一个信道的定义。〔2.5.1〕

channel substructure 信道子结构

F : sous-structure de canal

S : subestructura de canal

一个信道子结构是将一个信道划分成在较低抽象层次上的一组信道和多个功能块。〔3.2.3〕

channel substructure definition 信道子结构定义

F : définition de sous-structure de canal

S : definición de subestructura de canal

信道子结构定义是用 SDL/PR 描述的信道子结构的定义。〔3.2.3〕

channel substructure diagram 信道子结构图

F : diagramme de sous-structure de canal

S : diagrama de subestructura de canal

信道子结构图是以 SDL/GR 描述的信道子结构的定义。〔3.2.3〕

character　字符

F : caractère (character)

S : carácter; character

字符是在预定义的部分类型定义中给出定义的一个类别，它的值是CCITTNo. 5字母表中的元素（例如，1、A、B、C、等等）。对于字符类别，次序运算符是预先定义的。[5.6.2]

chartstring　字符串

F : chaîne de caractères (character string)

S : cadena-de-caracteres; chartstring

字符串是在预定义的部分类型定义中给出定义的一个类别，它的值是用字符组成的串，它的运算符是串预定义生成程序（用字符来例示）的那些运算符。[5.6.4]

comment　注释

F : commentaire

S : comentario

注释是附加到SDL规格、或是阐明SDL规格的信息。在SDL/GR中，注释可由一条虚线连接到任一符号上。在SDL/PR中，注释由关键字COMMENT引入。注释不具有SDL定义的意义。也可参见注。[2.2.6]

common textual grammar　公共正文文法

F : grammaire textuelle commune

S : gramática textual común

公共正文文法是具体正文文法的子集。它既可用于SDL/PR，也可用于SDL/GR。[1.2]

communication path　通信路径

F : trajet de communication

S : trayecto de comunicación

通信路径是一种传输手段，它将信号实例从一个进程实例（或从环境）传送到另一个进程实例（或环境）。通信路径或是由信道路径构成、或是由信号路由路径构成、或是由此二者组成。[2.7.4]

complete valid input signal set　完整有效输入信号集

F : ensemble complet de signaux d'entrée valides

S : conjunto completo de señales de entrada válidas

一个进程的完整有效的输入信号集是该进程有效输入信号集、局部信号、定时器信号和隐含信号的并集。[2.4.4]

concrete grammar　具体文法

F : grammaire concrète

S : gramática concreta

具体文法是连同具体语法的良形式规则一起的具体语法。SDL/GR 和 SDL/PR 都是 SDL 的具体文法。具体文法在决定其语义时被映射到抽象文法。[1.2]

concrete graphical grammar　具体图形文法

F : grammaire graphic concréte

S : gramática gráfica concreta

具体图形文法是 SDL/GR 图形部分的具体文法。

concrete graphical syntax　具体图形语法

F : syntaxe graphique concrète

S : sintaxis gráfica concreta

具体图形语法是 SDL/GR 图形部分的具体语法。建议 Z.100 中的具体图形语法是用扩展形式的 BNF 表达的。

concrete syntax　具体语法

F : syntaxe concrète

S : sintaxis concreta

用于各种 SDL 表示法的具体语法包括用来表示 SDL 的实际符号以及 SDL 语法规则所要求的符号之间的关系。建议 Z.100 中所用的两种具体语法是具体图形语法和具体正文语法。[1.2]

concrete textual syntax　具体正文语法

F : syntaxe textuelle concrète

S : sintaxis textual concreta

具体正文语法是用于 SDL/PR 和 SDL/GR 正文部分的具体语法。建议 Z.100 中的具体正文语法是用 BNF 表达的。[1.2，1.5.2]

conditional expression　条件表达式

F : expression conditionnelle

S : expresión condicional

条件表达式是含有布尔表达式的一个表达式，其中的布尔表达式控制究竟是对结果表达式进行解释、还是对替换表达式进行解释。[5.5.2.3]

connect　连接

F : connect

S : conectar

连接是指一个信道对于一个或多个信号路由的连接。[2.5.3]

connector　连接符

F : connecteur

S : conector

连接符是一个SDL/GR符号，它或者是一个入连接符，或者是一个出连接符。在同一个进程或过程中，隐含有一条流线从出连接符到它所关联的具有相同名字的入连接符。

consistent partitioning subset　一致性划分子集

F : sous-ensemble de subdivision cohérent

S : subconjunto de partición consistente

一致性划分子集是一个系统规格中的一组功能块和子功能块，它以相互连系的部分在相应的抽象层次上提供了系统的全貌。因此，当一个功能块或子功能块包含在一个一致性划分子集中时，其父辈和同辈也包含在该一致性划分子集中。[3.2.1]

consistent refinement subset　一致性具体化子集

F : sous-ensemble de raffinement cohérent

S : subconjunto de refinamiento consistente

一致性具体化子集是一个一致性划分子集，它包含使用信号的所有功能块和子功能块，这里信号指的是任一功能块或子功能块所使用的信号。[3.3]

continuous signal　连续信号

F : signal continu

S : señal continua

一个连续信号是这样一种手段，它规定：当一个状态中的与信号有关的布尔条件变成真时，跟在连续信号之后的跃迁将被解释。[4.11]

control flow diagram　控制流图

F : diagramme de liaison de contrôle

S : diagrama de flujo de control

一个控制流图或是一个进程图，或是一个过程图，或是一个服务图。

create　创建

F : créer

S : crear

创建是创建请求的同义词。

create request　创建请求

F : demande de création

S : petición de crear

创建请求是这样一种动作，它用一个特定的进程类型作为样板而创建和启动一个新的进程实例。创建

请求中的实在参量将取代该进程中的形式参量。[2.7.2]

create line area　创建线区

F : zone de ligne de création

S : área de línea de crear

一个功能块图中的创建线区把创建进程（PARENT）的进程区与被创建进程（OFFSPRING）的进程区连接起来。[2.4.3]

data type　数据类型

F : type de données

S : tipo de datos

数据类型包括下列集合的定义：值（类别）的集合、适用于这些值的一些运算符的集合和当运算符应用于这些值时规定其行为的代数规则的集合。[2.3.1]

data type definition　数据类型定义

F : définition de type de données

S : definición de tipo de datos

数据类型定义决定了在一个 SDL 规格的任意给定点上的表达式和表达式之间关系的合法性。[5.2.1]

decision　判定

F : décicion

S : decisión

一个判定是在跃迁中的一个动作，它提出一个问题，对此问题可立即得到回答，并相应地从判定引出的几个跃迁中选出一个来以继续解释。[2.7.5]

decision area　判定区

F : zone de décision

S : área de decisión

判定区是一个判定的 SDL/GR 表示。[2.7.5]

default　缺省

F : défault

S : por defecto

缺省赋值是指一个值从一开始就是与各自缺省子句类别的变量相关联的。缺省子句可以出现在数据类型定义中。[5.5.3.3]

description　描述

F : description

S : descripción

系统的描述是该系统实际行为的描述。[1.1]

diagram 图

F : diagramme

S : diagrama

一个图是对一个规格之一部分的 SDL/GR 表示。[2.4.2]

duration 持续时间

F : durée (duration)

S : duración; duration

持续时间是在一个预定义的部分类型定义中所定义的一个类别，它的值是实数，它表示两个时刻之间的时间间隔。[5.6.11]

enabling condition 允许条件

F : condition de validation

S : condición habilitante (o habilitadora)

允许条件是用于有条件地接受一个信号作为输入的手段。[4.12]

enabling condition area 允许条件区

F : zone de condition de validation

S : área de condición habilitante (o habilitadora)

允许条件区是一个允许条件的 SDL/GR 表示。[4.12]

entity class 实体类

F : classe d'entité

S : clase de entidad

实体类是一类以用法的相似性为基础的 SDL 类型。[2.2.2]

environment 环境

F : environnement

S : entorno

术语环境是一个系统的环境的同义词。另外，根据上下文的意思，它也可以是一个功能块、进程、过程或一个服务的环境的同义词。[1.3.2]

environment of a system 某系统的环境

F : environnement d'un système

S : entorno de un sistema

某系统的环境是被确定系统的外部世界。环境通过发送（接收）信号实例到（从）该系统而与系统相互作用。[1.3.2]

equation　等式

F : équation

S : ecuación

等式是相同类别的项之间的一个关系式，对于在等式中的每一个值标识符代之以所有可能的值时，这个关系式总是成立的。等式也可以是一个公理。[5.1.3，5.2.3]

error　错误

F : erreur

S : error

在对一个系统的合法规格进行解释期间，当违反了 SDL 的动态条件之一时，便出现一个错误。一旦有一个错误出现，随后的系统行为不能由 SDL 确定。[1.3.3]

export　出口

F : export

S : exportación

术语出口是出口操作的同义词。

exported variable　出口变量

F : variable exportée

S : variable exportada

出口变量是可以用在出口操作中的变量。[4.13]

exporter　出口者

F : exportateur

S : exportador

一个变量的出口者是一个进程实例，它拥有该变量并出口该变量的值。[4.13]

export operation　出口操作

F : opération d'exportation

S : operación de exportación

出口操作是出口者公开一个变量的值的操作。参见进口操作。[4.13]

expression　表达式

F : expression

S : expresión

一个表达式是一个字面值、一次运算符的应用、一个同义词、一次变量访问、一个条件表达式，或是作用于一个或多个表达式的一个命令运算符。当对一个表达式进行解释时，就得到一个值，（否则系统出错）。[2.3.4，5.4.2.1]

external synonym　外部同义词

F : synonyme externe

S : sinónimo externo

外部同义词是一个预定义类别，其值在系统规格中没有确定。[4.3.1]

extract!　抽取

F : extract!

S : extraer!; extract!

抽取是当一个变量之后紧跟着括号括起来的表达式时隐含在一个表达式中的一个运算符。[5.4.2.4，5.6.8]

flow line　流线

F : ligne de liaison

S : línea de flujo

流线是控制流图中用来把区连接起来的符号。[2.2.4，2.6.7.2.2]

formal parameter　形式参数

F : paramètre formel

S : parámetro formal

形式参数是一个变量名，它将被赋予实际的值或是由实际变量取代。[2.4.4，2.4.5，4.2，4.10]

formal parameter list　形式参数表

F : liste de paramètres formels

S : lista de parámetros formales

形式参数表是形式参数的列表。

functional behaviour　功能行为

F : comportement fonctionnel

S : comportamiento funcional

功能行为是行为的同义词。

general option area　一般任选区

F: zone d'option générale

S: área de opción general

一般任选区是一个任选项的 SDL/GR 表示。[4.3.3]

general parameters　一般参数

F : paramètres généraux

S : parámetros generales

在一个系统的规格和描述中，一般参数是关于温度界限、结构、交换机容量、服务等级等一类的事情，它们不在 SDL 中规定。[1.1]

generator　生成程序

F : générateur

S : generador

生成程序是一个不完全的新类型的描述。在生成程序表现为一个新类型的状态以前，必须向它提供所缺少了的信息而使之实例化。[5.4.1.12]

graph　图形

F : graphe

S : gráfico

抽象语法中的图形是 SDL 规格的一部分，例如，过程图形或进程图形。

ground expression　基本表达式

F : expression close

S : expresión fundamental

基本表达式是仅包含运算符、同义词和字面值的一种表达式。[5.4.2.2]

hierarchical structure　层次结构

F : structure hiérarchique

S : estructure jeràrquica

层次结构是系统规格的一种结构，在这种结构中，采用划分和具体化，允许在不同的抽象层次上对系统进行观察。层次结构使我们能处理复杂的系统规格。也可参见功能块树图。[3.1]

identifier　标识符

F : identificateur

S : identificador

标识符是一个物体的唯一标识，它由一个限定符部分和一个名字构成。[2.2.2]

imperative operator　命令运算符

F : opérateur impératif

S : operador imperativo

命令运算符是 now（现时）表达式、视见表达式、定时器活跃表达式、进口表达式或是 PId 表达式、SELF、PARENT、OFFSPRING、SENDER 中之一。[5.5.4]

implicit transition　隐式跃迁

F : transition implicite

S : transición implícita

具体语法中的隐式跃迁由完整有效输入信号集中的一个信号所启动，并且，该状态没有在输入或保存中指定。一个隐式跃迁不含有动作并直接回到原先的状态。[4.8]

import　进口

F : import

S : importación

术语进口是进口操作的同义词。[4.13]

imported variable　进口变量

F : variable importée

S : variable importada

进口变量是用于进口操作中的变量。[4.13]

importer　进口者

F : importeur

S : importador

进口变量的进口者是进口该变量的进程实例。[4.13]

import operation　进口操作

F : opération d'importation

S : operación de importación

进口操作是获得一个出口变量的值的操作。[4.13]

IN variable　IN 变量

F : variable "IN"

S : variable IN

IN 变量是形式参数的一个属性，指的是一个值通过一个实在参数传递给一个过程的情况。[2.4.5]

IN/OUT variable　IN/OUT 变量

F : variable "IN/OUT"

S : variable IN/OUT

IN/OUT 变量是形式参数的一个属性，指的是一个形式参数名用作该变量的一个同义词的情况（就是说，实在参数必须是一个变量）。[2.4.5]

in-connector　入连接符

F : connecteur d'entrée

S : conector de entrada

入连接符是一个连接符。

infix operator　中缀运算符

F : opérateur infixe

S : operador infijo

中缀运算符是SDL的预定义二元运算符（=〉，OR，XOR，AND，IN，/=，=，>，<，<=，>=，+，－，||，*，/，MOD，REM）之一。中缀运算符放在它的两个变元之间。[5.4.1.1]

informal text　非形式正文

F : texte informel

S : texto informal

非形式正文是包含在一个SDL规格中的正文，其语义没有用SDL定义，而是通过某种别的方式来定义。非形式正文由撇号“'”括起来。[2.2.3]

initial algebra　基础代数

F : algèbre initiale

S : álgebra inicial

基础代数是定义抽象数据类型的形式方法。[5.3]

inlet　引入线

F : accès entrant

S : acceso de entrada

引入线指的是一条线，例如，信道或流线，它进入一个SDL/GR宏调用。[4.2.3]

input　输入

F : entrée

S : entrada

一个输入是对来自输入端口的一个信号的消耗，它启动一次跃迁。在一个信号的消耗过程中，与该信号相关联的值变成进程实例可用的值。[2.6.4，4.10.2]

input area　输入区

F : zone d'entrée

S : área de entrada

输入区是一个输入的SDL/GR表示。[2.6.4]

input port　输入端口

F : port d'entrée

S : puerto de entrada

进程的输入端口是一个队列，它按到达的顺序接收和保留信号，直到信号由输入消耗掉。输入端口可以含有任意个数的保留信号。[2.4.4]

instance　实例

F : instance

S : instancia

某类型的一个实例是具有该类型（在定义中给出）的一个实物。[1.3.1]

instantiation　实例产生

F : instantiation

S : instanciación

实例产生是指某类型一个实例的创建。[1.3.1]

integer　整数

F : entier (integer)

S : entero; integer

整数是在一个预定义的部分类型定义中所定义的一个类别，其值为数学上的整数（……，－2，－1，0，＋1，＋2，……）。对于整数类别，其预定义运算符为＋，－，＊，/ 以及排序运算符。[5.6.5]

interaction diagram　相互作用图

F : diagramme d'interaction

S : diagrama de interacción

相互作用图是一个功能块图，系统图，信道子结构图，或功能块子结构图。

keyword　关键字

F : mot clé

S : palabra clave

关键字是具体正文语法中保留的词法单元。[2.2.1]

label　标号

F : étiquette

S : etiqueta

标号是跟有冒号“:”的名字，在具体正文语法中用它来进行连接。[2.6.6]

level 层

F : niveau

S : nivel

术语层是抽象层的同义词。

level of abstraction 抽象层

F : niveau d'abstraction

S : nivel de abstracción

抽象层是功能块树图层之一。系统的描述是最高抽象层上的功能块，它用功能块树图顶端的那个功能块来表示。[3.2.1]

lexical rules 词法规则

F : règles lexicales

S : reglas léxicas

词法规则是定义如何以字符组成词法单元的规则。[2.2.1，4.2.1]

lexical unit 词法单元

F : unités lexicales

S : unidad léxica

词法单元是具体正文语法的终结符。[2.2.1]

literal 字面值

F : littéral

S : literal

字面值是指一个值本身。[2.3.3，5.1.2，5.4.1.14]

macro 宏

F : macro

S : marco

宏是多个语法项或正文项的一个命名集合，要先用它来取代宏调用然后才考虑 SDL 表示的意思（即仅当置于一段特定的上下文中时，宏才有意义）。[4.2]

macro call 宏调用

F : appel de macro

S : llamada a (de) macro

宏调用指出了一个位置，在这个位置将展开带有同样名字的宏定义。[4.2.3]

macro definition　宏定义

F : définition de macro

S : definición de macro

宏定义是在SDL/PR中的宏的定义。[4.2.2]

macro diagram　宏图

F : diagramme de macro

S : diagrama de macro

宏图是在SDL/GR中宏的定义。[4.2.2]

Make!

F : make!

S : hacer!; make!

Make!，是仅用于数据类型定义中的一个操作，用来构成一个复合类型之一个值（例如，结构类别）。[5.4.1.10，5.6.8]

merge area　汇合区

F : zone de fusion

S : área de fusión

汇合区是某一流线连接到另一流线的地方。[2.6.7.2.2]

Meta IV

F : Meta IV

S : Meta IV

Meta IV是用来表示一种语言的抽象语法的形式化的表示法。[1.5.1]

model　模型

F : modèle

S : modelo

模型给出了根据预先定义的具体语法表达出的简化符号的映象。[1.4.1，1.4.2]

modify!

F : modify!

S : modificar!; modify!

modify!（修改!）是一个运算符。当一个变量之后紧跟着括号括起来的表达式然后是符号“：=”时，表达式中将隐含有modify!运算符。在公理中，modify!运算符的使用是显式的（参见Extract!）。[5.4.1.10，5.6.8]

name 名字

F : nom

S : nombre

名字是一个用来命名 SDL 实物的词法单元。[2.2.1，2.2.2]

natural 自然数

F : naturel

S : natural

自然数是在预定义的部分类型定义中所定义的同义类型，其值是非负整数（即 0，1，2，……）。其运算符就是整数类别的运算符。[5.6.6]

newtype 新类型

F : nouveau type (newtype)

S : niotipo

新类型引入一个类别，一组运算符和一组等式。要注意的是术语新类型很可能使人误解，这是因为实际上引入的是一个新的类别，但因历史原因仍沿用新类型一词。[5.2.1]

node 节点

F : noeud

S : nodo

在抽象语法中，一个节点是 SDL 基本概念之一的一个标志。

note 注

F: note

S: nota

注是由/ * 和 * /括起来的正文，它不具有 SDL 定义的语义。参见注释。[2.2.1]

null 空

F : null

S : null; nulo

null 是 PId 类别的字面值。[5.6.10]

OFFSPRING

F : DESCENDANT (OFFSPRING)

S : OFFSPRING; VASTAGO

OFFSPRING 是 PId 类别的一个表达式。当在某进程中求 OFFSPRING 值的时候，它给出由这个进程最近所创建之进程的 PId 值。如果该进程没有创建任何进程，则 OFFSPRING 的计算结果是 Null [2.4.4，5.5.4.3]

operator　运算符

F : opérateur

S : operador

运算符是操作的表示符号。在部分类型定义中给出运算符的定义，例如，+，－，*，/ 都是为整数类别定义的运算符的名字。[5.1.2，5.1.3]

operator signature　运算符标记

F : signature d'opérateur

S : signatura de operador

运算符标记定义了该运算符可以适用的值的类别和运算结果值的类别。[5.2.2]

option　任选

F : option

S : opción

任选是在类属的 SDL 系统规格中的具体语法构件，它允许在系统进行解释以前选择不同的系统结构。[4.3.3，4.3.4]

ordering operator　排序运算符

F : opérateurs de relation d'ordre

S : operadores de ordenación

排序运算符是<，<=，>或>=。[5.4.1.8]

out connector　出连接符

F : connecteur de sortie

S : conector de salida

出连接符是一个连接符。

outlet　引出线

F : accès sortant

S : acceso de salida

引出线表示从某宏图中离去的一条线，例如信道或流线。[4.2.2]

output　输出

F: sortie

S: salida

输出是在跃迁中的动作，它产生信号实例。

output area　输出区

F : zone de sortie
S : área de salida

控制流图中的输出区表示输出的 SDL/GR 概念。[2.7.4]

page　页

F : page
S : página

页是图之物理划分的组成成分。[2.2.5]

PARENT

F : PARENT
S : PARENT; PROGENITOR

PARENT 是一个 PId 表达式。当一个进程求该表达式的值时，其结果是其父辈进程的 PId 值，如果该进程创建于系统初始化之时，则结果为 null。[2.4.4，5.5.4.3]

partial type definition　部分类型定义

F : défintiion partielle de type
S : definición parcial de tipo

对于一个类别，部分类型定义规定了与该类别有关的一些特性。部分类型定义是数据类型定义的一部分。[5.2.1]

partitioning　划分

F : subdivision
S : partición

划分是指把单元再细分成一些较小的部分。当把所有这些部分看成是一个整体时，它具有与原来的单元一样的行为。划分不影响单元的静态界面。[3.1，3.2]

PId　进程实例标识符

F : PId
S : PId

PId 是在预定义的部分类型定义中所定义的类别，它有字面值 null。PId 是进程实例标识符的英文缩写形式，这一类别的值被用来识别进程实例。[5.5.4.3，5.6.10]

powerset　幂集

F : mode ensembliste
S : conjunista

幂集是用来引入数学上的集的预定义生成程序。幂集的运算符是 IN、Incl、Del、并、交和排序运算符。[5.6.9]

predefined data　预定义数据

F : données prédéfinies

S : datos predefinidos

为了简化描述，术语预定义数据既用于由部分类型定义引入的类别的预定义名字，也用于数据类型生成程序的预定义名字。布尔、字符、字符串、持续时间、整数、自然数、PId、实数和时间都是预定义的类别名字。数组、幂集和串都是预定义的数据类型生成程序名字。在所有的SDL系统中，预定义数据隐含地定义在系统层。[5.6]

procedure　过程

F : procédure

S : procedimiento

过程是进程中的一个封闭的行为。一个过程在一个地方给出定义，但可在同一进程中被多次引用。参见形式参数和实在参数。[2.4.5]

procedure call　过程调用

F : appel de procédure

S : llamada a (de) procedimiento

过程调用是指为解释一个命名的过程而调用该过程，同时向它传递实在参数。[2.7.3]

procedure call area　过程调用区

F : zone d'appel de procédure

S : área de llamada a (de) procedimiento

过程调用区是过程调用的SDL/GR表示。[2.7.3]

procedure definition　过程定义

F : définition de procédure

S : definición de procedimiento

过程定义是过程的SDL/PR定义。[2.4.5]

procedure diagram　过程图

F : diagramme de procédure

S : diagrama de procedimiento

过程图是过程的SDL/GR表示。[2.4.5]

procedure graph　过程图形

F : graphe de procédure

S : gráfico de procedimiento

过程图形是在抽象语法中表示过程的一种非终结符号。[2.4.5]

procedure return　过程返回

F : retour de procédure
S : retorno de procedimiento

过程返回是返回的同义词。

process　进程

F : processus
S : proceso

进程是进行通信的扩展有限态自动机。通信可通过信号或共享变量进行。进程的行为取决于在其输入端口处信号到达的次序。[2.4.4]

process area　进程区

F : zone de processus
S : área de proceso

在SDL/GR中，一个进程区用来表示一个进程；在一个相互作用图中，一个进程区用来引用一个进程。[2.4.3]

process definition　进程定义

F : définition de processus
S : definición de processo

进程定义是进程的SDL/PR表示。[2.4.4]

process diagram　进程图

F : diagramme de processus
S : diagrama de proceso

进程图是进程定义的SDL/GR表示。[2.4.4]

process graph　进程图形

F : graphe de processus
S : gráfico de proceso

进程图形是在抽象语法中表示进程的一种非终结符号。[2.4.4]

process instance　进程实例

F : instance de processus
S : instancia de proceso

进程实例是动态地被创建的进程的实例。参见SELF、SENDER、PARENT和OFFSPRING。[2.4.4]

qualifier 限定符

F : partie qualificative (qualificatif)

S : calificador

限定符是标识符的一部分，它是为保证唯一性而附加到标识符名字部分的信息。在抽象语法中，总是给出限定符，但是在具体语法中仅在保证唯一性需要时，才必须使用限定符。例如当一个标识符的限定符不能够从使用名字部分的上下文中导出时。[2.2.2]

real 实数

F : réel

S : real

实数是在预定义的部分类型定义中所定义的一个类别，其值是可以用一个整数除以另一个整数来表示的数。该实数类别的预定义运算符与整数类别的运算符有同样的名字。[5.6.7]

refinement 具体化

F : reaffinement

S : refinamiento

具体化是在某一抽象层上对功能添加新的细节。一个系统的具体化丰富了其行为或使之能处理更多类型的信号和信息，包括处理那些去往（或来自于）环境的信号。请与划分进行比较。[3.3]

remote definition 间接定义

F : définition distante

S : definición remota

间接定义是一种语法手段，它将一个系统定义分布在几个部分，并使这些部分彼此关联。[2.4.1]

reset 复位

F : reset (réinitialisation)

S : reincializar; reponer

复位是为定时器定义的一种操作，它使我们能让定时器处于静止状态。参见活跃定时器。[2.8]

retained signal 保留信号

F : signal retenu

S : señal retenida

保留信号是进程输入端口上的信号，也就是已经接收到、但该进程还没有消耗掉的信号。[2.4.4]

return 返回

F : retour

S : retorno

过程的返回就是把控制交还给调用本过程的过程或进程。[2.6.7.2.4]

reveal attribute　透露属性

F : attribut d'exposition

S : atributo revelado

进程拥有的变量可以有一种透露属性，在此情况下，允许同一功能块中的另一进程视见与该变量相关联的值。参见视见定义。[2.6.1.1]

save　保存

F : mise en réserve

S : conservación

保存是声明那些在一给定状态中不应被消耗掉的信号。[2.6.5]

save area　保存区

F : zone de mise en réserve

S : área de conservación

保存区是保存的 SDL/GR 表示。[2.6.5]

save signal set　保存信号集

F : ensemble de signaux de mise en réserve

S : conjunto de señales de conservación

状态的保存信号集就是那个状态的保存信号的集合。[2.6.5]

SDL（CCITT Specification and Description Language）　CCITT 规格和描述语言

F : LDS (langage de description et de spécification du CCITT)

S : LED (lenguaje de especificación y descripción del CCITT)

CCITT SDL（规格和描述语言）是一种形形语言，它提供了一组关于系的功能度的规格的构件。

SDL/GR

F: LDS/GR

S: LED/GR

SDL/GR 是 SDL 中的图形表示法。SDL/GR 的文法由具体图形文法和公共正文文法确定。[1.2]

SDL/PE

F : LDS/PE

S : LED/EP

SDL/PE 是一组，它可以与 SDL/GR 状态符号连系在一起使用的图像。[附件 E]

SDL/PR

F : LDS/PR

S : LED/PR

SDL/PR 是 SDL 中的正文短语表示法。SDL/PR 的文法由具体正文文法确定。[1.2]

scope unit 作用域单位

F : unité de portee

S : unidad de ámbito

具体文法中的作用域单位确定了标识符的视见范围。作用域单位的例子包括系统、功能块、进程、过程、部分类型定义和服务定义。[2.2.2]

selection 选择

F : sélection

S : selección

选择的意思是给出那些必须的外部同义词，用来从一个类属的系统规格生成一特定系统规格。[4.3.3]

SELF

F : SELF

S : SELF; MISMO

SELF 是一个 PId 表达式。当一个进程计算该表达式的值时，其结果是该进程的 PId 值。SELF 决不会产生 Null。也请参见 PARENT、OFFSPRING、PId。[2.4.4，5.5.4.3]

semantics 语义

F : sémantique

S : semántica

语义给出一个实体的意思，这些意思是：该实体所具有的特性、解释其行为的方法和必须达到的动态条件以使得该实体的行为能满足 SDL 规则。[1.4.1，1.4.2]

SENDER

F : SENDER (émetteur)

S : SENDER; EMISOR

SENDER 是一个 PId 表达式。当计算 SENDER 的值时，将得出启动当前跃迁之信号的发送进程的 PId 值。[2.4.4，2.6.4，5.5.4.3]

service 服务

F : service

S : servicio

服务是制定一个进程的规格时可选用的手段。每一服务可以定义一个进程的部分行为。[4.10]

service area　服务区

F : zone de service

S : área de servicio

一个服务区或者是一个服务图，或者是对一个服务的引用。[4.10.1]

service definition　服务定义

F : définition de service

S : definición de servicio

服务定义是用 SDL/PR 对服务作出的定义。[4.10.1]

service diagram　服务图

F : diagramme de service

S : diagrama de servicio

服务图是用 SDL/GR 对服务作出的定义。[4.10]

set　置定时

F : set (initialisation)

S : inicializar; poner

置定时是为定时器规定的一种操作，它允许我们使定时器变为活跃定时器。[2.8]

shorthand notation　简化符号

F : notation abrégée

S : notación taquigráfica (o abreviada)

简化符号是一种具体语法符号，它给出了一种更简明的表示法，该表示法隐含地联系到基本的 SDL 概念。[1.4.2]

signal　信号

F : signal

S : señal

信号是信号类型的实例，它传递信息给一个进程实例。[2.5.4]

signal definition　信号定义

F : définition de signal

S : definición de señal

信号定义确定了命名信号类型和关联的列表，该列表有零个或多个带有该信号名字的类别标识符。这允许信号携有值。[2.5.4]

signal list　信号表

F : liste de signaux

S : lista de señales

信号表是信号标识符的一个列表，它用在信道和信号路由定义中，以指明所有可以由此信道或信号路由朝一个方向传递的信号。[2.5.5]

signal list area　信号表区

F : zone de liste de signaux

S : área de lista de señales

相互作用图中的信号表区表示与信道或信号路由有关的信号表。[2.5.5]

signal route　信号路由

F : acheminement de signaux

S : ruta de señales

信号路由被用来指明信号的流动：信号在一个进程类型和同一个功能块中的另一个进程类型之间的流动；或是信号在一个进程类型和连接到该功能块的信道之间的流动。[2.5.2]

simple expression　简单表达式

F : expression simple

S : expresión simple

简单表达式是这样一种表达式，它仅包含预定义类别的运算符、同义词和字面值。[4.3.2]

sort　类别

F : sorte

S : género

类别是一组具有共同特性的值。类别总是非空的和不相交的。[2.3.3，5.1.3]

specification　规格

F : spécification

S : especificación

规格是一个系统要求的定义。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组成。规格也可以用作为“规格和（或）描述”的简化形式，例如，在SDL规格或系统规格之中。[1.1]

start　启动

F : départ

S : arranque

在进程中，启动在任一状态或动作之先被解释。启动通过用在创建中规定的实在参数来置换进程的形式参数而初始化该进程。[2.6.2]

state 状态

F : état

S : estado

状态是这样一种状况，在该状况中，一个进程实例可消耗掉一个信号。[2.6.3]

state area 状态区

F : zone d'état

S : área de estado

状态区是一个或多个状态的 SDL/GR 表示。[2.6.3]

state picture 状态图形

F : représentation graphique d'état

S : pictograma de estado

状态图形是一个包括有图形元素的状态符号，它用来把 SDL/GR 扩充到 SDL/PE。[附件 E]

stop 停止

F : arrêt

S : parada

停止是终结进程实例的动作。当对停止进行解释时，进程实例所拥有的所有变量被消除，并且，输入端口的所有保留信号不再使用。[2.6.7.2.3]

string 串

F : chaîne (string)

S : cadena; string

串是用于引入列表的一个预定义的生成程序。其预定义的运算符包括 Length、First、Last、Substring 和 concatenation. [5.6.3]

structured sort 结构类别

F : sorte structurée

S : género estructurado

结构类别是这样一种类别，它具有隐含运算符和等式，以及用于这些隐含运算符的专用的具体语法。采用结构类别通过所谓的字段来生成值。字段的值可以分别地被访问和修改。[5.4.1.10]

subblock 子功能块

F : sous-bloc

S : subbloque

子功能块是包含在另一个功能块中的功能块。当对功能块进行划分时，便形成了子功能块。[3.2.1，3.2.2]

subchannel　子信道

F : sous-canal

S : subcanal

子信道是当功能块被划分时所形成的信道。一个子信道将一个子功能块连接到被划分功能块的一条边界，或是将一个功能块连接到一个被划分信道的边界。[3.2.2，3.2.3]

subsignal　子信号

F : sous-signal

S : subseñal

子信号是信号的具体化产物，它还可以进一步被细分。[3.3]

symbol　符号

F : symbole

S : símbolo

符号是具体语法中的一个终结符。在具体图形语法中，一个符号可以是一组形状中的一个。

synonym　同义词

F : synonyme

S : sinónimo

同义词是代表一个值的名字。[5.4.1.13]

syntax diagram　语法图

F : diagramme de syntaxe

S : diagrama de sintaxis

语法图是具体正文语法之定义的图解形式。[附件 C2]

syntype　同义类型

F : syntype

S : sintipo

一个同义类型决定了一组值，该组值对应于父辈类型之值的一个子集。同义类型的运算符与其父辈类型的那些运算符是一样的。[5.4.19]

system　系统

F : système

S : sistema

一个系统是一组功能块，它们通过信道相互联结并且通过信道与环境连接。

system definition　系统定义

F : définition de système

S : definición de sistema

系统定义是一个系统的 SDL/PR 表示。[2.4.2]

system diagram

F : diagramme de système

S : diagrama de sistema

系统图是一个系统的 SDL/GR 表示。[2.4.2]

task　任务

F : tâche

S : tarea

任务是跃迁中的一个动作，它或是含有一系列赋值语句，或是含有非形式正文。任务的解释取决于系统所拥有的信息并可以作用于这些信息。[2.7.1]

task area　任务区

F : zone de tâche

S : área de tarea

任务区是任务的 SDL/GR 表示。[2.7.1]

term　项

F : terme

S : término

项在语法上等同于表达式。项仅用于公理中，并且，为了清晰起见，将项和表达式区分开。[5.2.3，5.3.3]

text extension symbol　正文扩展符号

F : symbole d'extension de te

S : símbolo de ampliación de t

正文扩展符是用来放置正文的，这些正文属于正文扩展符所依附的那个图形符号。正文扩展符中的正文是它所依附的那个符号中的正文的延续。[2.2.7]

time　时间

F : temps (time)

S : tiempo; time

时间是在预定义的部分类型定义中所定义的一种类别，其值表示为实数值。对时间和持续时间的预定义运算符是＋和－。[5.5.4.1，5.6.12]

timer　定时器

F : temporisateur

S : temporizador

定时器是进程实例所拥有的一个实物，它可以是活跃的，也可以是静止的。一个活跃定时器在一特定的时刻返回一个定时器信号给拥有它的进程实例。也请参见 set 和 reset。[2.8，5.5，4.5]

transition　跃迁

F : transition

S : transición

跃迁是一个动作序列，它出现在当一个进程实例从一种状态变换到另一种状态之时。[2.6.7.1]

transition area　跃迁区

F : zone de transition

S : àrea de transición

跃迁区是跃迁的 SDL/GR 表示。[2.6.7.1]

transition string　跃迁串

F : chaîne de transition

S : cadena de transición

跃迁串是零个或多个动作的一个序列。[2.6.7.1]

transition string area　跃迁串区

F : zone de chaîne de transition

F : área de cadena de transición

跃迁串区是跃迁串的 SDL/GR 表示。[2.6.7.1]

type　类型

F : type

S : tipo

类型是实体的一组特性。在 SDL 中，类型的类包括功能块、信道、信号路由、信号和系统。[1.3.1]

type definition　类型定义

F : définition de type

S : definición de tipo

类型定义规定了一个类型的特性。[1.3.1]

undefined　未定义

F : indéfini (undefined)

S : indefinido

未定义是每一种类别的一个“特殊的”值，它指的是那个类别的一个变量还没有被赋予一个正常的值。参见 access。[5.5.2.2]

valid input signal set　有效输入信号集

F : ensemble de signaux d'entrée valides

S : conjunto de señales de entrada válidas

进程的有效输入信号集是在该进程中由任一输入处理的所有外部信号的名单。它由指向该进程的信号路由中的那些信号组成。请与完整有效输入信号集作比较。[2.4.4，2.5.2]

valid specification　合法规格

F : spécification valide

S : especificación válida

合法规格是遵循具体语法和遵守静态良形式规则的规格。[1.3.3]

value　值

F : valeur

S : valor

类别的值是与该类别的一个变量相关联的多个值中的一个，它可以与需要该类别的一个值的运算符一起使用。值是解释一个表达式的结果。[2.3.3，5.1.3]

variable　变量

F : variable

S : variable

变量是进程实例或过程所拥有的实体，它可以通过赋值语句关联于一个值。当被访问时，变量将给出最后赋给它的值。[2.3.2]

variable definition　变量定义

F : définition de variable

S : definición de variable

变量定义表明：在含有该定义的进程、过程或服务之中，所列出的变量名字将是可视见的。[2.6.1.1]

view definition　视见定义

F : définition de visibilité

S : definición de visión

视见定义规定了在另一个进程中具有透露属性的一个变量标识符。这将允许视见进程访问那个变量的值。[2.6.1.2]

view expression　视见表达式

F : expression de vue

S : expresión de visión

在一个表达式中使用视见表达式以获得一个被视见变量的当前值。[5.5.4.4]

visibility　视见度

F : visibilité

S : visibilidad

标识符的视见度是在其中可以使用此标识符的作用域单位(一个或多个)。在同一个作用域单位中并且是属于同一个实体类的定义，不允许有两个定义具有同样的名字。[2.2.2]

well-formedness rules　良形式规则

F : règles de bonne formation

S : reglas de formación correcta

良形式规则是在具体语法上施加的约束用来实施不直接由语法规则表达的静态条件。[1.4.1，1.4.2]

附 件 B

（附于建议 Z.100）

抽象语法概要

```
标识符          :: 限定符名
限定符          =  路径项+
路径项          =  系统限定符|
                   功能块限定符|
                   功能块子结构限定符|
                   信号限定符|
                   进程限定符|
                   过程限定符|
                   类别限定符
系统限定符      :: 系统名
功能块限定符    :: 功能块名
功能块子结构限  :: 功能块子结构名
定符
进程限定符      :: 进程名
过程限定符      :: 过程名
信号限定符      :: 信号名
类别限定符      :: 类别名
名字            :: 记号
非形式正文      :: …
系统定义        :: 系统名
                   功能块定义 set
                   信道定义 set
                   信号定义 set
                   数据类型定义
                   同义类型定义 set
系统名          =  名字
功能块定义      :: 功能块名
                   进程定义 set
```

信号定义 **set**
信道至路由连接 **set**
信号路由定义 **set**
数据类型定义
同义类型定义 **set**
[功能块子结构定义]

功能块名 = 名字

进程定义 :: 进程名
实例数
进程形式参量 *
进程定义 **set**
信号定义 **set**
数据类型定义
同义类型定义 **set**
变量定义 **set**
视见定义 **set**
定时器定义 **set**
进程图形

实例数 :: 整数 整数

进程名 = 名字

进程图形 :: 进程起始节点
状态节点 **set**

进程形式参量 :: 变量名
类别引用标识符

过程定义 :: 过程名
过程形式参量 *
过程定义 **set**
数据类型定义
同义类型定义 **set**
变量定义 **set**
过程图形

过程名 = 名字

过程形式参量 = *In* 参量 |
Inout 参量

In 参量 :: 变量名
类别引用标识符

Inout 参量 :: 变量名
类别引用标识符

过程图形 :: 过程起始节点
状态节点 **set**

过程起始节点 :: 跃迁

信道定义 :: 信道名
信道路径
[信道路径]

信道路径 :: 发端功能块
目的地功能块
信号标识符 **set**

发端功能块 = 功能块标识符 |
ENVIRONMENT

目的地功能块 = 功能块标识符 |
ENVIRONMENT

功能块标识符 = 标识符

信号标识符 = 标识符

信道名 = 名字

信号路由定义 :: 信号路由名
信号路由路径
[信号路由路径]

信号路由路径 :: 发端进程
目的地进程
信号标识符 **set**

发端进程 = 进程标识符 |
ENVIRONMENT

目的地进程 = 进程标识符 |
ENVIRONMENT

信号路由名 = 名字

信道至路由连接 :: 信道标识符
信号路由标识符 **set**

信号路由标识符 = 标识符

信号定义 :: 信号名
类别引用标识符 *
[信号具体化]

信号名 = 名字

变量定义 :: 变量名字
类别引用标识符
[REVEALED]

变量名 = 名字

视见定义 :: 变量标识符
类别引用标识符

进程起始节点 :: 跃迁

状态节点 :: 状态名
保存信号集
输入节点 **set**

状态名 = 名字

输入节点 :: 信号标识符
[变量标识符] *
跃迁

变量标识符 = 标识符

保存信号集 :: 信号标识符 **set**

跃迁 :: 图形节点 *
(终端符|判定节点)

图形节点 :: 任务节点|
输出节点|
创建请求节点|
调用节点|
置定时节点|
复位节点

终端符 :: 下一状态节点|
停止节点|
返回节点

下一状态节点 :: 状态名

返回节点 :: ( )

停止节点 :: ( )

任务节点 :: 赋值语句|
非形式正文

创建请求节点 :: 进程标识符
[表达式] *

进程标识符 = 标识符

调用节点 :: 过程标识符

[表达式] *

过程标识符 = 标识符

判定节点 :: 判定问题
判定回答 **set**
[Else 回答]

判定问题 = 表达式|
非形式正文

判定回答 :: (范围条件|非形式正文) 跃迁

Else 回答 :: 跃迁

输出节点 :: 信号标识符
[表达式] *
[信号目的地]
传送径由

信号目的地 = 表达式

传送径由 = 信号路由标识符 **set**

定时器定义 :: 定时器名
类别引用标识符 *

定时器名 = 名字

置定时节点 :: 时间表达式
定时器标识符
表达式 *

复位节点 :: 定时器标识符
表达式 *

定时器标识符 = 标识符

时间表达式 = 表达式

功能块子结构定义 :: 功能块子结构名
子功能块定义 **set**
信道连接 **set**
信道定义 **set**
信号定义 **set**
数据类型定义
同义类型定义 **set**

功能块子结构名 = 名字

子功能块定义 = 功能块定义

信道连接 :: 信道标识符
子信道标识符 **set**

```
子信道标识符        = 信道标识符
信道标识符          = 标识符
信号具体化          :: 子信号定义 set
子信号定义          :: 〔REVERSE〕信号定义
数据类型定义        :: 类型名
                       类型并集
                       类别集
                       标记 set
                       等式集
类型并集            = 类型标识符 set
类型标识符          = 标识符
类别集              = 类别名 set
类型名              = 名字
类别名              = 名字
等式集              = 等式 set
标记                = 字面值标记 |
                       运算符标记
字面值标记          :: 字面值运算符名
                       结果
运算符标记          :: 运算符名
                       变元表
                       结果
变元表              = 类别引用标识符+
结果                = 类别引用标识符
类别引用标识符      = 类别标识符 |
                       同义类型标识符
字面值运算符名      = 名字
运算符名            = 名字
类别标识符          = 标识符
等式                = 非量化等式 |
                       量化等式集 |
                       条件等式 |
                       非形式正文
非量化等式          :: 项
                       项
量化等式集          :: 值名字 set
```

类别标识符
等式集

值名字 = 名字

项 = 基本项|
复合项|
错误项

复合项 :: 值标识符|
运算符标识符 项+|
条件复合项

值标识符 = 标识符

运算符标识符 = 标识符

基本项 :: 字面值运算符标识符
运算符标识符基本项+|
条件基本项

字面值运算符标识符 = 标识符

条件等式 :: 限制条件 **set**
受限等式

限制条件 = 非量化等式

受限等式 = 非量化等式

条件复合项 = 条件项

条件基本项 = 条件项

条件项 :: 条件
后果项
替换项

条件 = 项

后果项 = 项

替换项 = 项

错误项 :: ()

同义类型标识符 = 标识符

同义类型定义 :: 同义类型名
父辈类别标识符
范围条件

同义类型名 = 名字

父辈类别标识符 = 类别标识符

范围条件 :: **Or** 运算符标识符
条件项 **set**

```
条件项              = 开范围|闭范围
开范围              :: 运算符标识符
                       基本表达式
闭范围              :: And 运算符标识符
                       开范围
                       开范围
Or 运算符标识符     = 标识符
And 运算符标识符    = 标识符
表达式              = 基本表达式|
                       主动表达式
基本表达式          = 基本项
变量访问            = 变量标识符
主动表达式          = 变量访问|
                       条件表达式|
                       运算符应用|
                       命令运算符
命令运算符          = NOW 表达式|
                       PId 表达式|
                       视见表达式|
                       定时器活跃表达式
NOW 表达式          :: ()
PId 表达式          = 自身表达式|
                       父辈表达式|
                       后代表达式|
                       发送者表达式
自身表达式          :: ()
父辈表达式          :: ()
后代表达式          :: ()
发送者表达式        :: ()
视见表达式          :: 变量标识符
                       表达式
定时器活跃表达式    :: 定时器标识符
                       表达式 *
条件表达式          :: 布尔表达式
                       后果表达式
                       替换表达式
```

布尔表达式 = 表达式

后果表达式 = 表达式

替换表达式 = 表达式

运算符应用 :: 运算符标识符
表达式+

赋值语句 :: 变量标识符
表达式

附 件 C1

（附于建议 Z.100）

具体图形语法概要

C1.1 引言

C1.1.1 元语言

对于图形文法，用下面的元符号对 SDL 建议 § 1.5.2 中所描述的元语言进行了扩展：

a) **contains** （包含）

b) **is associated with** （关联于）

c) **is followed by** （后随）

d) **is connected to** （连接到）

e) **set** （集合）

set 元符号是一个后缀运算符，它作用于在它前面紧挨着它的波形括号内的语法元素上，指明是由一些项目组成的一个（无序的）集合。这种项目可以是任意的语法元素，在这种情况下，它必须在 **set** 元符号之前应用。例如

{{〈系统正文区〉}*{〈宏图〉} * 〈功能块相互作用区〉} **set**

是一个集合，其元素为零个或多个〈系统正文区〉、零个或多个〈宏图〉以及一个〈功能块相互作用区〉。

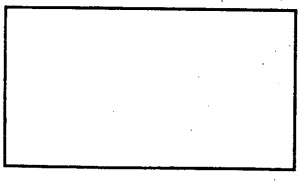
所有其它的元符号都是中缀运算符，它们有一个图形非终结符作为左边变元。右边变元或是一组在波形括号中的语法元素、或是一个单个的语法元素。如果一个产生式规则的右边有一图形非终结符作为第一个元素，并含有一个或多个这些中缀运算符，则此图形非终结符是每一个这些中缀运算符的左边变元。图形非终结符是一个非终结符，在其中紧接在大于号＞之前带有一个字“symbol”（“符号”）。

元符号 **contains** 表示其右边变元应放在其左边变元和可能有的相关的〈正文扩展符〉之中。例如：

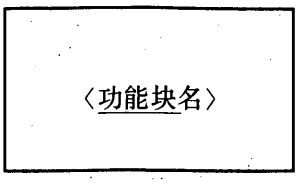
〈功能块引用〉 ::=

〈功能块符号〉 **contains** 〈功能块名字〉

〈功能块符号〉 ::=



其含义如下



元符号 **is associated with** 表示其右边变元在逻辑上是关联于其左边变元的（仿佛它“包含”在那个变元中，其明确的联系由适当的绘图规则来保证）。

元符号 **is followed by** 意指其右边变元（在逻辑上和画法上）跟在其左边变元的后面。

元符号 **is connected to** 意指其右边变元（在逻辑上和画法上）被连接到其左边变元。

C1.1.2 一般规则

C1.1.2.1 图的划分

下面的图划分定义并非具体图形文法的一部分，但也使用了同样的元语言。

〈页〉 ::=
〈框架符〉 **contains**
〈标题区〉〈页号区〉
{〈语法单元〉} *

〈标题区〉 ::=
〈隐式正文符〉 **contains** 〈标题〉

〈页号区〉 ::=
〈隐式正文区〉 **contains**[〈页号〉[(〈总页数〉)]]

〈页号〉 ::=
〈字面值名〉

〈总页数〉 ::=
〈自然数字面值名〉

〈页〉是一个起始非终结符，因此，它在任何产生式规则中都不涉及。一个图可以分成若干〈页〉，在这种情况下，确定图之界限的〈框架符〉和图的〈标题〉由每一页的一个〈框架符〉和一个〈标题〉所取代。

〈隐式正文符〉并没有画出，它是隐含的，使用它是为了清晰地区分〈标题区〉和〈页号区〉。〈标题区〉放在〈框架符〉的左上角。〈页号区〉放在〈框架符〉的右上角。〈标题〉及〈语法单元〉决定于图的类型。

C1.1.2.2　注释

〈注释区〉　::=
　　〈注释符〉 **contains** 〈正文〉
　　is connected to 〈虚线关联符〉
〈注释符〉　::=

〈虚线关联符〉　::=

虚线关联符的一端必须连接到〈注释符〉垂直段的中点。

〈注释符〉通过一〈虚线关联符〉连接到任意的图形符号。通过（在想象中）把〈注释符〉补画成矩形，我们可以把〈注释符〉看成是一个闭合的符号，它包围着与图形符号有关的注释正文。

C1.1.2.3　正文扩展

〈正文扩展区〉::=
　　〈正文扩展符〉 **contains** 〈正文〉
　　is connected to 〈实线关联符〉
〈正文扩展符〉::=
　　〈注释符〉
〈实线关联符〉::=

〈正文扩展符〉可通过〈实线关联符〉连接到任一种图形符号。通过（在想象中）把〈正文扩展符〉补画成矩形，我们可以把〈正文扩展符〉看成是一个闭合的符号。

〈实线关联符〉的一端必须连接到〈正文扩展符〉之垂直段的中点。

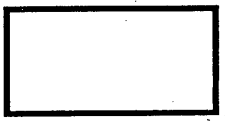
包含在〈正文扩展符〉中的正文是图形符号中正文的继续，并被认为是包含在那个图形符号之中。

C1.2 系统定义

〈具体系统定义〉::=
{〈系统定义〉|〈系统图〉}{〈间接定义〉}*
〈间接定义〉::=
〈定义〉
| 〈图〉
〈图〉 ::=
〈功能块图〉
| 〈进程图〉
| 〈过程图〉
| 〈功能块子结构图〉
| 〈信道子结构图〉
| 〈服务图〉
| 〈宏图〉

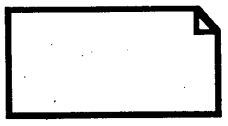
C1.3 系统图

〈系统图〉::=
〈框架符〉 **contains**
〈系统标题〉
{ { 〈系统正文区〉} *
{ 〈宏图〉} *
〈功能块相互作用区〉} **set**
〈框架符〉::=



〈系统标题〉::=
SYSTEM 〈系统名〉
〈系统正文区〉::=
〈正文符〉 **contains**
{ 〈信号定义〉
| 〈信号表定义〉
| 〈数据定义〉
| 〈宏定义〉
| 〈选择定义〉} *

〈正文符〉 ::=



〈功能块相互作用区〉 ::=

{〈功能块区〉 | 〈信道定义区〉}$^{+}$

〈功能块区〉 ::=

〈图形功能块引用〉

| 〈功能块图〉

〈图形功能块引用〉 ::=

〈功能块符〉 **contains** 〈功能块名〉

〈功能块符〉 ::=

〈信道定义区〉 ::=

〈信道符〉

is associated with{〈信道名〉

{ [{ 〈信道标识符〉 | 〈功能块标识符〉}]

〈信道表区〉

[〈信号表区〉]} **set**}

is connected to {〈功能块区〉

{ 〈功能块区〉 | 〈框架符〉}

[〈信道子结构关联区〉]} **set**

此〈信道标识符〉给一个外部信道做出标志，该外部信道连接到由〈框架符〉定界的〈功能块子结构图〉。〈功能块标识符〉给一个外部功能块做出标志，对于由〈框架符〉定界的〈信道子结构图〉来说，这个外部功能块是一个信道端点。

〈信道符〉 ::=

〈信道符 1〉

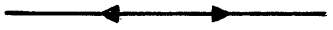
| 〈信道符 2〉

| 〈信道符 3〉

〈信道符 1〉 ::=

〈信道符 2〉 ::=

〈信道符 3〉 ::=



〈信号表区〉 ::=

〈信号表符〉 **contains** 〈信号表〉

〈信号表符〉 ::=

C1.4 功能块图

〈功能块图〉 ::=

〈框架符〉

contains { 〈功能块标题〉

{{ 〈功能块正文区〉}*

{ 〈宏图〉}*

[〈进程相互作用区〉]

[〈功能块子结构区〉]} **set**}

is associated with {〈信道标识符〉}*

此〈信道标识符〉标识了一个信道，此信道连接到在此〈功能块图〉中的一个信号路由。要把〈信道标识符〉放在〈框架符〉之外，靠近在〈框架符〉上的信号路由端点。如果〈功能块图〉不包含〈进程相互作用区〉，则它必须包含一个〈功能块子结构区〉。

〈功能块标题〉 ::=

BLOCK { 〈功能块名〉 | 〈功能块标识符〉}

〈功能块正文区〉 ::=

〈系统正文区〉

〈进程相互作用区〉 ::=

{ 〈进程区〉

| 〈创建线区〉

| 〈信号路由定义区〉} +

〈进程区〉 ::=

〈图形进程引用〉

| 〈进程图〉

〈图形进程引用〉 ::

〈进程符〉 **contains** { 〈进程名〉 [〈实例数〉]}

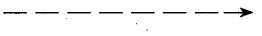
〈进程体〉 ::=

〈创建线区〉 ::=

〈创建线符〉

is connected to ｛〈进程区〉〈进程区〉｝

〈创建线符〉 ::=



〈信号路由定义区〉 ::=

〈信号路由符〉

is associated with ｛〈信号路由名〉

｛[〈信道标识符〉

〈信号表区〉

〈信号表区〉]｝**set**｝

is connected to

｛〈进程区〉｛〈进程区〉｜〈框架符〉｝｝**set**

当此〈信号路由符〉连接到〈框架符〉时，在此处用此〈信道标识符〉来标识连接到此信号路由的信道。

〈信号路由符〉 ::=

〈信号路由符 1〉

｜〈信号路由符 2〉

〈信号路由符 1〉 ::=

〈信号路由符 2〉 ::=

C1.5 进程图

〈进程图〉 ::=

〈框架符〉

contains｛〈进程标题〉

｛｛〈进程正文区〉｝*

｛〈过程区〉｝*

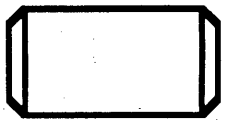
｛〈宏图〉｝*

｛〈进程图形区〉｜〈服务相互作用区〉｝｝**set**｝

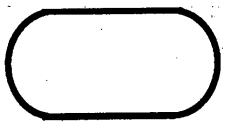
[**is associated with** ｛〈信号路由标识符〉｝$^{+}$]

此〈信号路由标识符〉标识了一个外部信号路由，后者连接到在〈进程图〉中的一个信号路由。把此标识符放在〈框架符〉外，靠近信号路由在〈框架符〉上的端点。

〈进程标题〉::=
PROCESS｛〈进程名〉|〈进程标识符〉｝
[〈实例数〉〈结束〉][〈形式参量〉]
〈进程正文区〉::=
〈正文符〉**contains**[〈有效输入信号集〉]
｛〈信号定义〉
|〈信号表定义〉
|〈变量定义〉
|〈视见定义〉
|〈进口定义〉
|〈数据定义〉
|〈宏定义〉
|〈定时器定义〉
|〈选择定义〉｝*
〈图形过程引用〉::=
〈过程符〉**contains**〈过程名〉
〈过程符〉::=



〈进程图形区〉::=
〈起始区〉｛〈状态区〉|〈入连接符区〉｝*
〈起始区〉::=
〈起始符〉**is followed by**〈跃迁区〉
〈起始符〉::=



〈跃迁区〉::=
[〈跃迁串区〉]**is followed by**
｛〈状态区〉
|〈下一状态区〉
|〈判定区〉
|〈停止符〉
|〈合并区〉
|〈出连接符区〉
|〈返回符〉
|〈跃迁任选区〉

〈合并区〉::=

〈合并符〉 **is connected to** 〈流线符〉

〈合并符〉::=

〈流线符〉

〈流线符〉::=

———————

〈跃迁串区〉::=

{ 〈任务区〉

| 〈输出区〉

| 〈优先输出区〉

| 〈置定时区〉

| 〈复位区〉

| 〈出口区〉

| 〈创建请求区〉

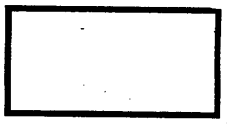
| 〈过程调用区〉

[**is followed by** 〈跃迁串区〉]

〈任务区〉::

〈任务符〉 **contains** 〈任务体〉

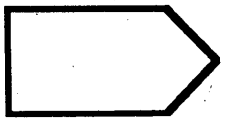
〈任务符〉::=



〈输出区〉::=

〈输出符〉 **contains** 〈输出体〉

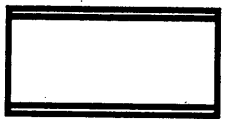
〈输出符〉::=



〈创建请求区〉::=

〈创建请求符〉 **contains** 〈创建体〉

〈创建请求符〉::=



〈过程调用区〉 ::=

〈过程调用符〉 **contains** 〈过程调用体〉

〈过程调用符〉 ::=

〈状态区〉 ::=

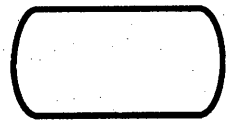
〈状态符〉 **contains** 〈状态表〉 **is associated with** { 〈输入关联区〉

| 〈优先输入关联区〉

| 〈连续信号关联区〉

| 〈保存关联区〉} *

〈状态符〉 ::=



〈输入关联区〉 ::=

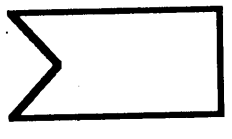
〈实线关联符〉 **is connected to** 〈输入区〉

〈输入区〉 ::=

〈输入符〉 **contains** 〈输入表〉

is followed by{[〈允许条件区〉] 〈跃迁区〉}

〈输入符〉 ::=



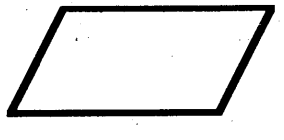
〈保存关联区〉 ::=

〈实线关联符〉 **is connected to** 〈保存区〉

〈保存区〉 ::=

〈保存符〉 **contains** 〈保存表〉

〈保存符〉 ::=



〈入连接符区〉::=

〈入连接符〉 **contains** 〈连接符名〉

is followed by 〈跃迁区〉

〈入连接符〉::=

〈下一状态区〉::=

〈状态符〉 **contains** 〈下一状态体〉

〈判定区〉::=

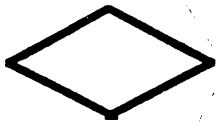
〈判定符〉 **contains** 〈问题〉

is followed by

{{〈图形回答部分〉〈图形 Else 部分〉}**set**

| {〈图形回答部分〉{〈图形回答部分〉}+[〈图形 Else 部分〉]}**set**}

〈判定符〉::=



〈图形回答〉::=

〈回答〉 | (〈回答〉)

〈图形回答部分〉::=

〈流线符〉 **is associated with** 〈图形回答〉

is followed by 〈跃迁区〉

〈图形 Else 部分〉::=

〈流线符〉 **is associated with** ELSE

is followed by 〈跃迁区〉

〈置定时区〉::=

〈任务符〉 **contains** 〈置定时〉

〈复位区〉::=

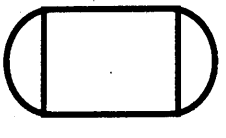
〈任务符〉 **contains** 〈复位〉

〈停止符〉::=

〈出连接符区〉::=
〈出连接符〉 **contains** 〈连接符名〉
〈出连接符〉::=
〈入连接符〉

C1.6 过程图

〈过程图〉::=
〈框架符〉 **contains** {〈过程标题〉
{{〈过程正文区〉}*
{〈过程区〉}*
{〈宏图〉}*
〈过程图形区〉} **set**}
〈过程标题〉::=
PROCEDURE{〈过程名〉|〈过程标识符〉}
[〈过程形式参量〉]
〈过程区〉::=
〈图形过程引用〉
|〈过程图〉
〈过程正文区〉::=
〈正文符〉 **contains**
{〈变量定义〉
|〈数据定义〉
|〈宏定义〉
|〈选择定义〉}*
〈过程图形区〉::=
〈过程起始区〉{〈状态区〉|〈入连接符区〉}*
〈过程起始区〉::=
〈过程起始符〉 **is followed by** 〈跃迁区〉
〈过程起始符〉::=



〈返回符〉::=

C1.7　功能块子结构

〈功能块子结构区〉::=
　　〈图形功能块子结构引用〉
　　| 〈功能块子结构图〉
〈图形功能块子结构引用〉::=
　　〈功能块子结构符〉**contains**〈功能块子结构名〉
〈功能块子结构符〉::=
　　〈功能块符〉
〈功能块子结构图〉::=
　　〈框架符〉
　　contains { 〈功能块子结构标题〉
　　{ { 〈功能块子结构正文区〉} *
　　{ 〈宏图〉} *
　　〈功能块相互作用区〉} **set**}
　　is associated with { 〈信道标识符〉} *

〈信道标识符〉标识了一个信道，它连接到在〈功能块子结构图〉中的一个子信道。要把此〈信道标识符〉放在〈框架符〉之外，靠近子信道在〈框架符〉上的端点。

〈功能块子结构标题〉::=
　　SUBSTRUCTURE { 〈功能块子结构名〉| 〈功能块子结构标识符〉}
〈功能块子结构正文区〉::=
　　〈系统正文区〉

C1.8 信道子结构

〈信道子结构关联区〉::=〈虚线关联符〉
is connected to〈信道子结构区〉
〈信道子结构区〉::=
〈图形信道子结构引用〉
| 〈信道子结构图〉
〈图形信道子结构引用〉::=
〈信道子结构符〉**contains**〈信道子结构名〉
〈信道子结构符〉::=
〈功能块符〉
〈信道子结构图〉::=
〈框架符〉
contains{〈信道子结构标题〉
{{〈信道子结构正文区〉}*
{〈宏图〉}*
〈功能块相互作用区〉}**set**}
is associated with{〈功能块标识符〉|ENV}+

此〈功能块标识符〉或 ENV 标识了被划分的信道的端点。要把〈功能块标识符〉放在〈框架符〉外、靠近相关的子信道在〈框架符〉上的端点。

〈信道子结构标题〉::=
SUBSTRUCTURE{〈信道子结构名〉
|〈信道子结构标识符〉}
〈信道子结构正文区〉::=
〈系统正文区〉

C1.9 宏

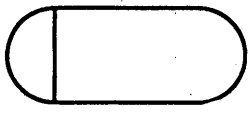
C1.9.1 宏图

〈宏图〉::=
〈框架符〉**contains**〈宏标题〉〈宏体区〉

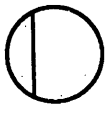
〈宏标题〉::=
MACRODEFINITION〈宏名字〉[〈宏形式参量〉]

〈宏体区〉::=
{ { 〈任意区〉}*
〈任意区〉[**is connected to**〈宏体端口1〉]} **set**
| { 〈任意区〉**is connected to**〈宏体端口2〉
〈任意区〉**is connected to**〈宏体端口2〉
{ 〈任意区〉[**is connected to**〈宏体端口2〉]}* }**set**

〈宏引入线符〉::=



〈宏引出线符〉::=



〈宏体端口1〉::=
〈引出线符〉[**is associated with**〈宏标号〉]
is connected to { 〈框架符〉
| 〈宏引入线符〉
| 〈宏引出线符〉

〈宏体端口2〉::=
〈引出线符〉**is associated with**〈宏标号〉
is connected to { 〈框架符〉
| 〈宏引入线符〉
| 〈宏引出线符〉}

〈宏标号〉::=
〈名字〉

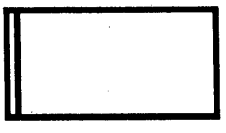
〈引出线符〉::=
〈空引出线符〉
| 〈流线符〉
| 〈信道符〉
| 〈信号路由符〉
| 〈实线关联符〉
| 〈虚线关联符〉
| 〈创建线符〉
〈空引出线符〉::=
〈实线关联符〉
〈任意区〉::=
| 〈系统正文区〉
| 〈功能块相互作用区〉
| 〈信号表区〉
| 〈功能块区〉
| 〈功能块正文区〉
| 〈进程相互作用区〉
| 〈图形过程引用〉
| 〈过程区〉
| 〈进程正文区〉
| 〈进程图形区〉
| 〈合并区〉
| 〈跃迁串区〉
| 〈状态区〉
| 〈输入区〉
| 〈保存区〉
| 〈正文扩展区〉
| 〈信道子结构关联区〉
| 〈信道子结构区〉
| 〈功能块子结构区〉
| 〈优先输入区〉
| 〈连续信号区〉
| 〈入连接符区〉
| 〈下一状态区〉
| 〈进程区〉
| 〈信道定义区〉
| 〈创建线区〉
| 〈信号路由定义区〉
| 〈图形进程引用〉
| 〈进程图〉
| 〈起始区〉
| 〈输出区〉
| 〈置定时区〉
| 〈复位区〉
| 〈出口区〉

| 〈优先输出区〉
| 〈任务区〉
| 〈创建请求区〉
| 〈过程调用区〉
| 〈判定区〉
| 〈出连接符区〉
| 〈过程正文区〉
| 〈过程图形区〉
| 〈过程起始区〉
| 〈功能块子结构正文区〉
| 〈功能块相互作用区〉
| 〈服务区〉
| 〈服务信号路由定义区〉
| 〈服务正文区〉
| 〈服务图形区〉
| 〈服务起始区〉
| 〈注释区〉
| 〈宏调运区〉

C1.9.2 宏调用

〈宏调用区〉 ::=
〈宏调用符〉 **contains** { 〈宏名字〉 [〈宏调用体〉]}
[**is connected to**
{ 〈宏调用端口1〉 | 〈宏调用端口2〉 { 〈宏调用端口2〉} +}]

〈宏调用符〉 ::=



〈宏调用端口1〉 ::=
〈引入线〉 [**is associated with** 〈宏标号〉]
is connected to 〈任意区〉

〈宏调用端口2〉 ::=
〈引入线符〉 **is associated with** 〈宏标号〉
is connected to 〈任意区〉

〈引入线符〉 ::=
〈空引入线符〉
| 〈流线符〉
| 〈信道符〉
| 〈信号路由符〉
| 〈实线关联符〉
| 〈虚线关联符〉
| 〈创建线符〉

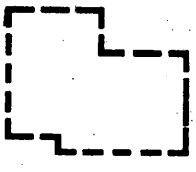
〈空引入线符〉 ::=
〈实线关联符〉

C1.10 类属系统

C1.10.1 任选定义

〈任选区〉::=
〈任选符〉 **contains**
{SELECT IF (〈布尔简单表达式〉)
{〈功能块区〉
|〈信道子结构区〉
|〈系统正文区〉
|〈功能块正文区〉
|〈进程正文区〉
|〈过程正文区〉
|〈功能块子结构正文区〉
|〈信道子结构正文区〉
|〈服务正文区〉
|〈宏图〉
|〈任选区〉
|〈进程区〉
|〈信号路由定义区〉
|〈创建线区〉
|〈过程区〉
|〈服务区〉
|〈服务信号路由定义区〉} +}

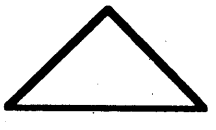
〈任选符〉是一虚线多边形，但拐角处要用实线拐角，例如：



C1.10.2 任选跃迁

〈跃迁任选区〉::=
〈跃迁任选符〉 **contains** {〈备择问题〉}
is followed by {〈任选 引出线1〉{〈任选引出线1〉|〈任选引出线2〉}
{〈任选引出线1〉} * } **set**

〈跃迁任选符〉::=



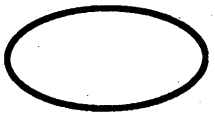
〈任选引出线1〉::=
〈流线符〉 **is associated with** 〈图形回答〉
is followed by {〈跃迁区〉|〈合并区〉}

〈任选引出线2〉 ::=
〈流线符〉 **is associated with** ELSE
is followed by { 〈跃迁区〉 | 〈合并区〉}

C1.11 服务

C1.11.1 服务分解

〈服务相互作用区〉 ::=
{ 〈服务区〉 | 〈服务信号路由定义区〉}+
〈服务区〉 ::=
〈图形服务引用〉
| 〈服务图〉
〈图形服务引用〉 ::=
〈服务符〉 **contains** 〈服务名〉
〈服务符〉 ::=



〈服务信号路由定义区〉 ::=
〈信号路由符〉
is associated with { 〈服务信号路由名〉 [〈信号路由标识符〉]
〈信号表区〉 [〈信号表区〉]} **set**
is connected to { 〈服务区〉
{ 〈服务区〉 | 〈框架符〉}} **set**

当〈信号路由符〉连接到〈框架符〉时，此信号路由连接到一个外部信号路由，而上式中的〈信号路由标识符〉标识了该外部信号路由。

C1.11.2 服务图

〈服务图〉 ::=
〈框架符〉 **contains**
{ 〈服务标题〉
{ { 〈服务正文区〉} *
{ 〈过程区〉} *
{ 〈宏图〉} *
〈服务图形区〉} **set**}
〈服务标题〉 ::=
SERVICE { 〈服务名〉 | 〈服务标识符〉}

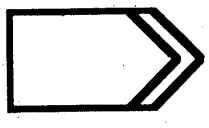
〈服务正文区〉::=
〈正文符〉 **contains**
{〈变量定义〉
|〈视见定义〉
|〈进口定义〉
|〈数据定义〉
|〈宏定义〉
|〈定时器定义〉
|〈选择定义〉} *

〈服务图形区〉::=
〈进程图形区〉

〈优先输入关联区〉::=
〈实线关联符〉 **is connected to**〈优先输入区〉

〈优先输入区〉::=
〈优先输入符〉 **contains**〈优先输入表〉

〈优先输入符〉::=



〈优先输出区〉::=
〈优先输出符〉 **contains**〈优先输出表〉

〈优先输出符〉::=

C1.12 连续信号

〈连续信号关联区〉::=
〈实线关联符〉 **is connected to**〈连续信号区〉

〈连续信号区〉::=
〈允许条件符〉 **contains**
{〈布尔表达式〉[〈结束〉 PRIORITY〈整数 字面值名〉]}
is followed by〈跃迁区〉

C1.13 允许条件

〈允许条件区〉::=
〈允许条件符〉**contains**〈布尔表达式〉
〈允许条件符〉::=

C1.14 出口

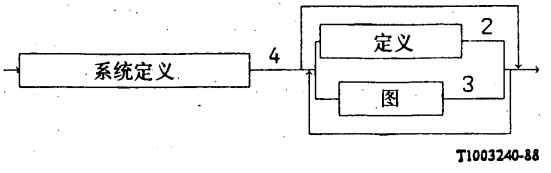
〈出口区〉::=
〈任务符〉**contains**〈出口〉

附 件 C2

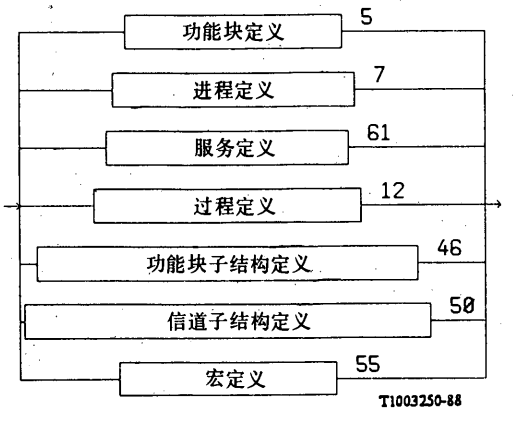
(附于建议 Z.100)

SDL PR 语法概要

1 系统



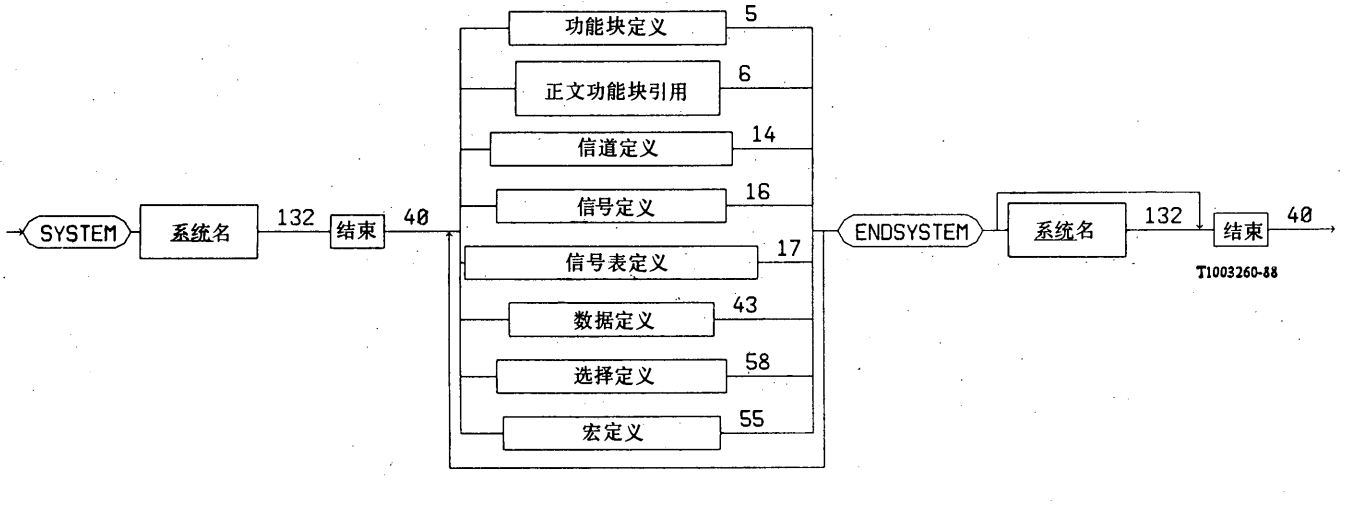
2 定义



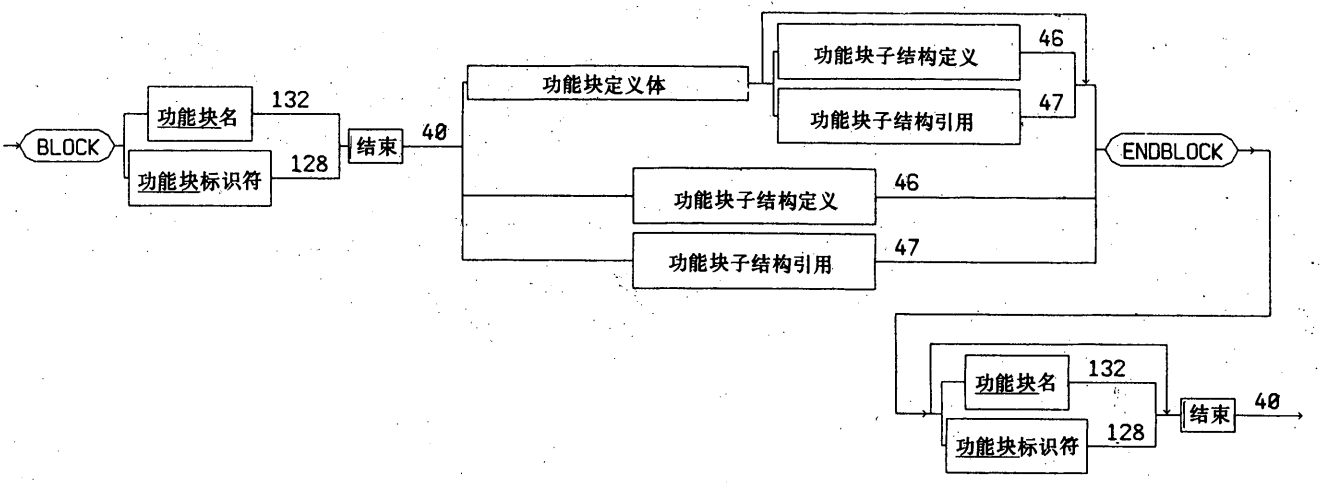
3 图

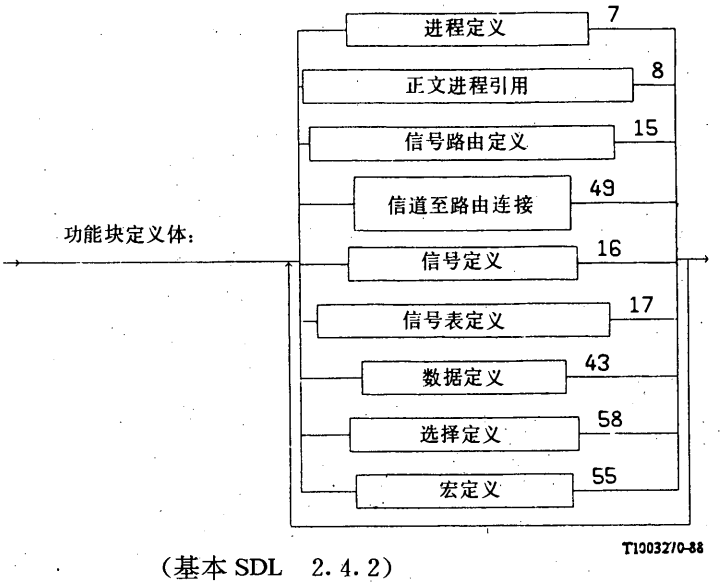
图在 GR 语法概要中定义

4 系统定义 (基本 SDL2.4.1)



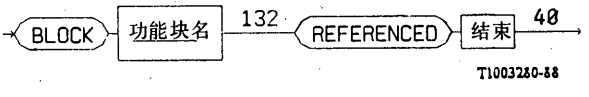
5　功能块定义（基本 SDL2. 4. 2）



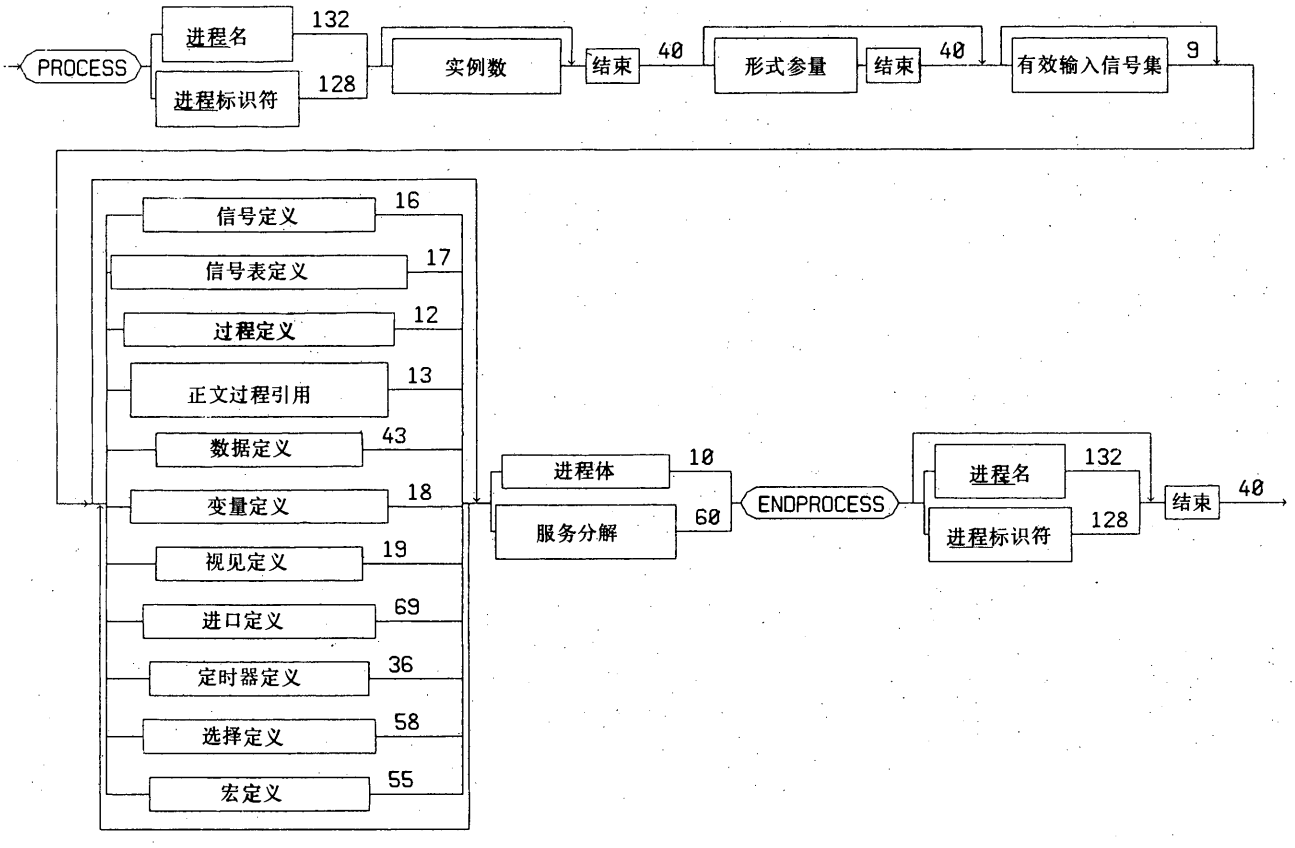


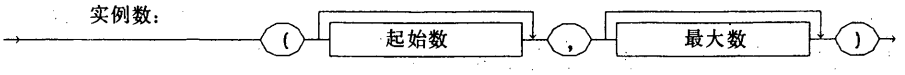
（基本 SDL　2. 4. 2）

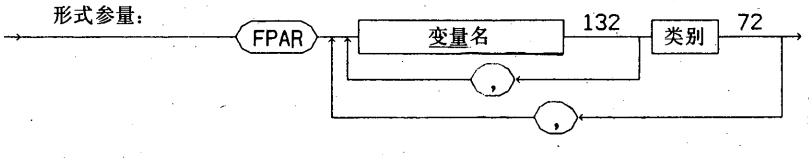
6　正文功能块引用（基本 SDL2. 4. 2）

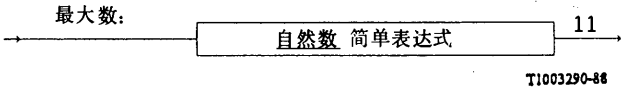


7　进程定义（基本 SDL2.4.3）



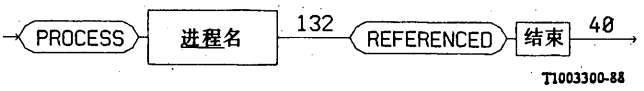




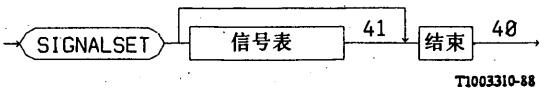


T1003290-88

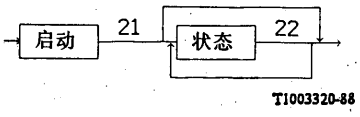
8　正文进程引用（基本 SDL2. 4. 3）



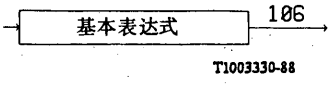
9　有效输入信号集



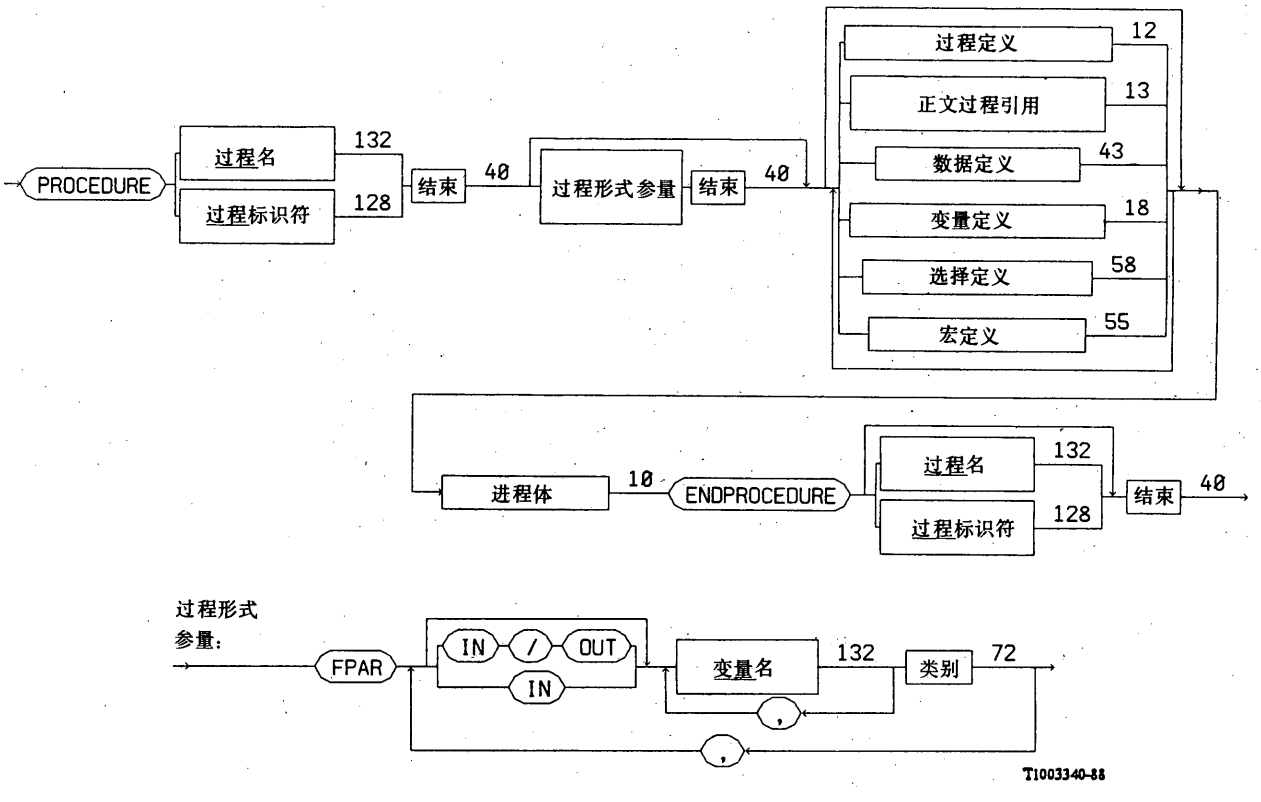
10　进程体（基本 SDL2. 4. 3）



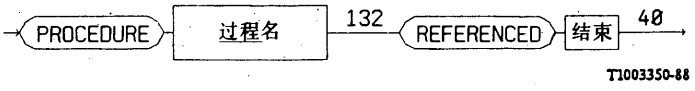
11　简单表达式



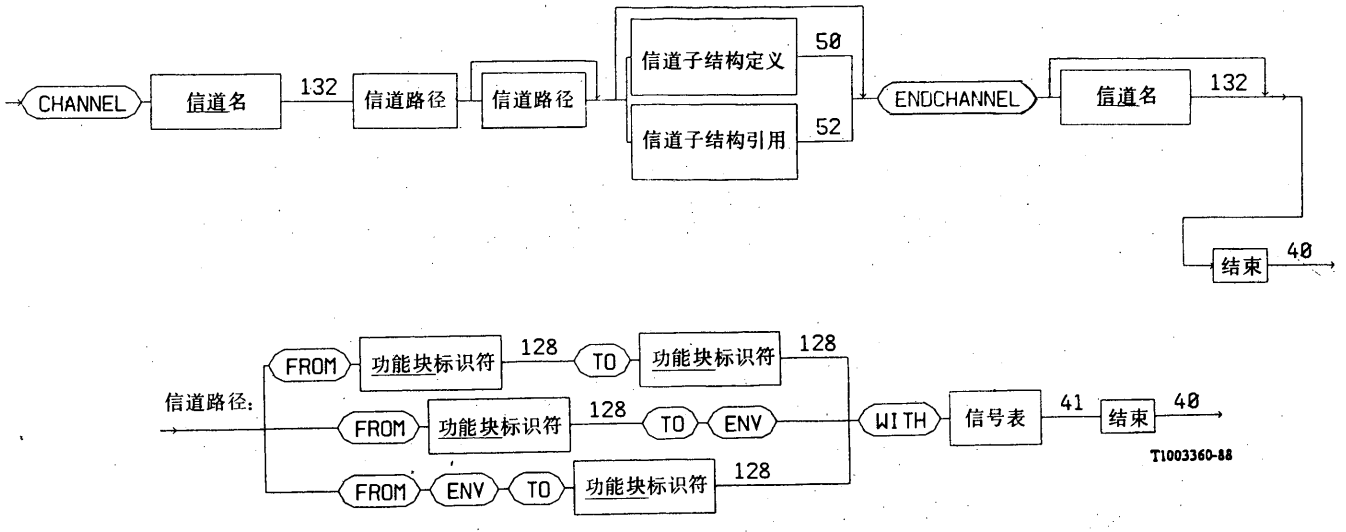
12　过程定义（基本 SDL2. 3. 4）



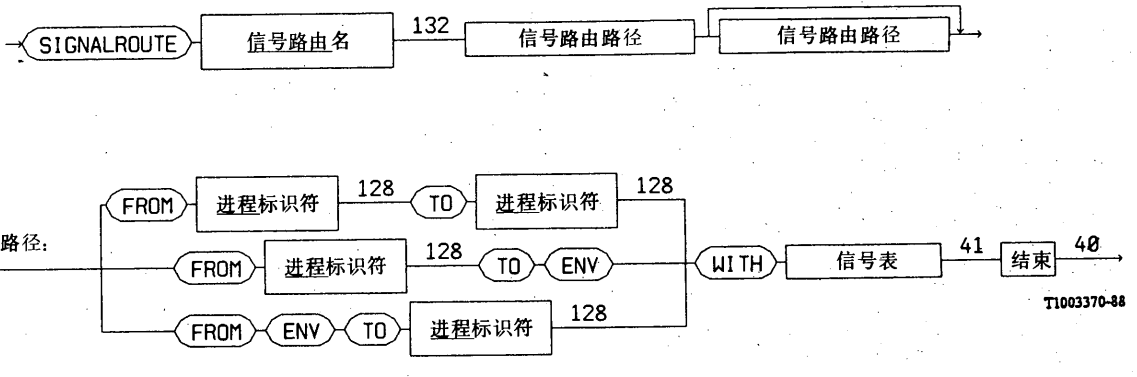
13　正文过程引用（基本 SDL2.3.4）



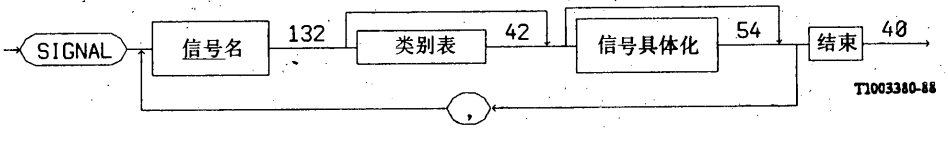
14　信道定义（基本 SDL2.5.1）



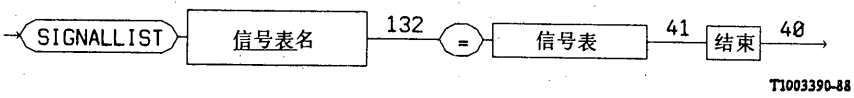
15　信号路由定义（基本 SDL2.5.2）



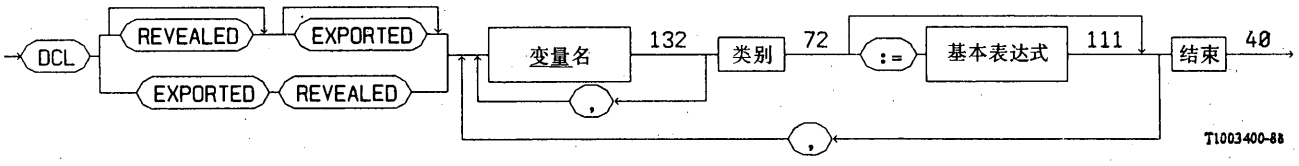
16　信号定义



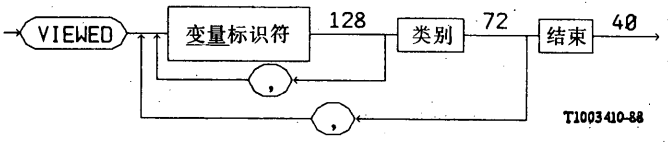
17　信号表定义（基本 SDL2.5.5）



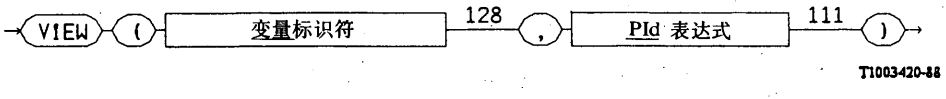
18　变量定义（基本 SDL2.6.1.1）



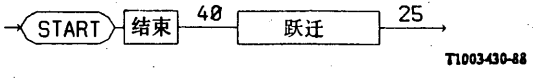
19　视见定义（基本 SDL2.6.1.2）



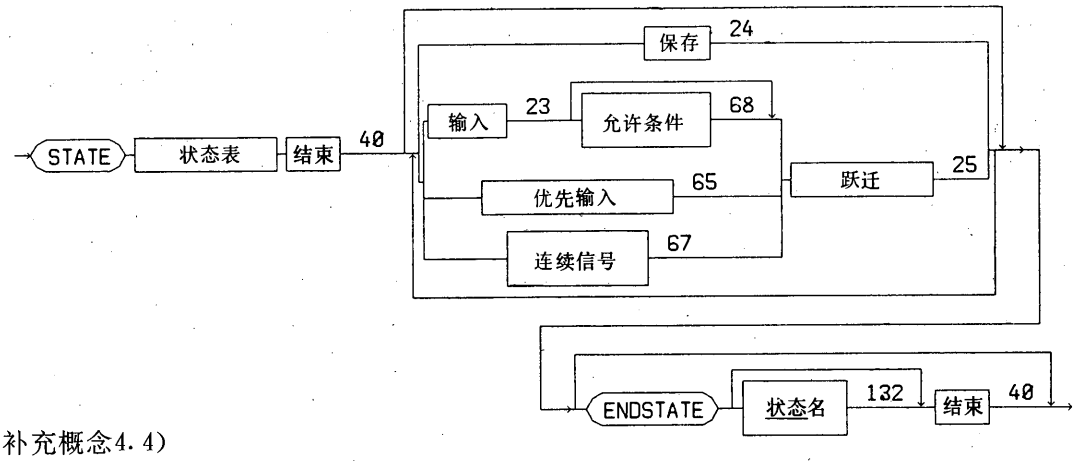
20　视见表达式（数据5.5.4.4）



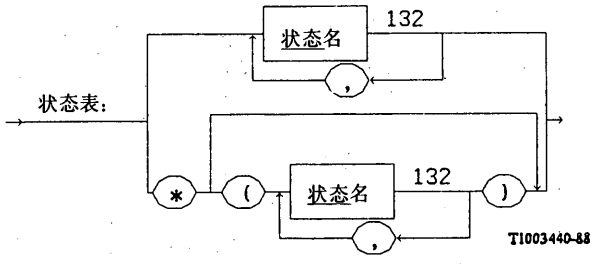
21　启动（基本 SDL2.6.2）



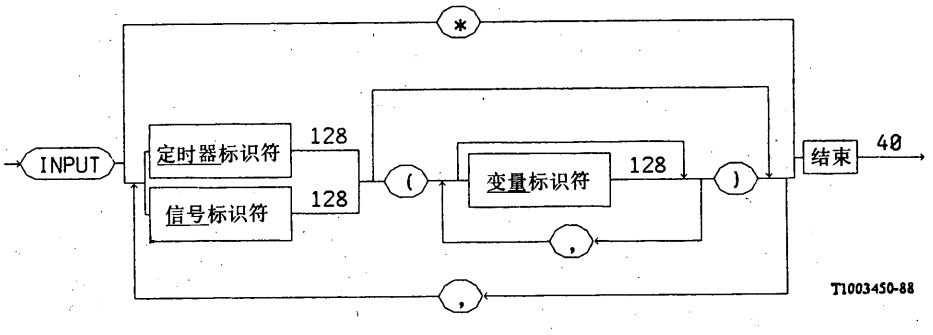
22　状态（基本 SDL2. 6. 3）



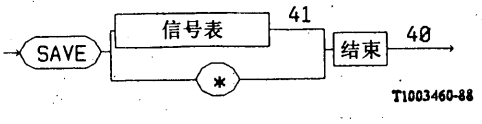
（补充概念4. 4）



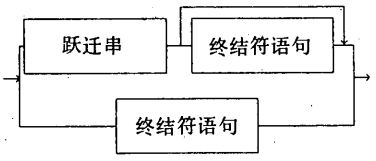
23　输入（基本 SDL2. 6. 4）

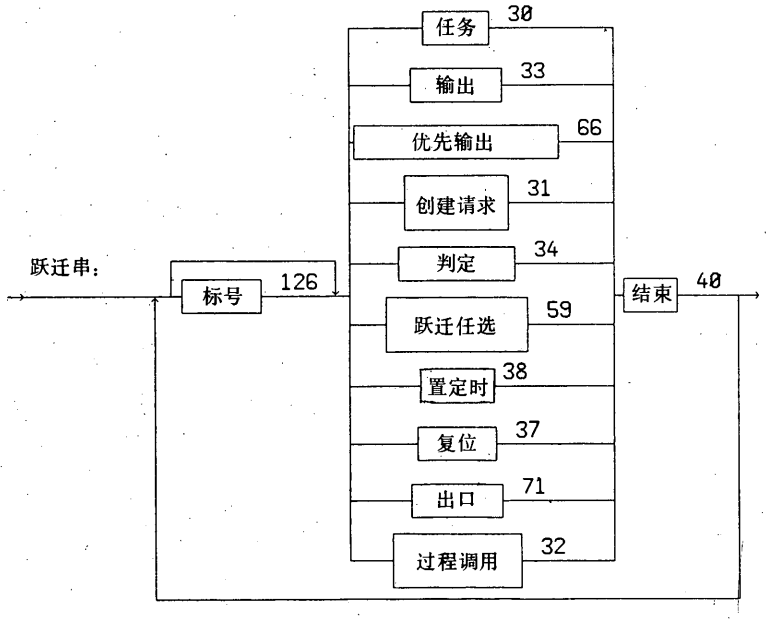


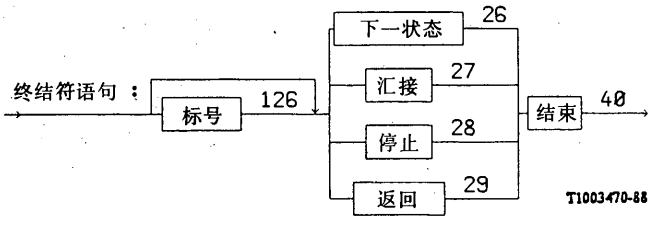
24　保存（基本 SDL2. 6. 5）



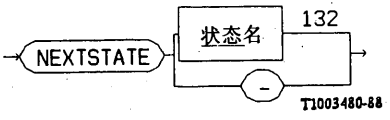
25　跃迁（基本 SDL2.6.7）



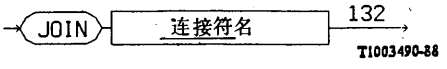




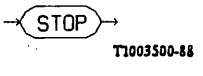
26　下一状态（基本 SDL2.6.7.2.1）



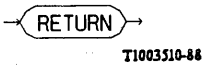
27　汇接（基本 SDL2.6.7.2.2）



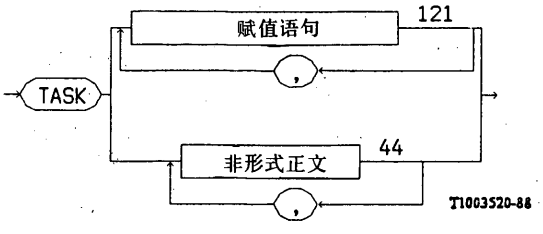
28　停止（基本 SDL2.6.7.2.3）



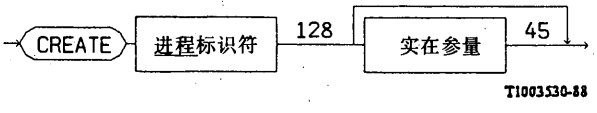
29　返回（基本 SDL2.6.7.2.4）



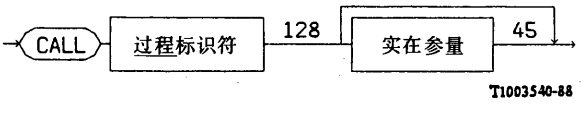
30　任务（基本 SDL2.7.1）



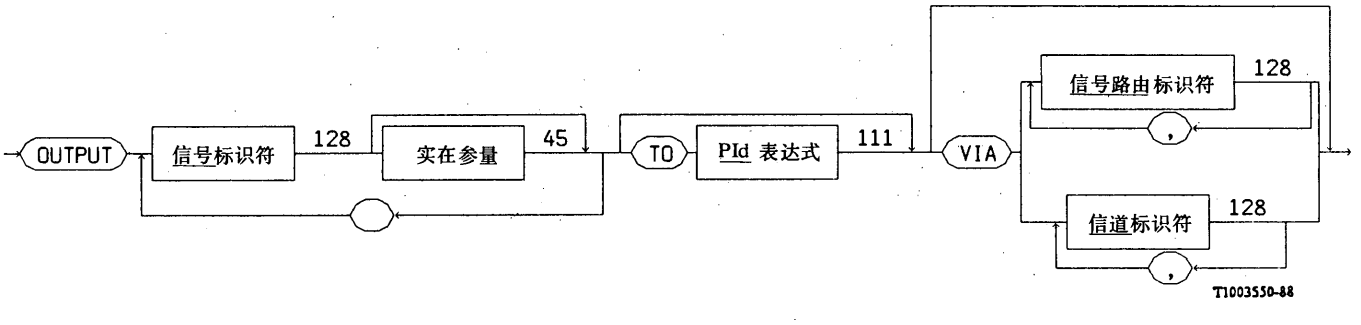
31　创建请求（基本 SDL2.7.2）



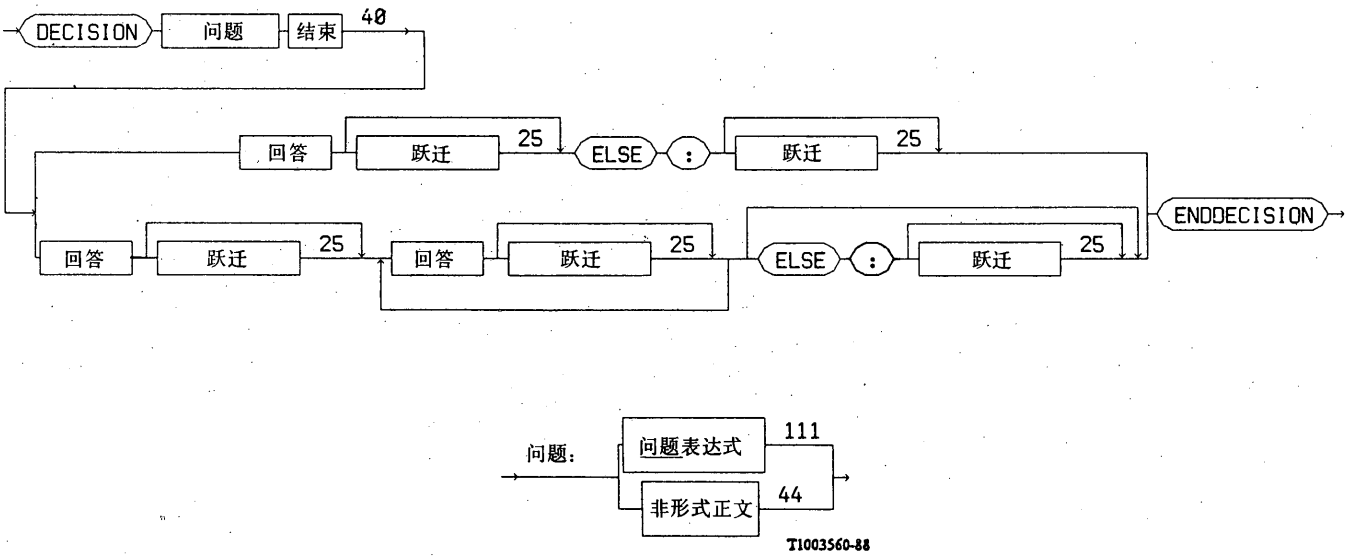
32　过程调用（基本 SDL2.7.3）



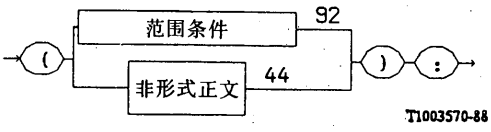
33　输出（基本 SDL2.7.4）



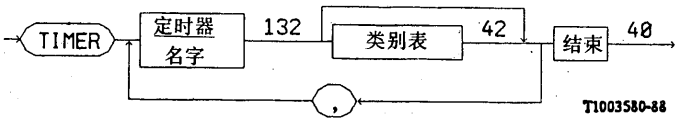
34　判定（基本 SDL2.7.5）



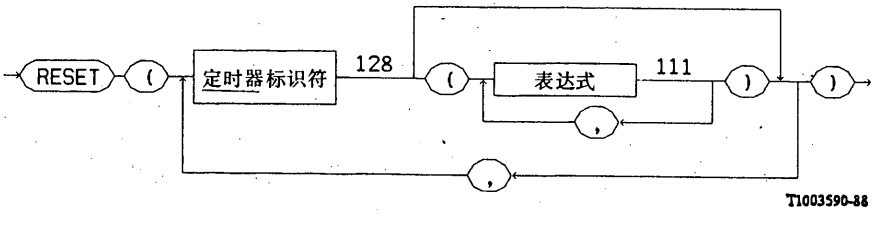
35　回答（基本 SDL2.7.5）



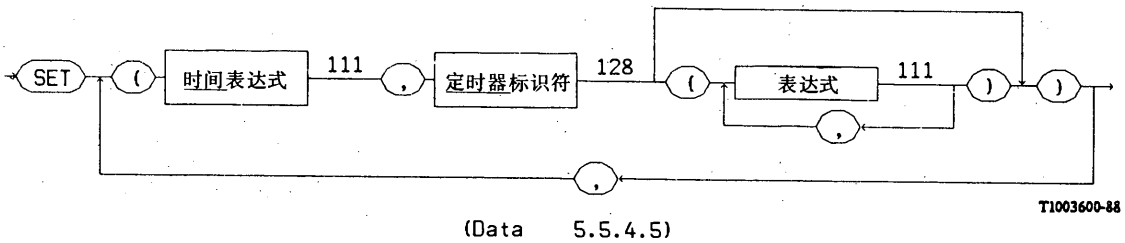
36　定时器定义（基本 SDL2.8）



37　复位（基本 SDL2.8）

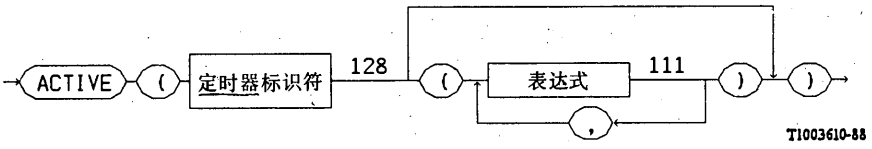


38　置定时（基本 SDL2.8）

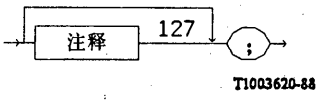


(Data　5.5.4.5)

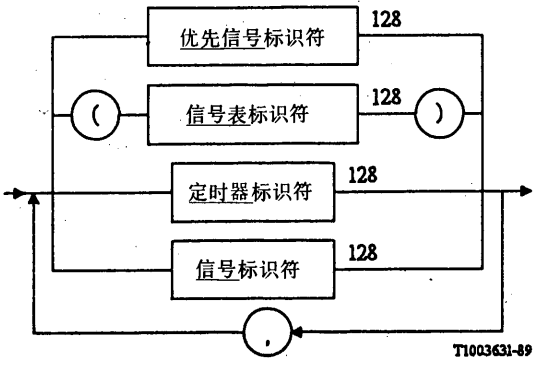
39 定时器活跃表达式（数据5.5.4.5）



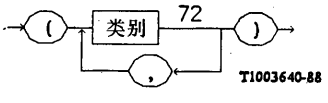
40 结束



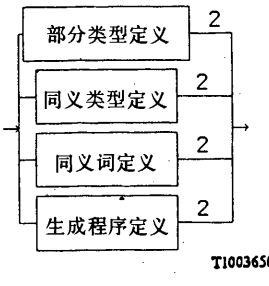
41 信号表



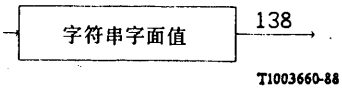
42 类别表



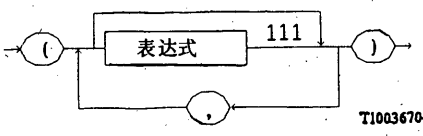
43 数据定义



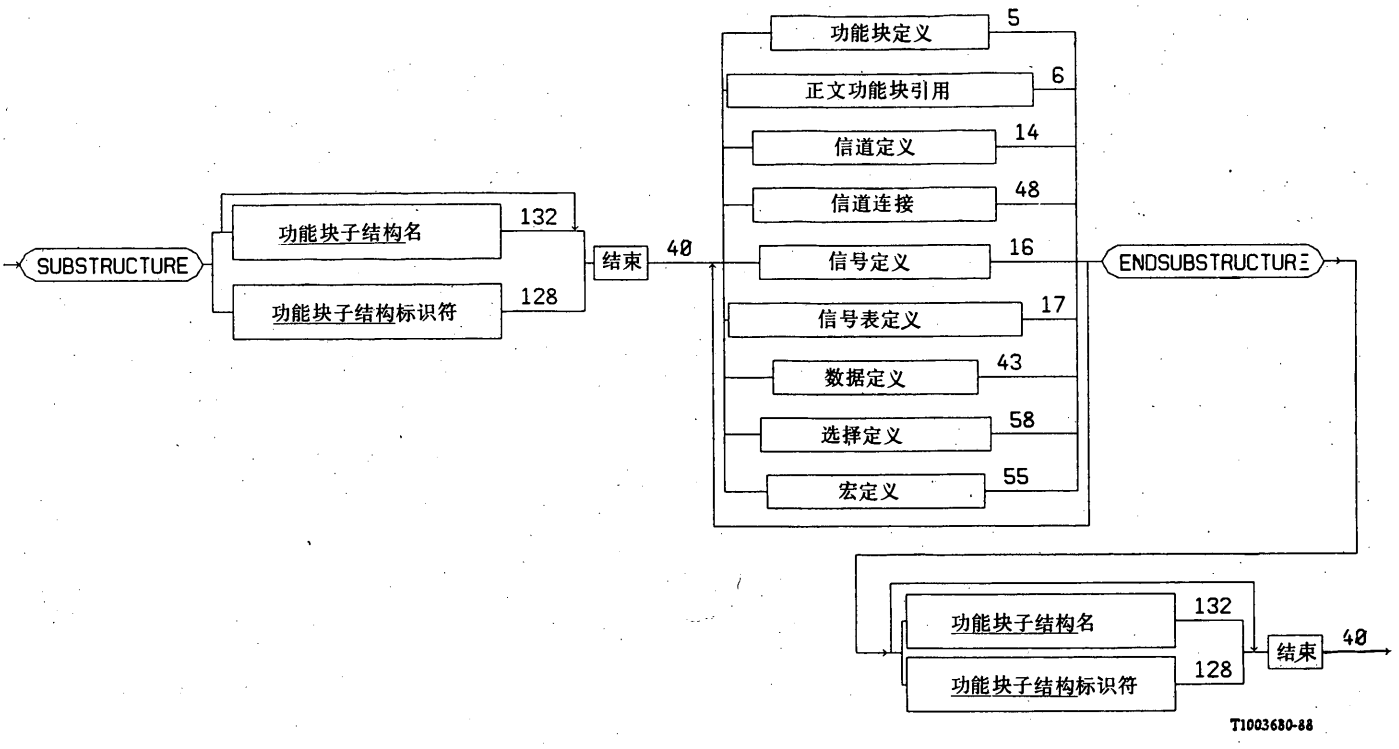
44 非形式正文



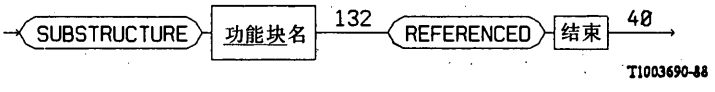
45 实在参量



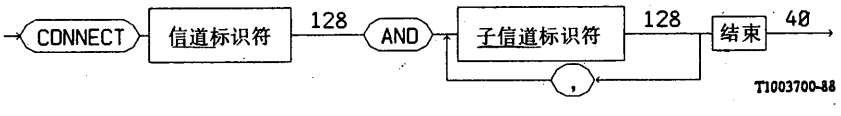
46 功能块子结构定义（结构概念3.2.2）



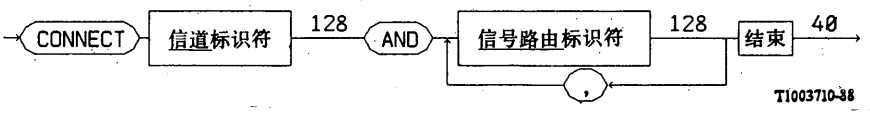
47 功能块子结构引用（结构概念3.2.2）



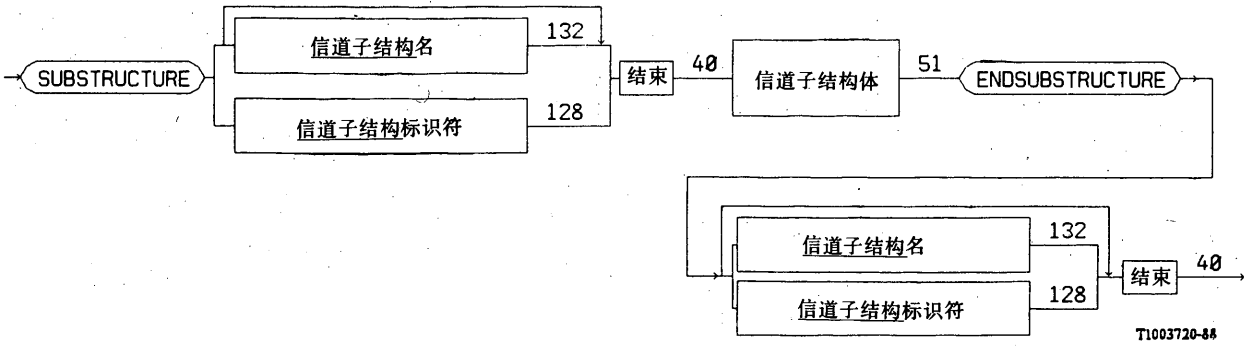
48 信道连接（结构概念3.2.2）



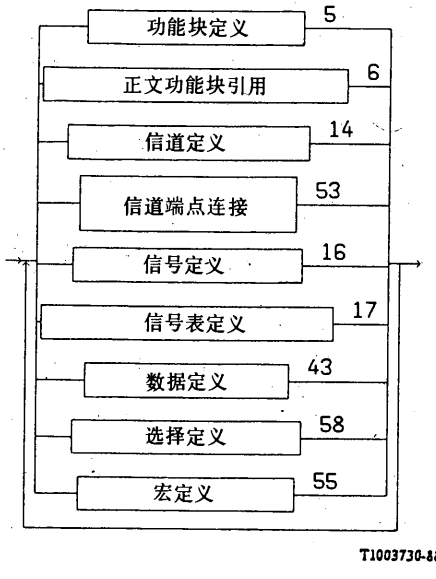
49　信道至路由连接（结构概念3.2.2）



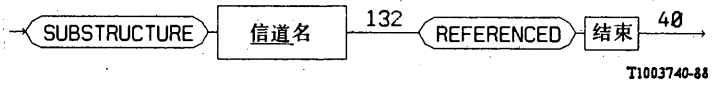
50　信道子结构定义（结构概念3.2.3）



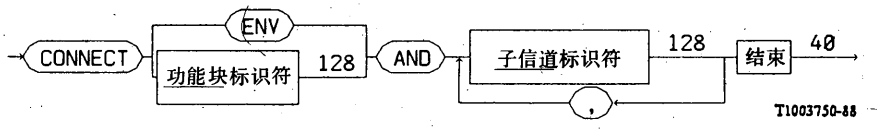
51　信道子结构体（结构概念3.2.3）



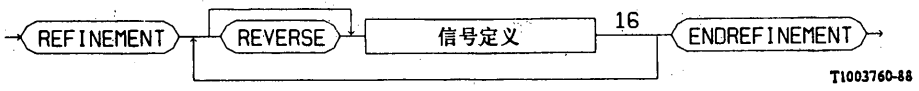
52　信道子结构引用（基本 SDL2.5.1）



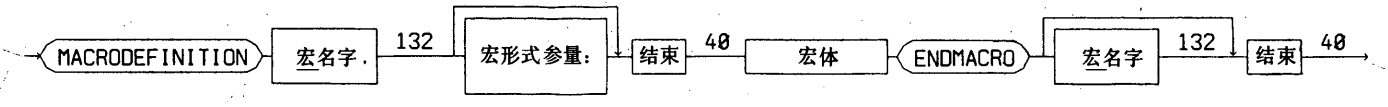
53　信道端点连接（结构概念3.2.3）

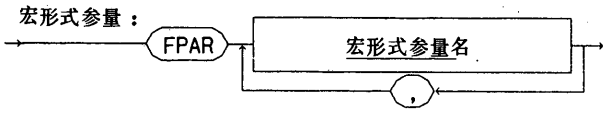


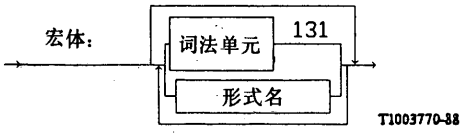
54　信号具体化（结构概念3.3）



55　宏定义（补充概念4.2.2）

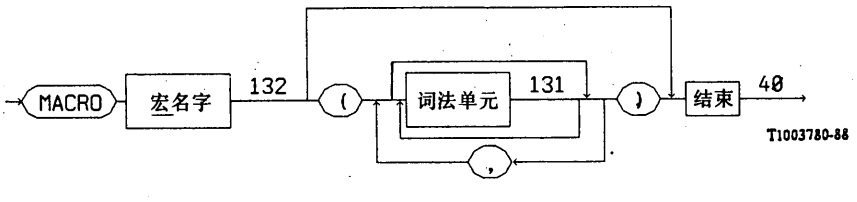




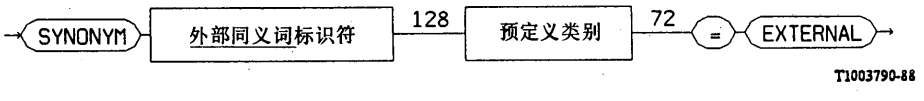


（补充概念4.2.3）

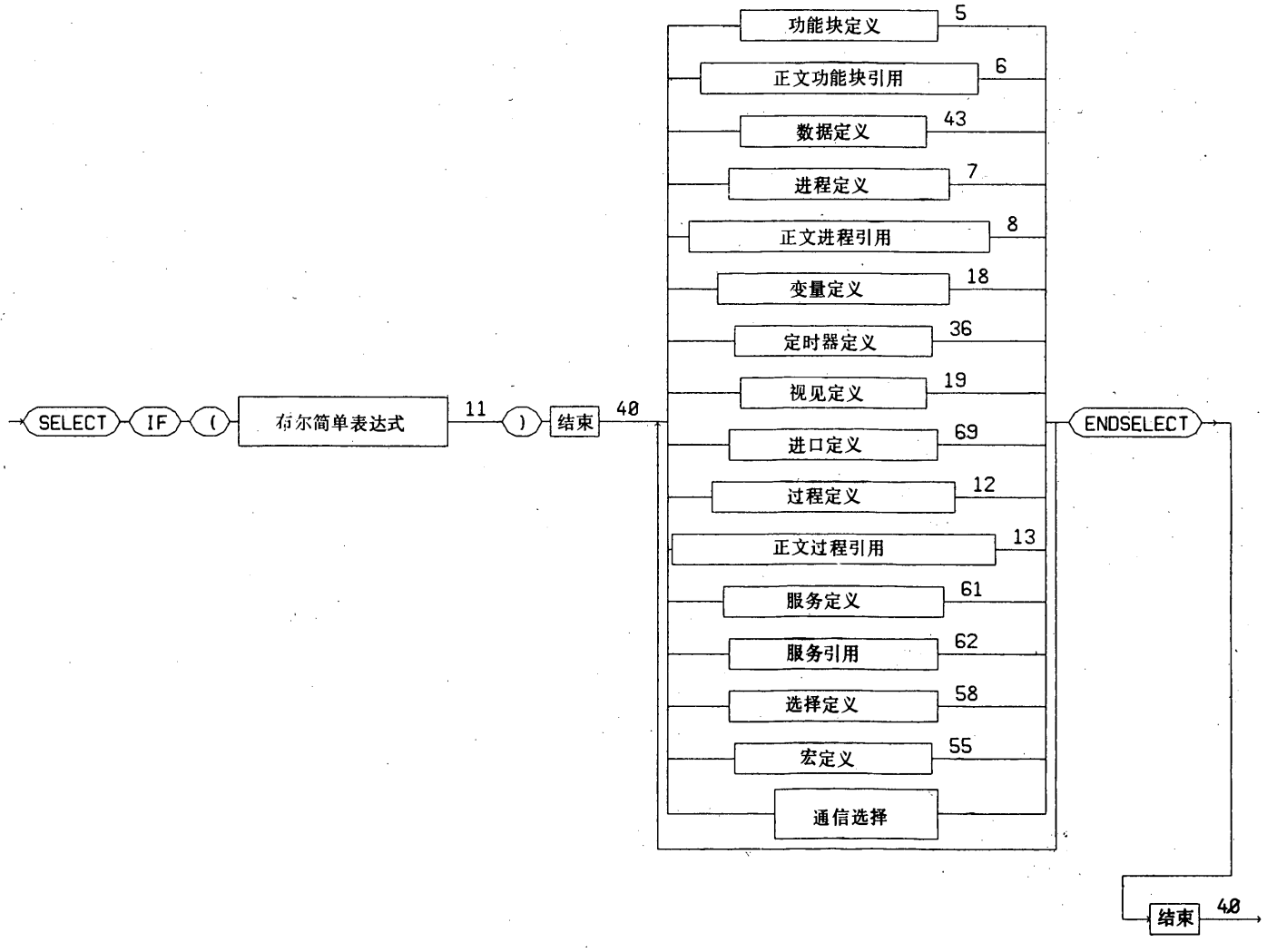
56　宏调用（补充概念4.2.3）

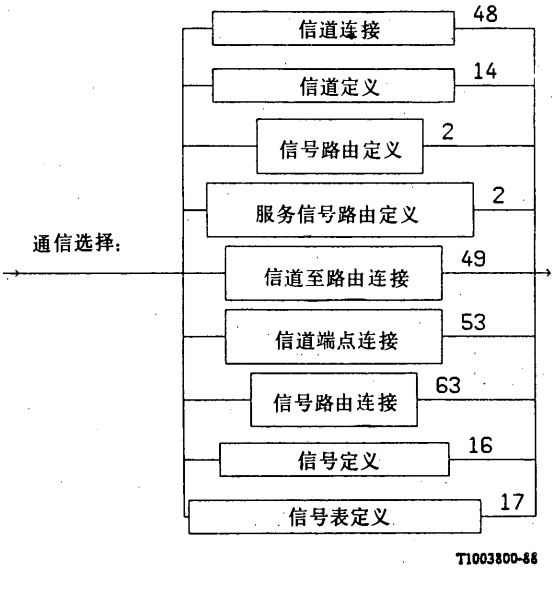


57　外部同义词定义（补充概念4.3.1）

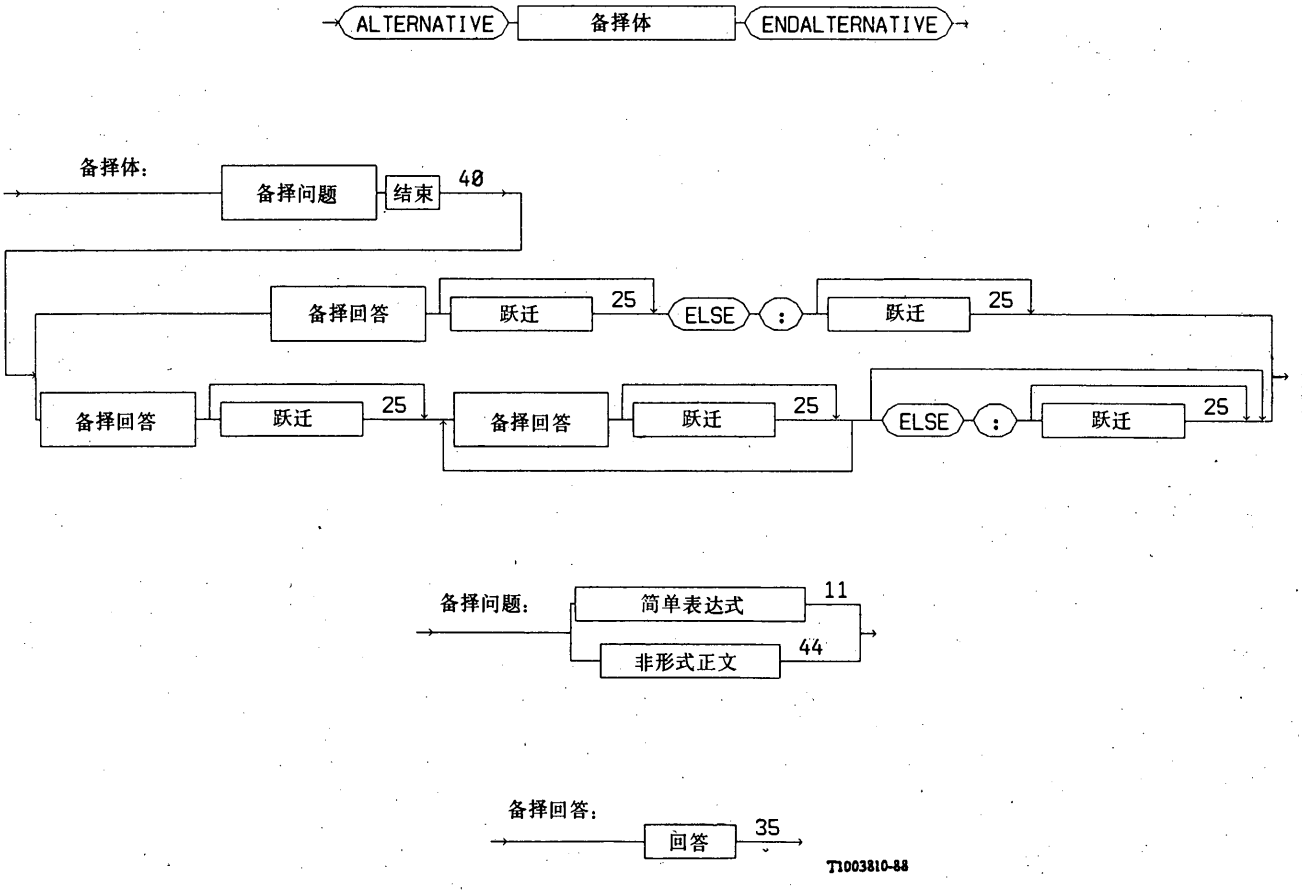


58 选择定义（补充概念4.3.3）

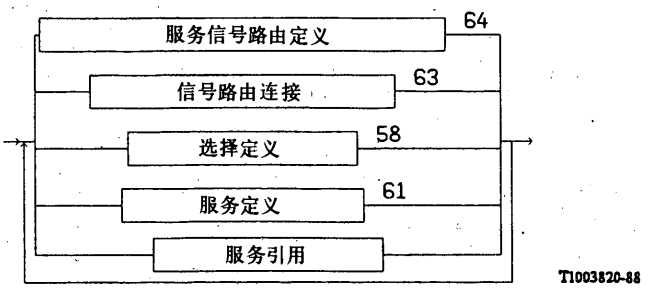




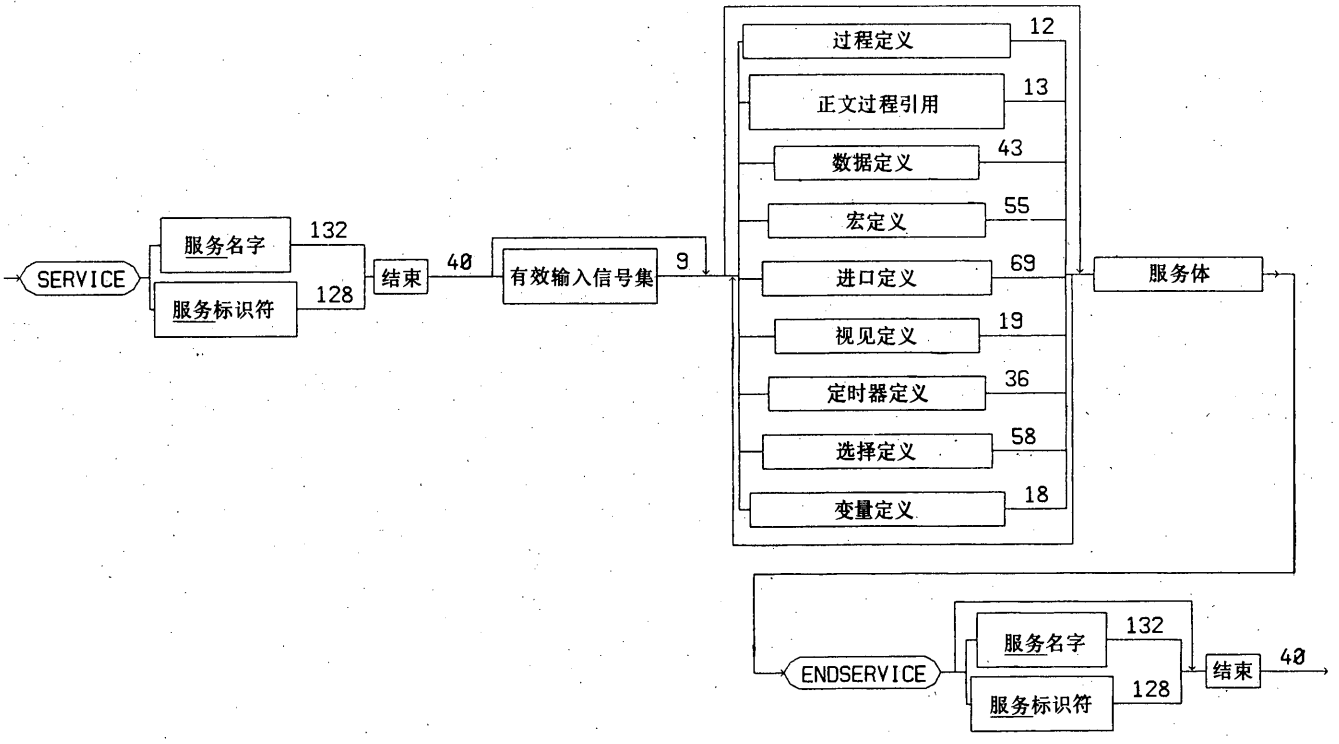
59　跃迁任选（补充概念4.3.4）

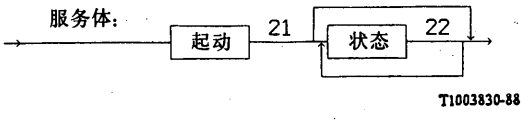


60　服务分解（补充概念4.10.1）



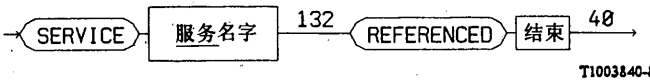
61 服务定义（补充概念4.10.2）





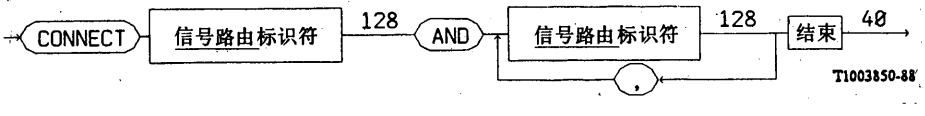
T1003830-88

62 服务引用（补充概念4.10.1）



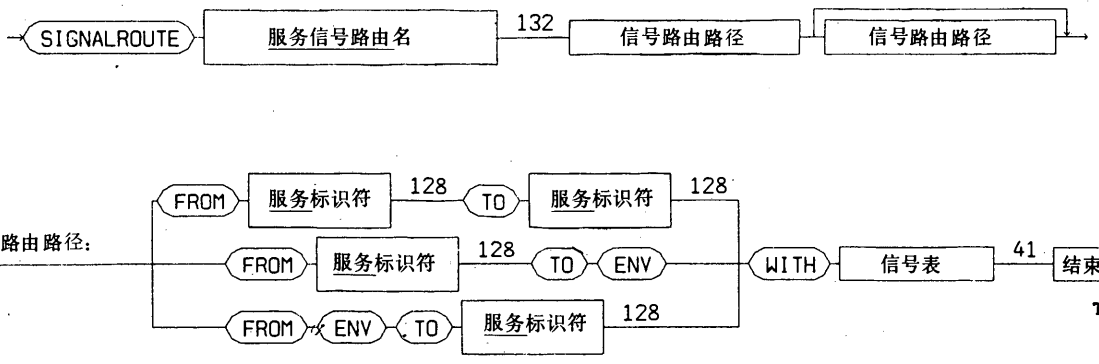
T1003840-88

63 信号路由连接（补充概念4.10.1）

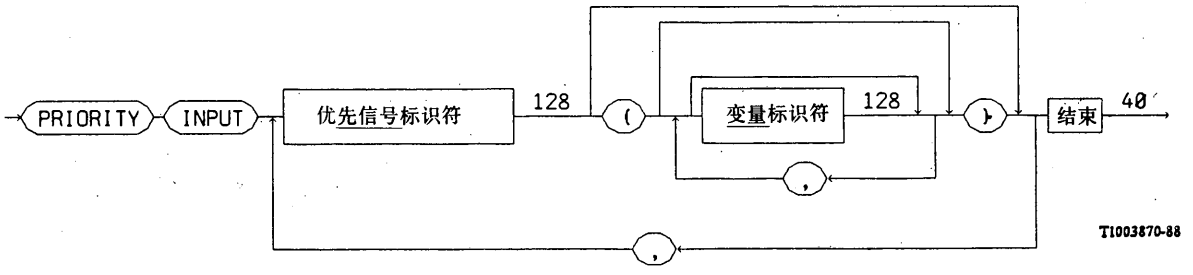


T1003850-88

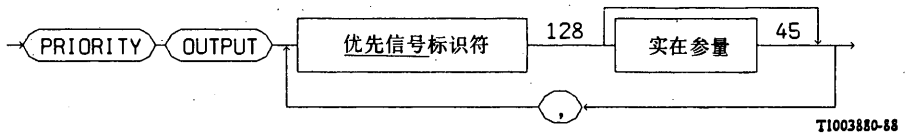
64　服务信号定义（补充概念4.10.1）



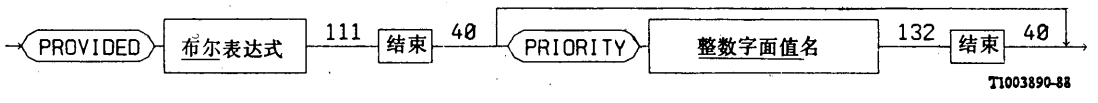
65　优先输入（补充概念4.10.2）



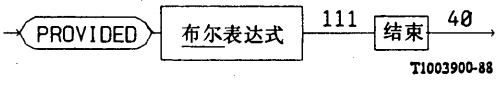
66　优先输出（补充概念4.10.2）



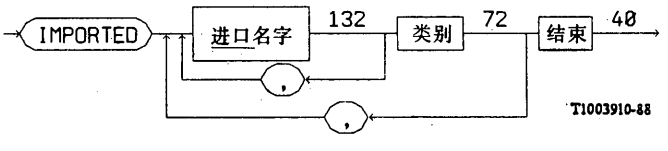
67　连续信号（补充概念4.11）



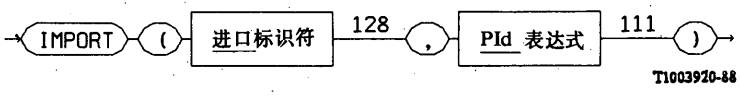
68　允许条件（补充概念4.12）



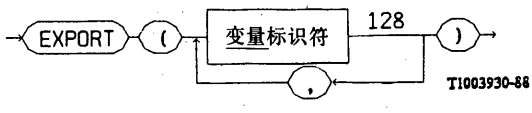
69　进口定义（补充概念4.13）



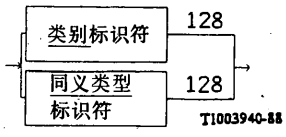
70　进口表达式（补充概念4.13）



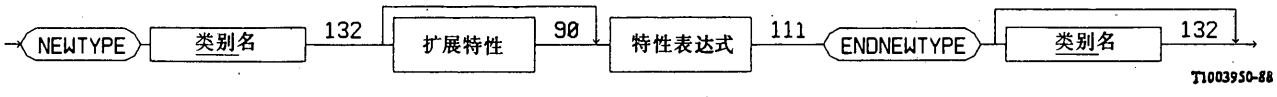
71　出口（补充概念4.13）



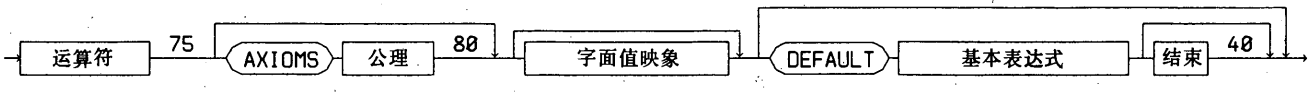
72　类别（数据5.2.2）

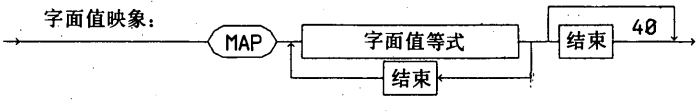


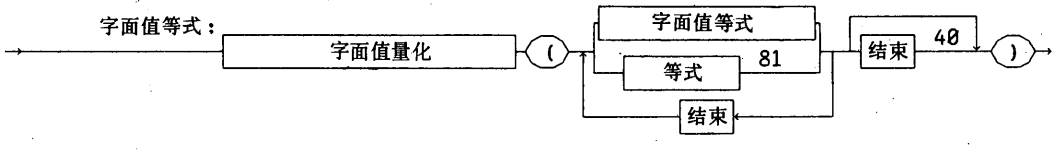
73　部分类型定义（数据5.2.1）

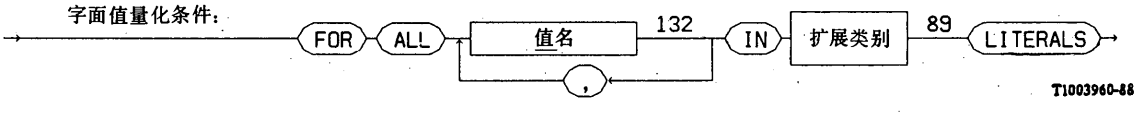


74　特性表达式（数据5.2.1和5.5.3.3）

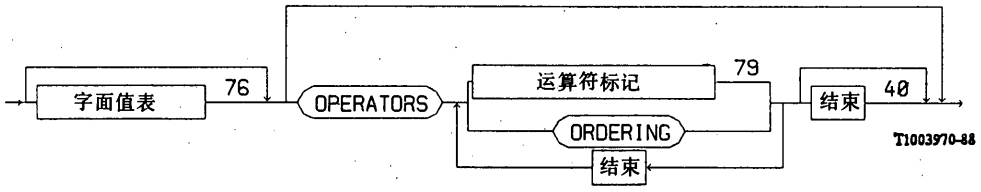




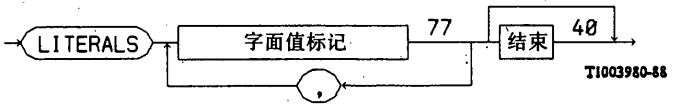




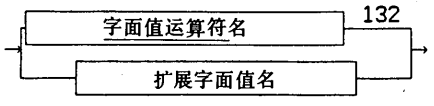
75　运算符（数据5.2.2和5.4.1.8）

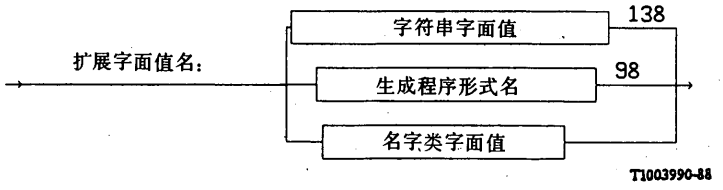


76　字面值表（数据5.2.2）

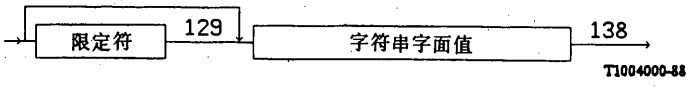


77　字面值标记（数据5.2.2和5.4.1.8）

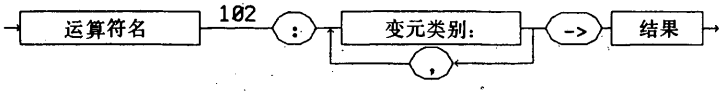


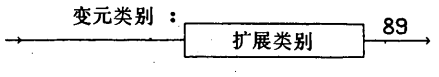


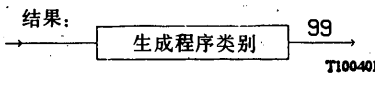
78　字符串字面值标识符



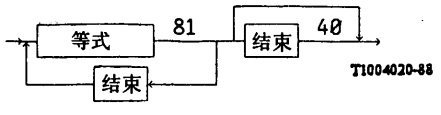
79　运算符标记（数据5.2.2）



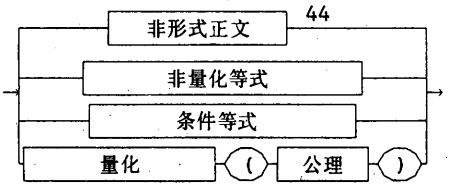


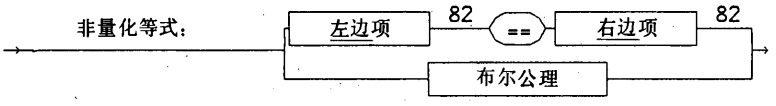


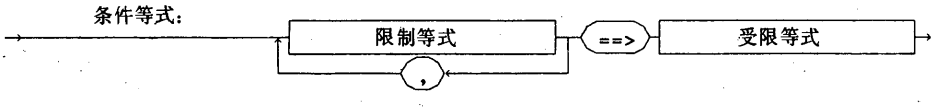
80　公理（数据5.2.3和5.2.4）

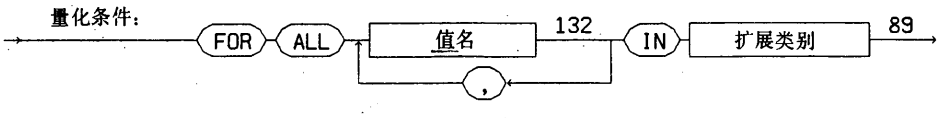


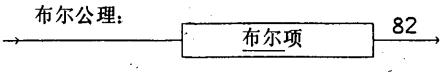
81 等式

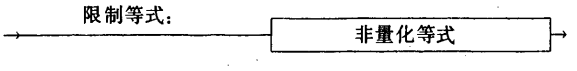


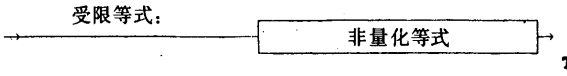






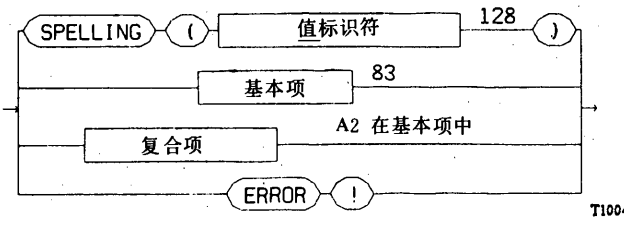






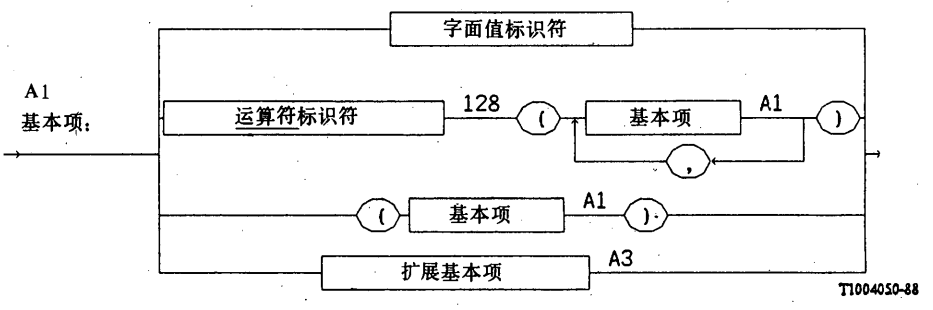
T1004030-88

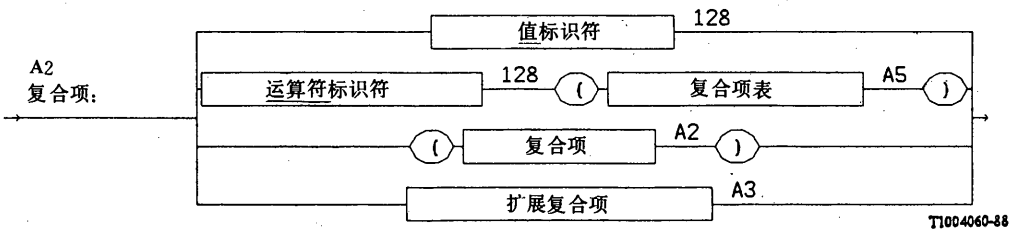
82 项

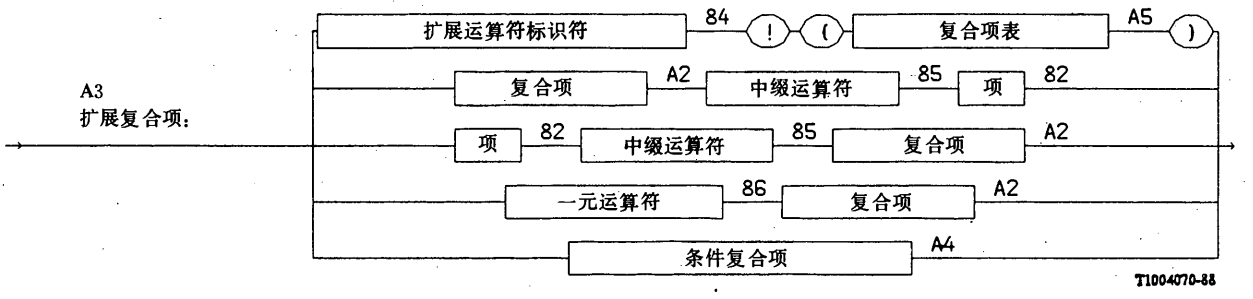


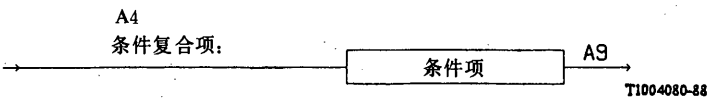
T1004040-88

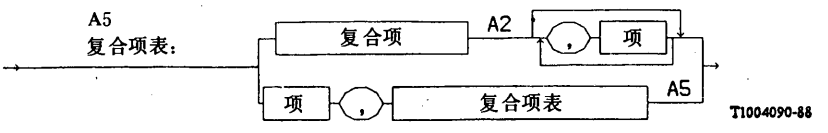
83 基本项

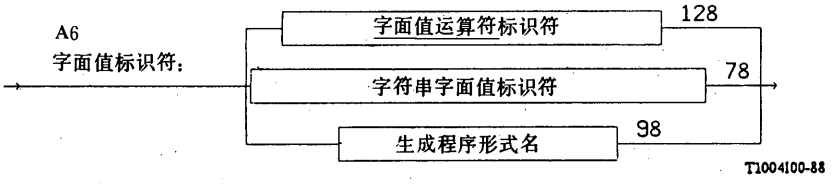

A1
基本项：
字面值标识符
运算符标识符
128
(
基本项
A1
)
,
(
基本项
A1
)
扩展基本项
A3
T1004050-88

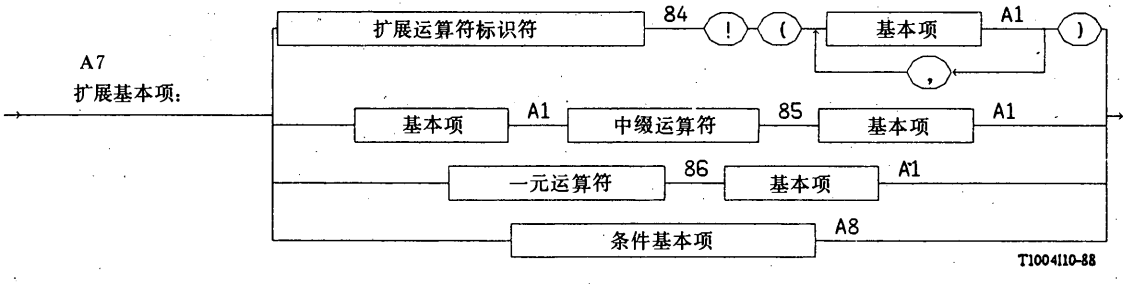

A2
复合项：
值标识符
128
运算符标识符
128
(
复合项表
A5
)
(
复合项
A2
)
扩展复合项
A3
T1004060-88

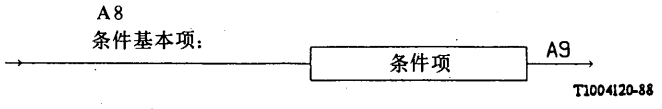

A3
扩展复合项：
扩展运算符标识符
84
!
(
复合项表
A5
)
复合项
A2
中缀运算符
85
项
82
项
82
中缀运算符
85
复合项
A2
一元运算符
86
复合项
A2
条件复合项
A4
T1004070-88

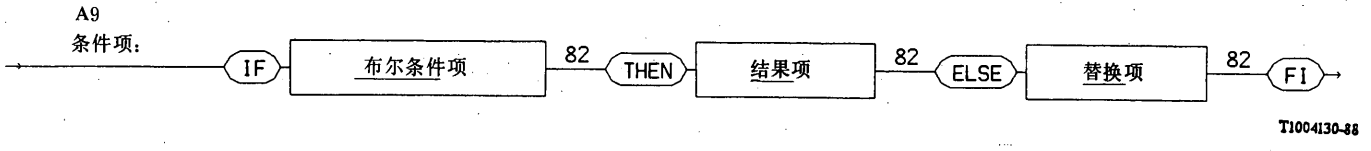

A4
条件复合项：
条件项
A9
T1004080-88


A5
复合项表：
复合项
A2
,
项
项
,
复合项表
A5
T1004090-88

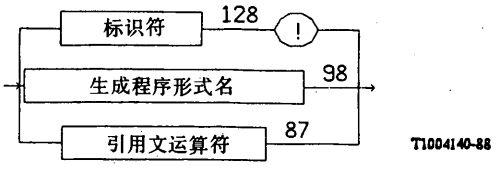

A6
字面值标识符：
字面值运算符标识符
128
字符串字面值标识符
78
生成程序形式名
98
T1004100-88


A7
扩展基本项：
扩展运算符标识符
84
!
(
基本项
A1
)
,
基本项
A1
中缀运算符
85
基本项
A1
一元运算符
86
基本项
A1
条件基本项
A8
T1004110-88

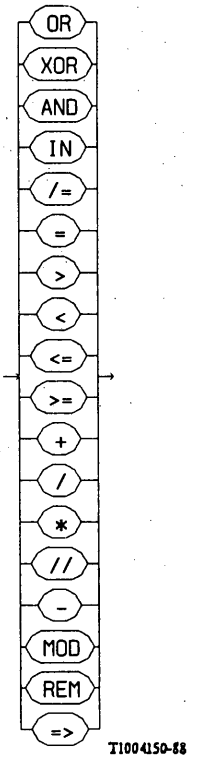

A8
条件基本项：
条件项
A9
T1004120-88


A9
条件项：
IF
布尔条件项
82
THEN
结果项
82
ELSE
替换项
82
FI
T1004130-88

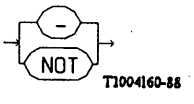
84　扩展运算符标识符


标识符
128
!
生成程序形式名
98
引用文运算符
87
T1004140-88

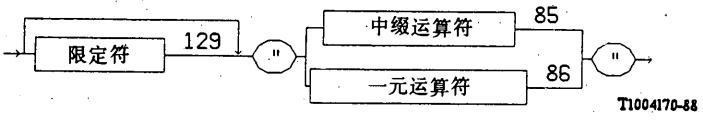
85 中缀运算符



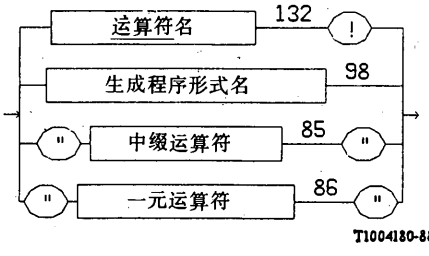
86 一元运算符



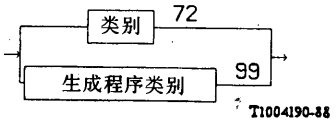
87　引用文运算符



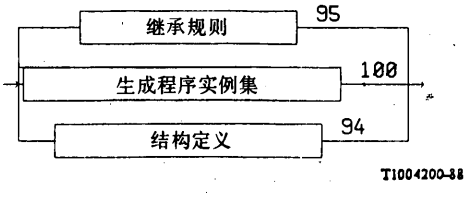
88　扩展运算符名（数据5.4.1）



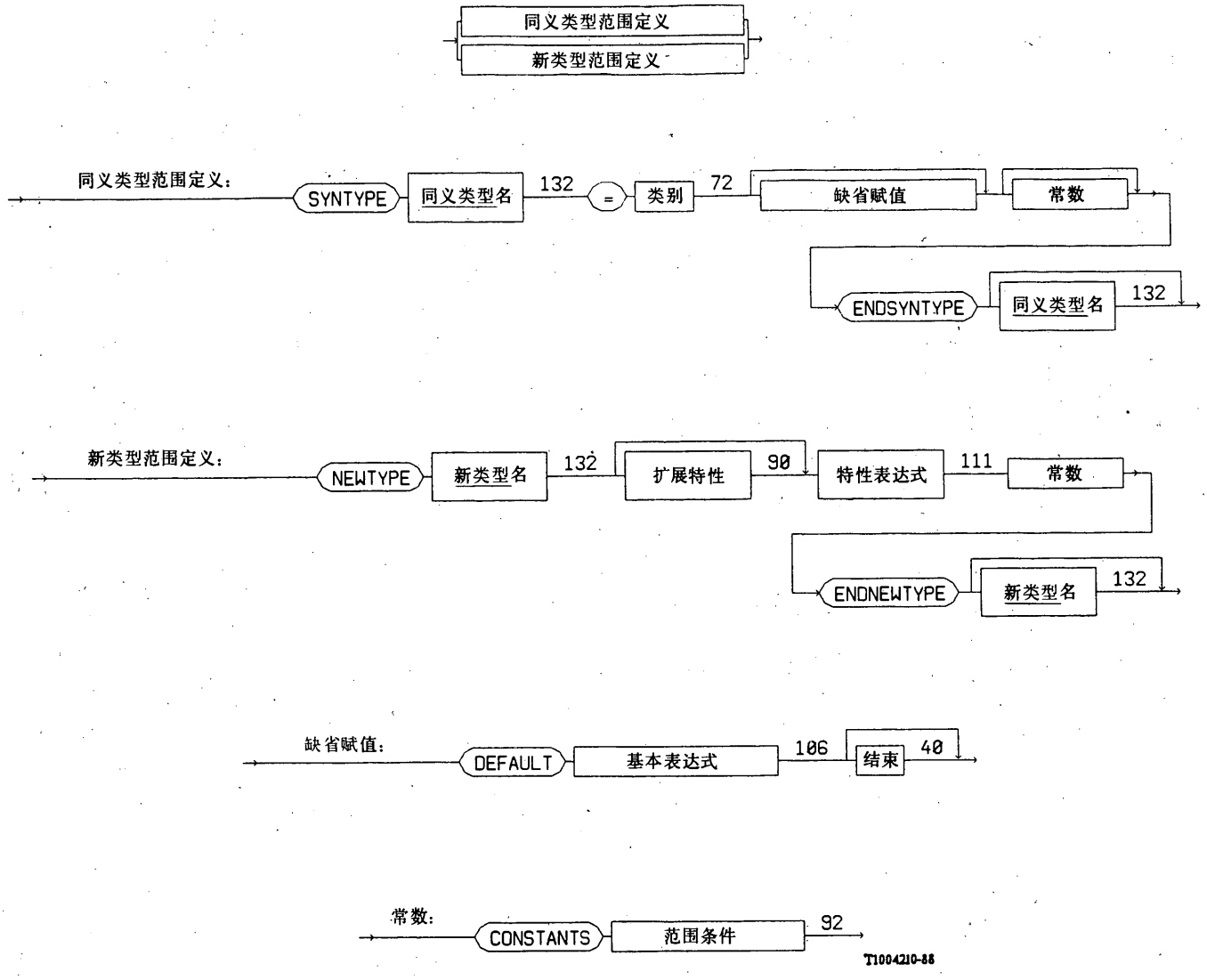
89　扩展类别（数据5.4.1.9）



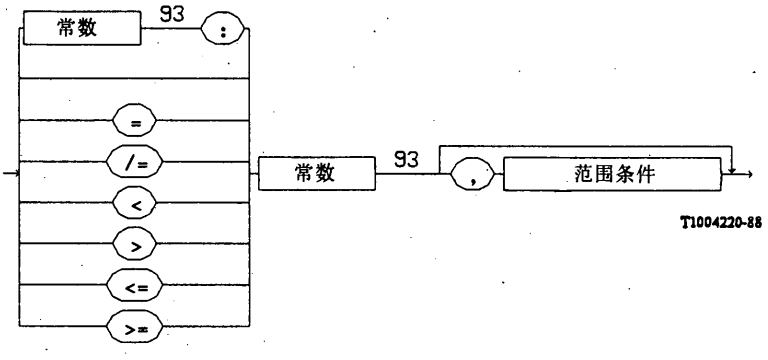
90　扩展特性（数据5.4.1）



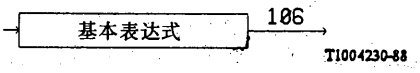
91　同义类型定义（数据5.4.1.9）



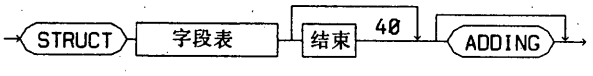
92　范围条件（数据5.4.1.9.1）

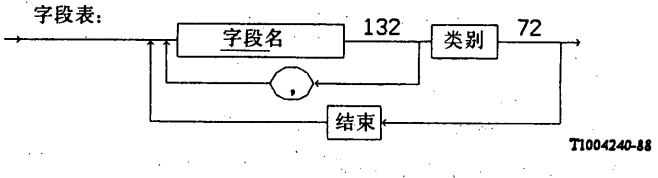


93　常数（数据5.4.1.9）

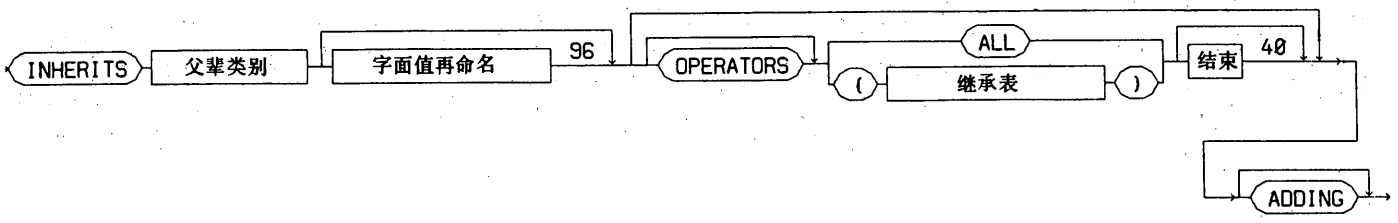


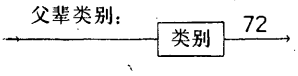
94　结构定义（数据5.4.1.10）

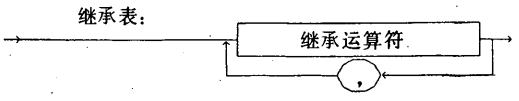


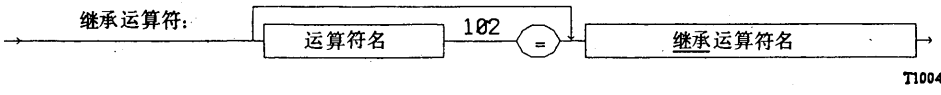


95　继承规则（数据5.4.1.11）



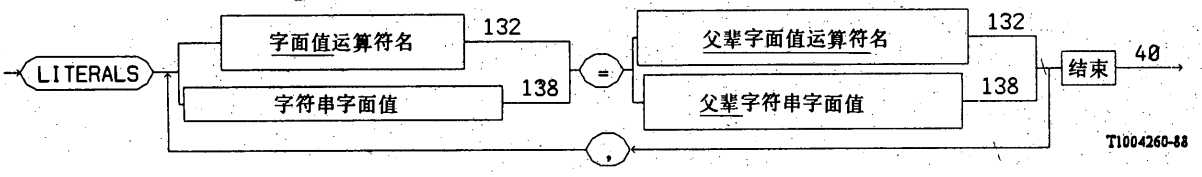




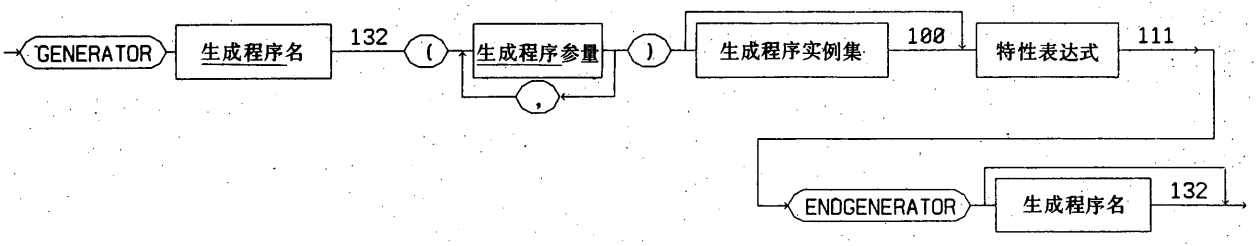


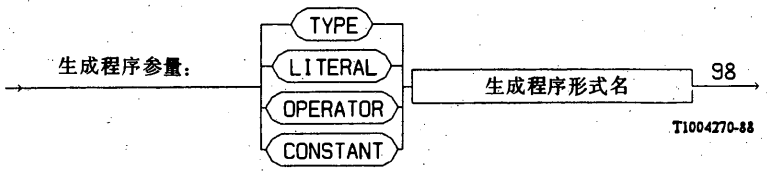
T1004250-88

96　字面值再命名（数据5.4.1.11）

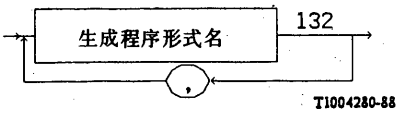


97　生成程序定义（数据5.4.1.12.1）

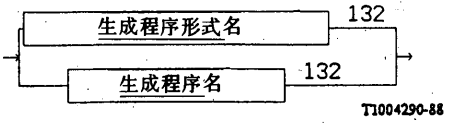




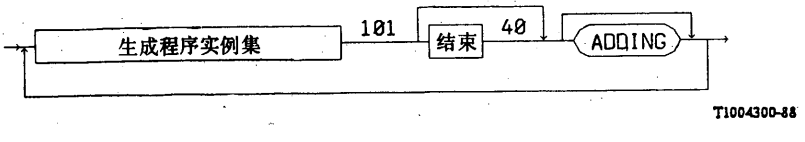
98　生成程序形式名（数据5.4.1.12.1）



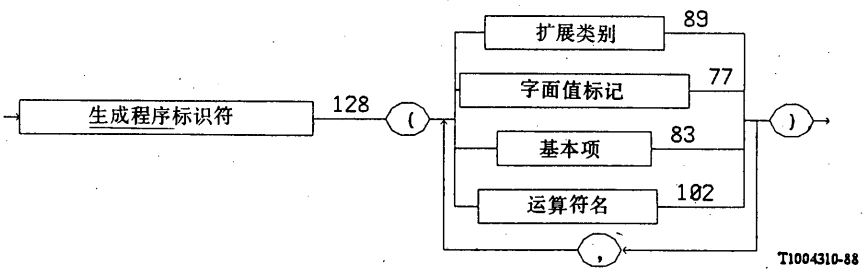
99　生成程序类别（数据5.4.1.12.1）



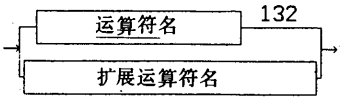
100　生成程序实例集（数据5.4.1.12.2）

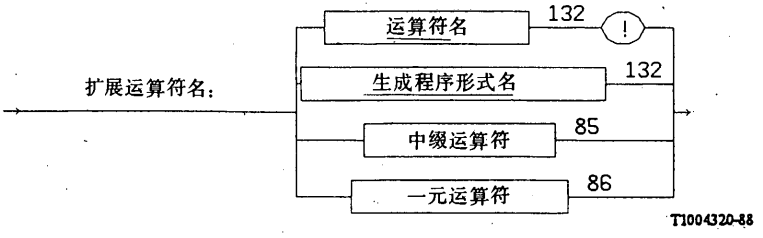


101 生成程序实例（数据5.4.1.12.2）

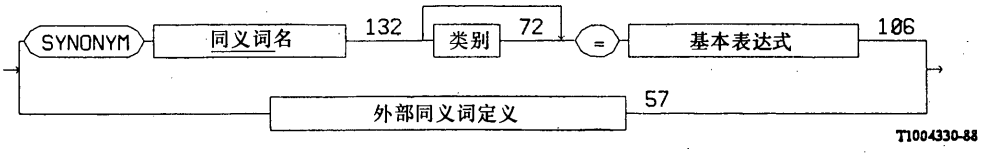


102 运算符名

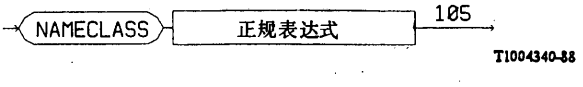




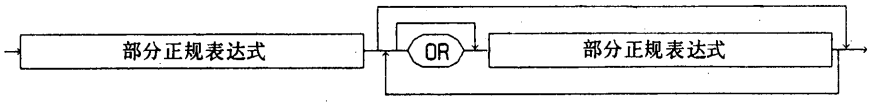
103 同义词定义（数据5.4.1.13）

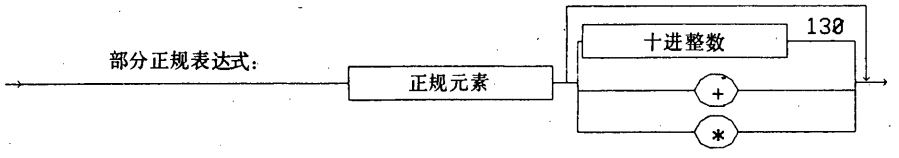


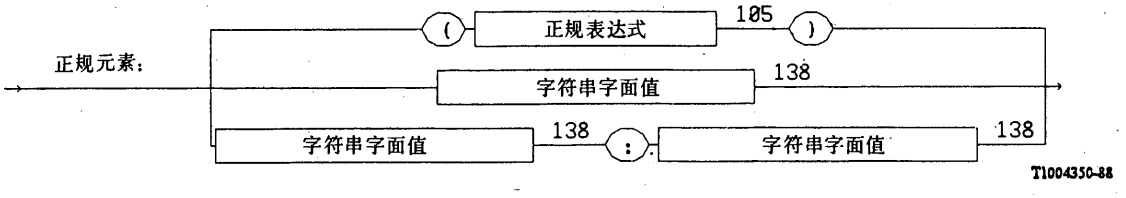
104 名字类字面值（数据5.4.1.14）



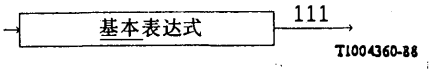
105 正规表达式（数据5.4.1.14）



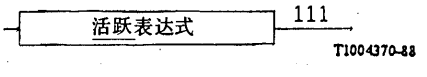




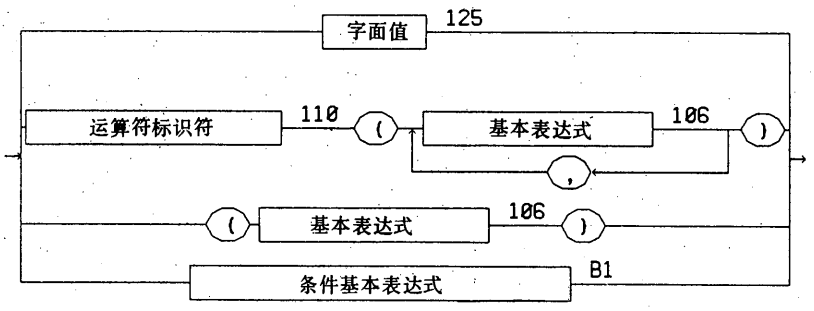
106 基本表达式（数据5.4.2.1）

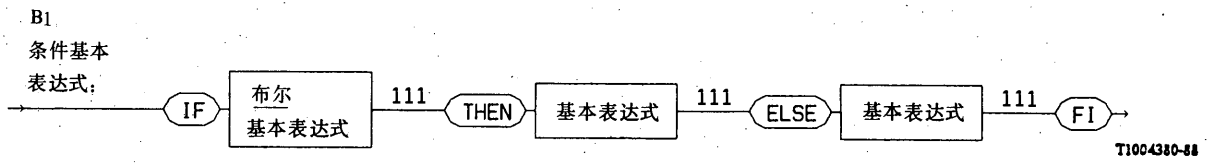


107 活跃表达式（数据5.4.2.2）

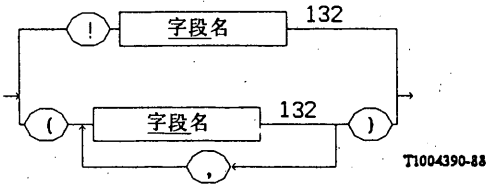


108 基本初始数（数据5.4.2.2）

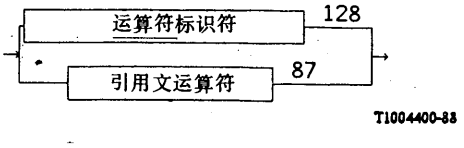




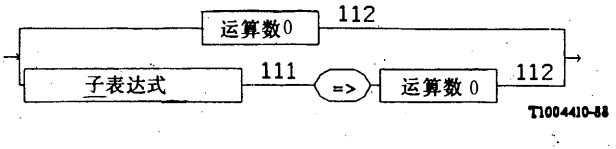
109 字段选择



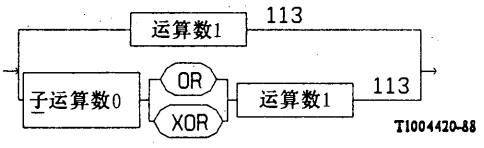
110 运算符标识符



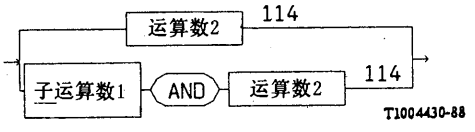
111 表达式（数据5.5.2.1）



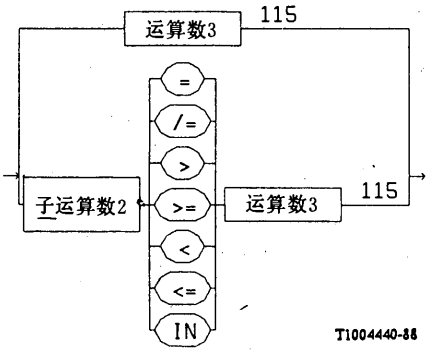
112　运算数0（数据5.5.2.1）



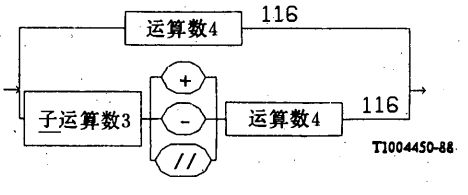
113　运算数1（数据5.5.2.1）



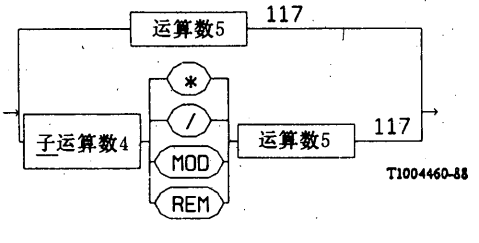
114　运算数2（数据5.5.2.1）



115　运算数3（数据5.5.2.1）

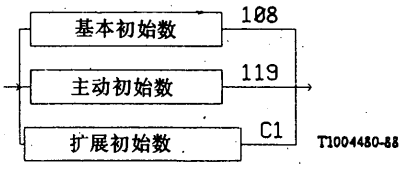


116　运算数4（数据5.5.2.1）

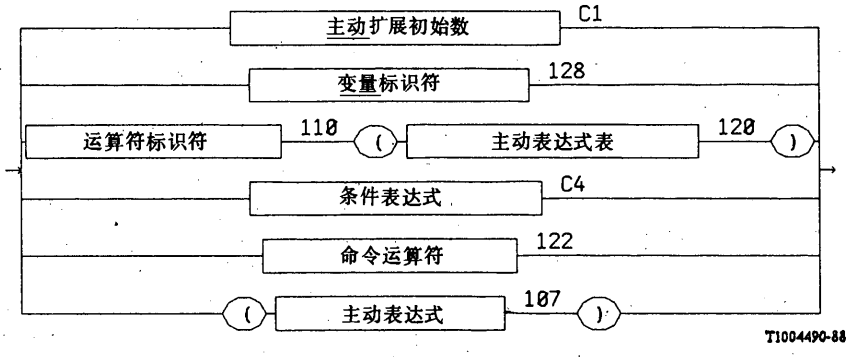


117　运算数5（数据5.5.2.1）

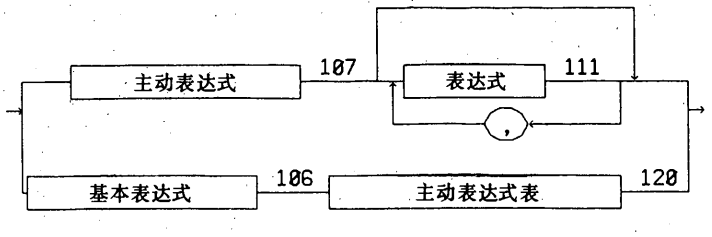
118　初始数

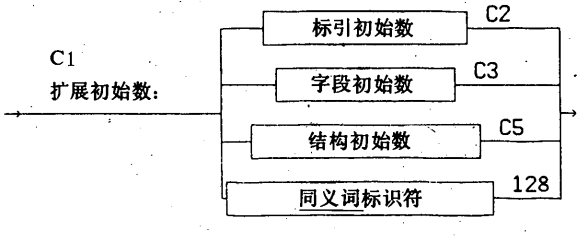


119　主动初始数

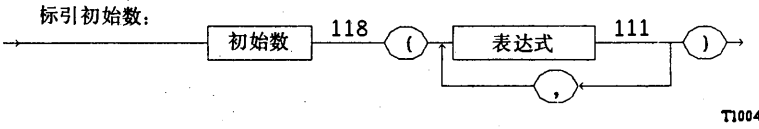


120　主动表达式表（数据5.5.2.1）



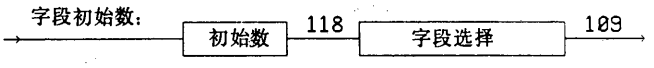


C2
标引初始数：

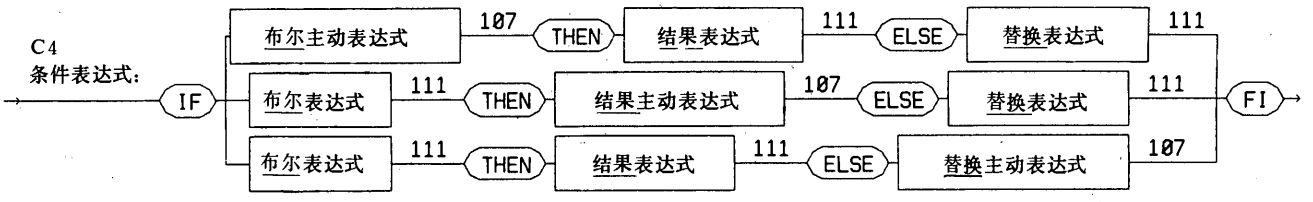


T1004510-88

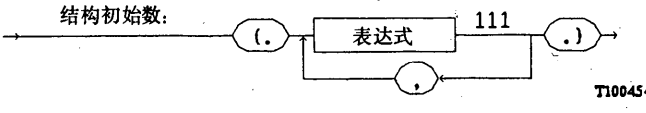
C3
字段初始数：



C4
条件表达式：

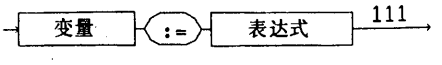


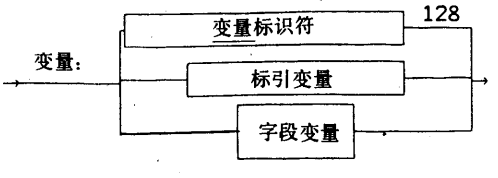
C5
结构初始数：

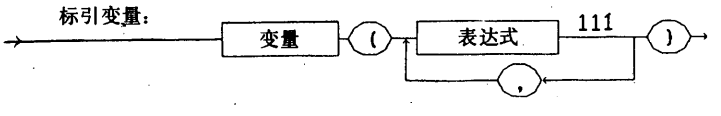


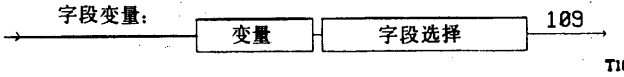
T1004540-88

121 赋值语句



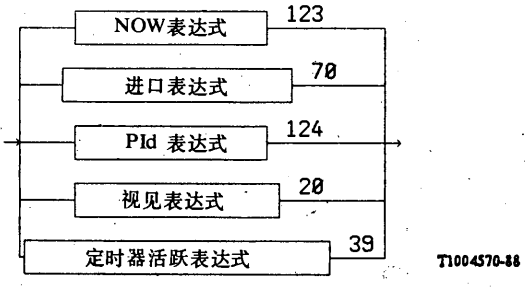




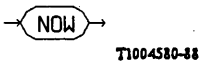


T1004550-88

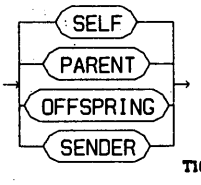
122 命令运算符（数据5.5.4）



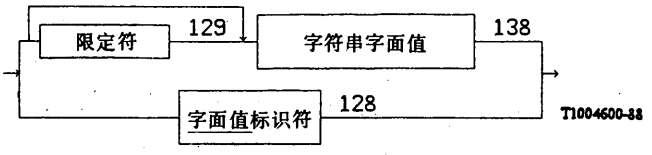
123 NOW 表达式



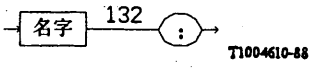
124 PId 表达式



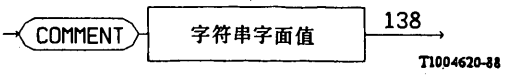
125 字面值



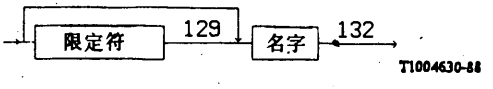
126 标号



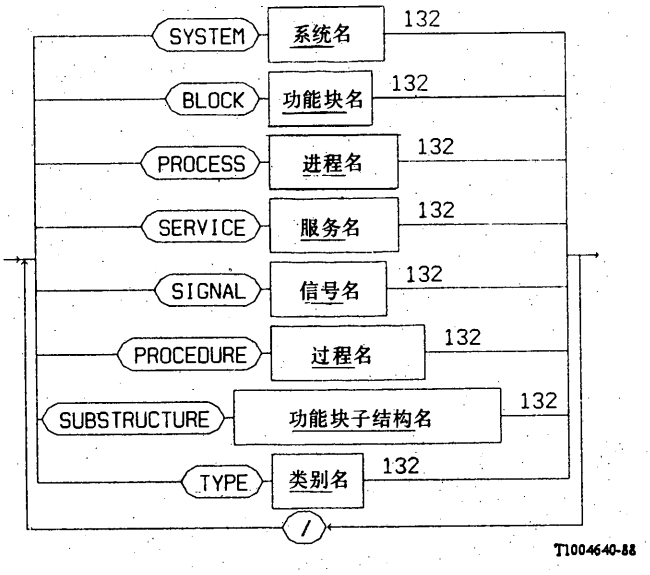
127 注释



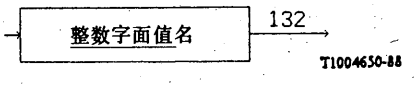
128 标识符



129 限定符

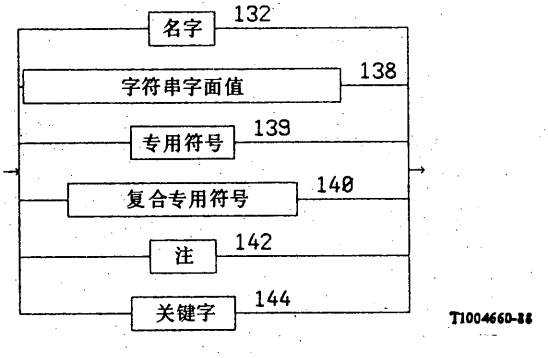


130 十进整数

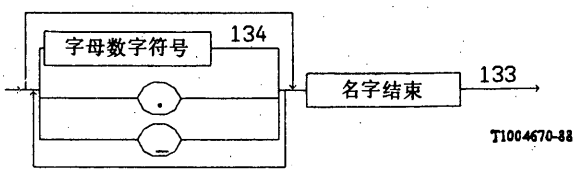


词法规则语法图

131 词法单元

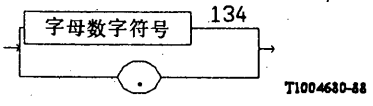


132 名字

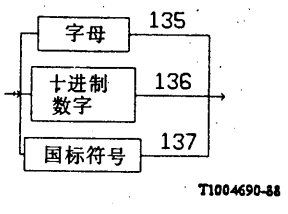


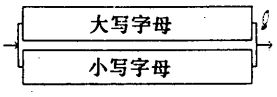
名字最后必须含有一个字母字符

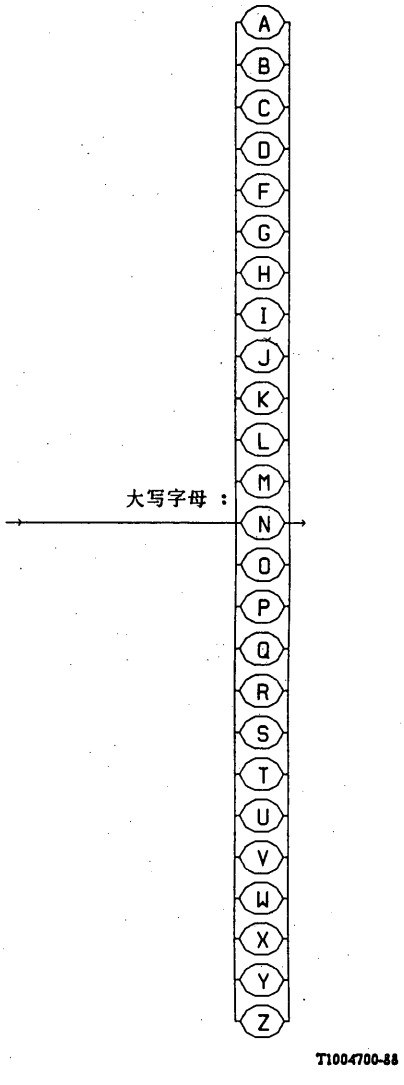
133 名字结束

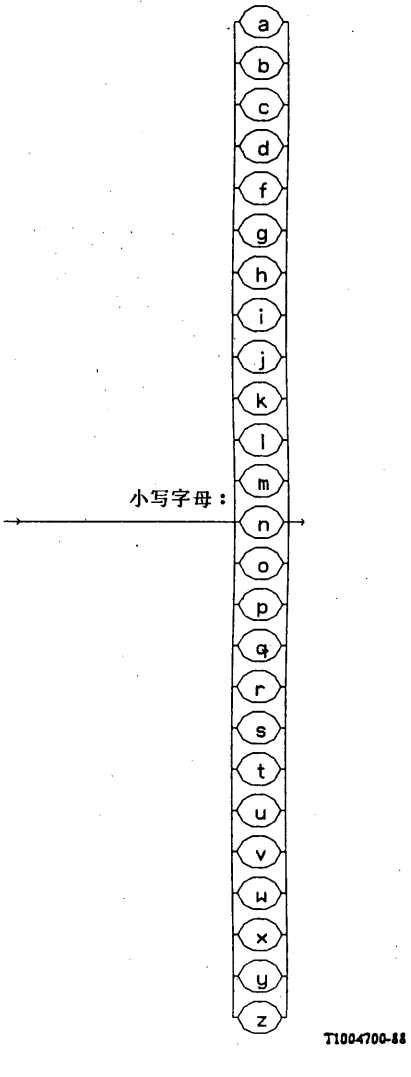


134 字母数字字符

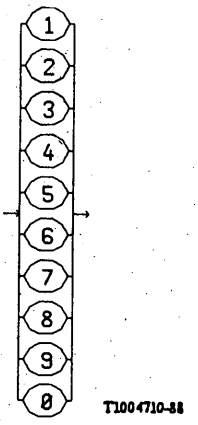



大写字母
小写字母

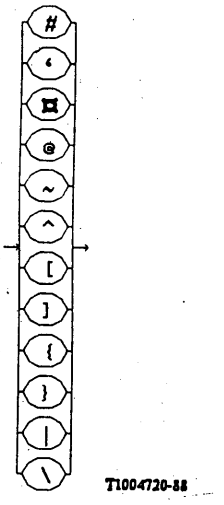

大写字母 :
A
B
C
D
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
T1004700-88


小写字母 :
a
b
c
d
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
T1004700-88

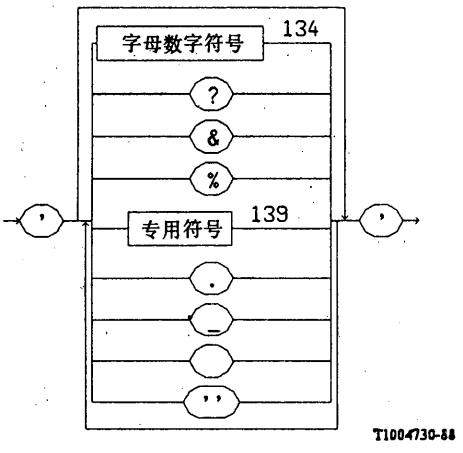
136　十进制数字



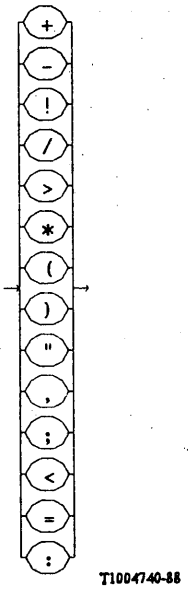
137　国标符号



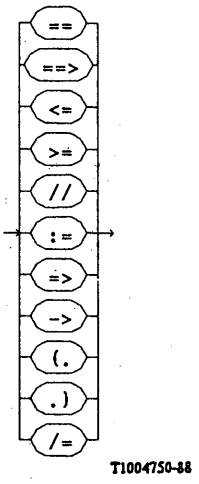
138　字符串字面值



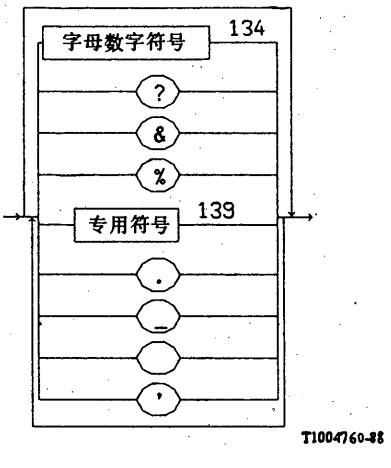
139　专用符号



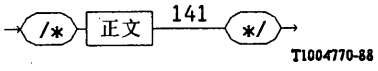
140 复合专用符号



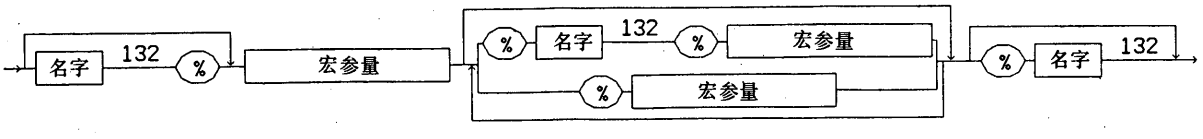
141 正文

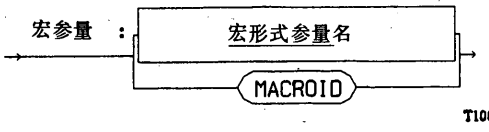


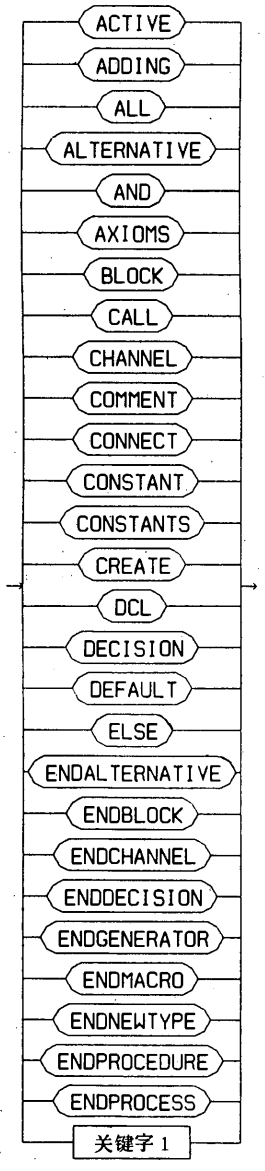
142 注


/*
正文
141
*/
T1004770-88

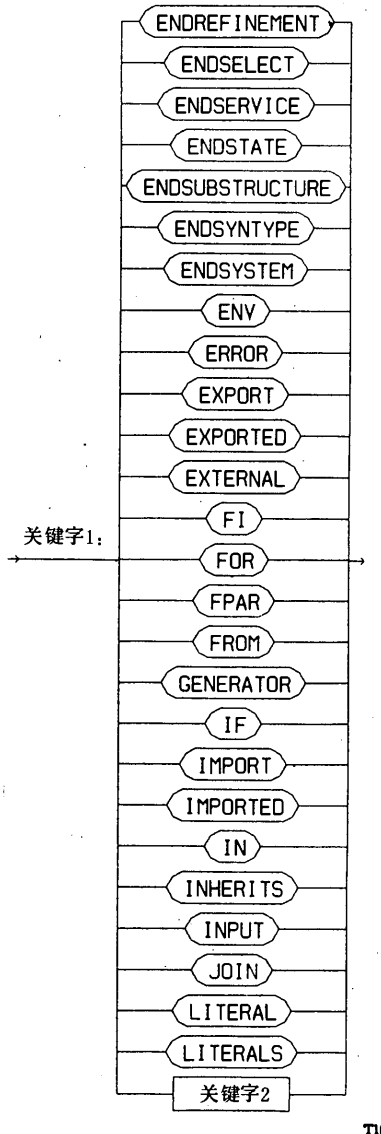
143 形式名


名字
132
%
宏参量
%
名字
132
%
宏参量
%
宏参量
%
名字
132

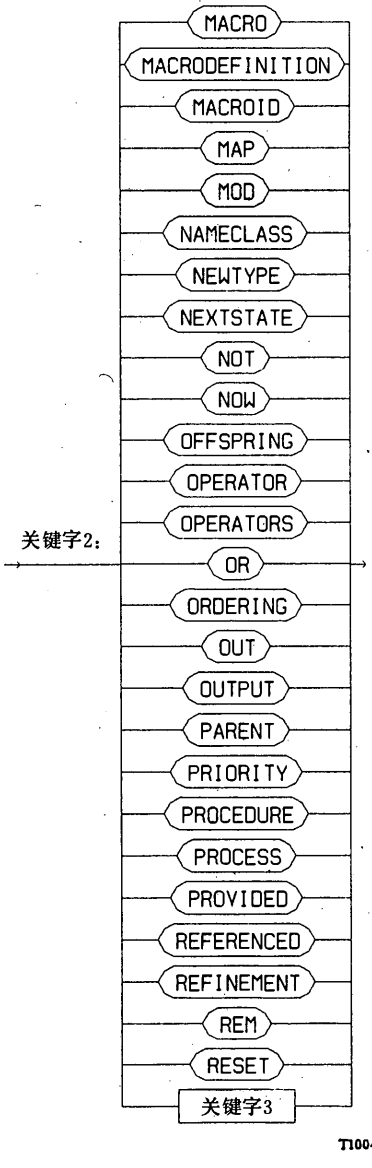

宏参量
:
宏形式参量名
MACROID
T1004780-88

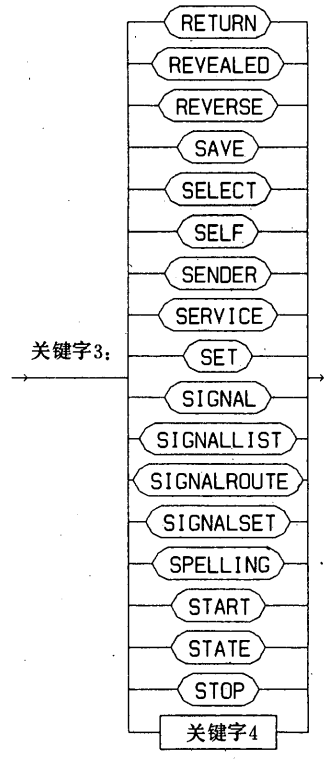


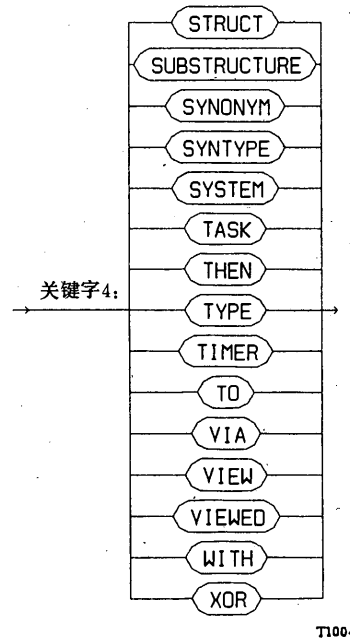
T1004790-88



T1004790-88


关键字2:
MACRO
MACRODEFINITION
MACROID
MAP
MOD
NAMECLASS
NEWTYPE
NEXTSTATE
NOT
NOW
OFFSPRING
OPERATOR
OPERATORS
OR
ORDERING
OUT
OUTPUT
PARENT
PRIORITY
PROCEDURE
PROCESS
PROVIDED
REFERENCED
REFINEMENT
REM
RESET
关键字3
T1004800-88


关键字3:
RETURN
REVEALED
REVERSE
SAVE
SELECT
SELF
SENDER
SERVICE
SET
SIGNAL
SIGNALLIST
SIGNALROUTE
SIGNALSET
SPELLING
START
STATE
STOP
关键字4


关键字4:
STRUCT
SUBSTRUCTURE
SYNONYM
SYNTYPE
SYSTEM
TASK
THEN
TYPE
TIMER
TO
VIA
VIEW
VIEWED
WITH
XOR
T1004800-88

索　　引

附 件 E

（附于建议 Z.100）

面向状态表示法及图形元素

E.1 引言

SDL 是以“扩展的”有限状态自动机（FSM）模型为基础的，这就是说，对有限状态自动机 FSM，用变量、资源等等事物来进行扩展。自动机停留在某一状态。当接收到一个信号时，自动机就执行一个跃迁，跃迁中将采取有关的动作（例如，资源的分配和（或）回收、资源控制、信号发送、判定等等）。因此，可以这样来说明一个扩展 FSM 的动态行为：即以过程方式来描述 FSM 的每一个跃迁中作用于各种事物的动作序列。

作为状态跃迁的结果，自动机到达一个新的状态。一个扩展 FSM 的状态的特性可以用下面的内容来刻画：与此状态有关的事物、辅助事物的信息（例如，变量的值、资源状态、资源之间的关系），以及在那个状态中可以接收的信号等。例如，在电话呼叫处理中“等待第一个数字状态”可表征如下：

主叫： 摘机

拨号音发送器： 拨号音正在发送

数字接收器： 已准备好可以接收

定时器： 监测固定的信号定时

路径： 主叫被连接到拨号音发送器和数字接收器等等。

可以看到，每一个状态可以用和那个状态有关联的事物和辅助信息（限定正文）来静态地定义。

为了定义关联于每一个状态的事物，我们用**图形元素**对 SDL/GR 进行了扩展。利用图形元素来作出的状态定义叫做**状态图**。SDL/GR 状态符号可以包含状态图，这是 SDL/GR 中可任选的部分。图 E-1给出了一个状态定义的例子：“等待第一个数字状态”。

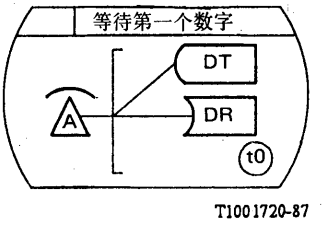


图 E-1

利用图形元素的状态定义例子

在很多情况下，在跃迁中需要的、关于每一个事物的动作可以从此跃迁以前的和以后的状态定义之间的差别中推出。例如，如果某资源仅在跃迁之后出现，则这意味着在跃迁中一定有资源分配动作。因此，如果给出了详细的状态定义，扩展 FSM 跃迁中的全部行为通常就能从先前的状态定义与后来的状态定义之间的差别中推出来。然而，跃迁中行为的顺序不能从状态定义差别中推导出来。这样，在 SDL 图中，当动作的顺序不那么重要时，那些可以从状态定义推导出来的跃迁动作不需要显式地描述。在其它情况下，则希望显式地描述动作的顺序。

在一个 SDL 图中如果跃迁都由显式的动作符号来详尽地描述，则称此图是 **SDL/GR 的面向跃迁型**。

在一个 SDL/GR 图中，如果采用状态图描述状态、并且跃迁动作减到最少，则称此图为 **SDL/GR 的面向状态型**、或称为**带有图形元素的面向状态的 SDL（SDL/PE）**。在某些系统定义中，应用状态图有很多好处，可以获得更紧凑、更清晰、而且文字较少的进程图。

混合型也是可行的。这样，有三种 SDL/GR 型式：

a) 面向跃迁型

— 跃迁序列由显式的动作符号详尽地描述。

— 这是扩展 FSM 的一种程序性的说明。

— 当动作的顺序是重要的，而详细的状态描述并不重要时，适合于采取这种型式。

b) 面向状态型

— 应用图型元素唯一地描述状态。这种图称之为状态图。

— 跃迁前的状态定义和跃迁后的状态定义之间的差别隐含了“跃迁的动作序列”。

— 它就是扩展 FSM 的一种说明性规格。

— 当每一个跃迁中的动作顺序并不那么重要、而希望有图形说明、或希望有一种紧凑的表示法的情况下，适合于采用此模型。

c) 混合型

— 当在每一个跃迁中的动作顺序和详细的状态描述都需要考虑的情况下，就适合于采用混合型。

在图 E-2、E-3和 E-4中，给出了这三种型式的例子。

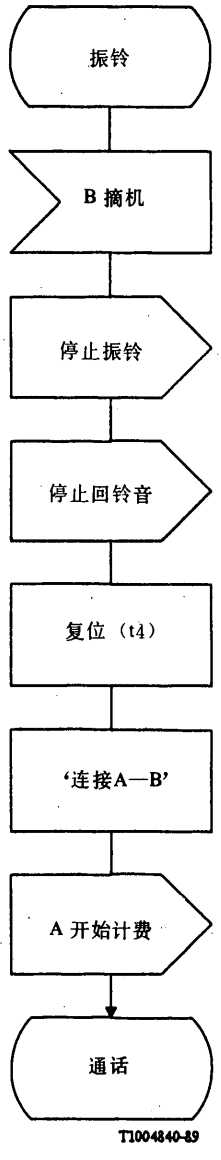


图 E-2

面 向 跃 迁 型

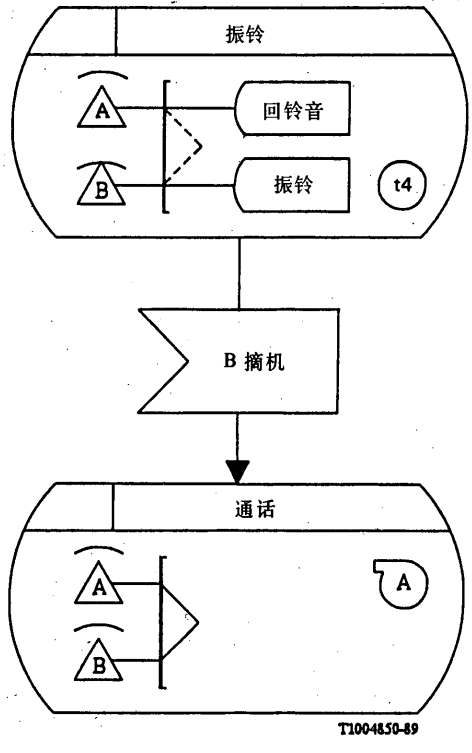


图 E-3

面向状态型

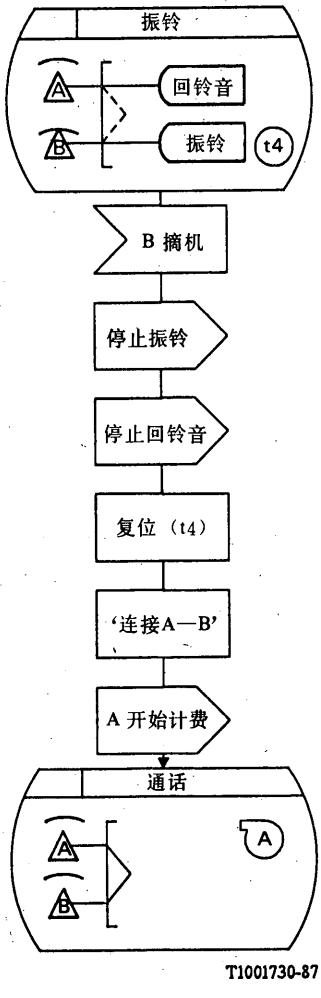

振铃
A
回铃音
B
振铃
t4
B 摘机
停止振铃
停止回铃音
复位（t4）
‘连接A—B’
A 开始计费
通话
A
A
B
T1001730-87

图 E-4

混 合 型

E.2 SDL/GR 中的图形元素

在 SDL 建议 Z.100中定义的语法和语义适用于图形元素。对这些语义和语法还作了如下扩展：

图形元素代表各种事物。全部图形元素的数量原则上不受限制，这是因为可以创造新的图型元素来适应 SDL 的任何新的应用。然而，在电信交换和信号功能的应用中，下述的一组图形元素具有很大的通用性：

— 功能块边界（左或右），
— 终端设备（多种）
— 信号接收器
— 信号发送器
— 信号收发器
— 监控定时器
— 交换路径（已连接，已保留）
— 交换模块
— 正在计费
— 控制元素
— 不确定符号

在 E.2.2节中推荐了这些图形元素的标准符号。

E.2.1 解释规则

1) 状态符号可以包含状态图。状态图使用图形元素和限定正文来定义状态。
2) 状态图中的每一图形元素代表关联于此状态的一个事物，例如：
   — 资源，
   — 变量，
   — 内部边界和外部边界，
   — 事物间的关系，
   — 在那个状态可以接收的信号，
   — 等等。
3) 每一图形元素可以具有伴随的限定正文。限定正文可以用于阐明：
   — 详细的资源名称，
   — 资源状态，
   — 变量的值，
   — 与该事物相关的信号，
   — 等等。
4) 功能块边界：
   a) 功能块边界用来表示一个图形元素对进程来说究竟是“内部的”还是“外部的”。内部的图形元素表示进程所拥有的事物，外部的图形元素表示属于另一个有关的进程所拥有的事物。
   b) 规则 a) 也适用于内部限定正文和外部限定正文之间的区分，这只需以术语“限定正文”取代规则 a) 中的“图形元素”。
5) 跃迁解释规则：

   在一个进程从一个状态发展到下一个状态的过程中所进行的全部处理动作是下面内容的组合：
   — 从两个状态定义的差别中可导出全部有关事物的改变，及为了造成这些改变所需要的处理动作。
   — 在跃迁中显式地描述的处理动作，例如输出和任务。

这样：

a）如果表示某个资源的一个图形元素没有出现在一个状态中，却出现在下一状态中，这就意味着：在所有连接两个状态的跃迁中必定要分配这个资源。这可以由一个在跃迁中表明此资源分配的任务来等效地表示。

b）如果规则 a）中的“出现”与“没有出现”互换，则“分配”应改为“回收”。

c）在规则 a）中如果“图形元素”由“外部的图形元素”所代替，则该任务将由一个输出信号来代替。此信号向拥有该资源的进程提出申请，要求分配资源。或者该任务仅简单地由来自该进程的一个输入信号所取代，说明该资源已分配了。

d）如果在规则 a）中“出现”和“没有出现”互换，同时“图形元素”也由“外部图形元素”取代，则规则 c）中的“分配”，应改为“回收”。

e）通过在规则 a）、b）、c）、d）中以术语“限定正文”代替图形元素，这些规则也适用于在状态图中限定正文的出现和消失。

6）在一给定的进程图中，特定的图形元素（或一个图形元素与限定正文的特定组合）无论何时出现，应该总是放置在状态图中的某个位置，以便在一个状态符号中这个图形元素（或组合）是否存在，能够通过比较该状态图与其它状态图就能迅速地判断。

7）当信号发送器出现在一个状态图中时，其限定正文要指明一个信号，该信号将在后续的跃迁期间被送出。

8）当一个常规信号（例如回铃音）的发送器出现在状态图中时，其限定正文指明了此信号在本状态中已开始发送，并在后继的跃迁期间发送。

9）有一些跃迁动作不能够从跃迁前的状态定义和跃迁后的状态定义之间的区别推导出来。这些动作应该在跃迁中显式地描述。例如，如果带有出口变量的某个资源没有出现在跃迁前的和跃迁后的状态中，则最好在跃迁中描述这些必需的动作。

E.2.2 推荐的图形元素符号

当使用图形元素时，每一状态用一个状态符号表示，其格式在图 E-5中给出。此符号中含有一个状态图。

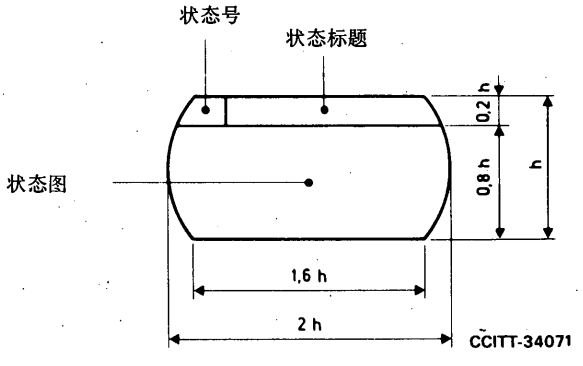


图 E-5

状态符号的推荐格式，其中含有状态图

已推荐了图形元素的一个基本集合，用于SDL/GR，应用于电信呼叫处理过程的系统描述，包括信令协议、网络服务及信号互相配合过程。这些图形元素的大多数都能够适用于呼叫处理过程以外的SDL/GR的应用。

图E-6给出了图形元素基本集合的推荐符号，而图E-7展示了图形元素符号的推荐特性。图E-8给出了图形元素基本集的应用例子。

E.2.3 在SDL/GR面向状态的扩展中使用的专门约定和解释

本节针对面向状态型的SDL/GR定义了若干的专门约定和解释，它们包括：

— 根据所谓的跃迁解释规则（参见§E.2.1，规则5），进程图所要求的专门解释。

— 在状态图中的图形元素（或图形元素与限定正文）的唯一位置，这在使用图形元素时是需要的（参见§E.2.1，规则6）。

— 对于外部图形元素及外部限定正文所表示的变量，需要有专门的解释，这些变量是关联于其它进程的代替变量。

E.3 图形元素的选择准则

关于图形元素符号的选择一直是以下面的理由和一般选择准则为基础的。在开发针对SDL的更广泛应用的其它图形元素符号以前，应当参考这些准则。

1） 容易复制

为了使得用染线或晒图的复制方法来复制SDL图，和照象复制及影印方法一样方便，图形元素符号应该由清晰的线条构成，不带有阴影或颜色。

2） 容易理解

a） 适合性——每一个符号的形状应该适合于此符号所表示的概念。

b） 区别性——在选择符号的基本集时，应当考虑使每一个符号能容易地与集合中的其它符号区别开来。

c） 相似性——代表不同的、但又有联系的功能的图形元素，例如接收器和发送器，其形状应当明显地有联系。

d） 缩写限定正文与符号的结合——某些情况下为指明图形元素的类别，希望把缩写正文与图形元素相结合；例如，将字母MFC与一个接收器符号结合在一起，就表示接收多频编码信号。在这些情况下，图形元素应含有封闭的空间，以便写入少量的字母数字字符。

e） 有限集——为了使图形方法学起来容易，符号的总数应保持在最低限度。

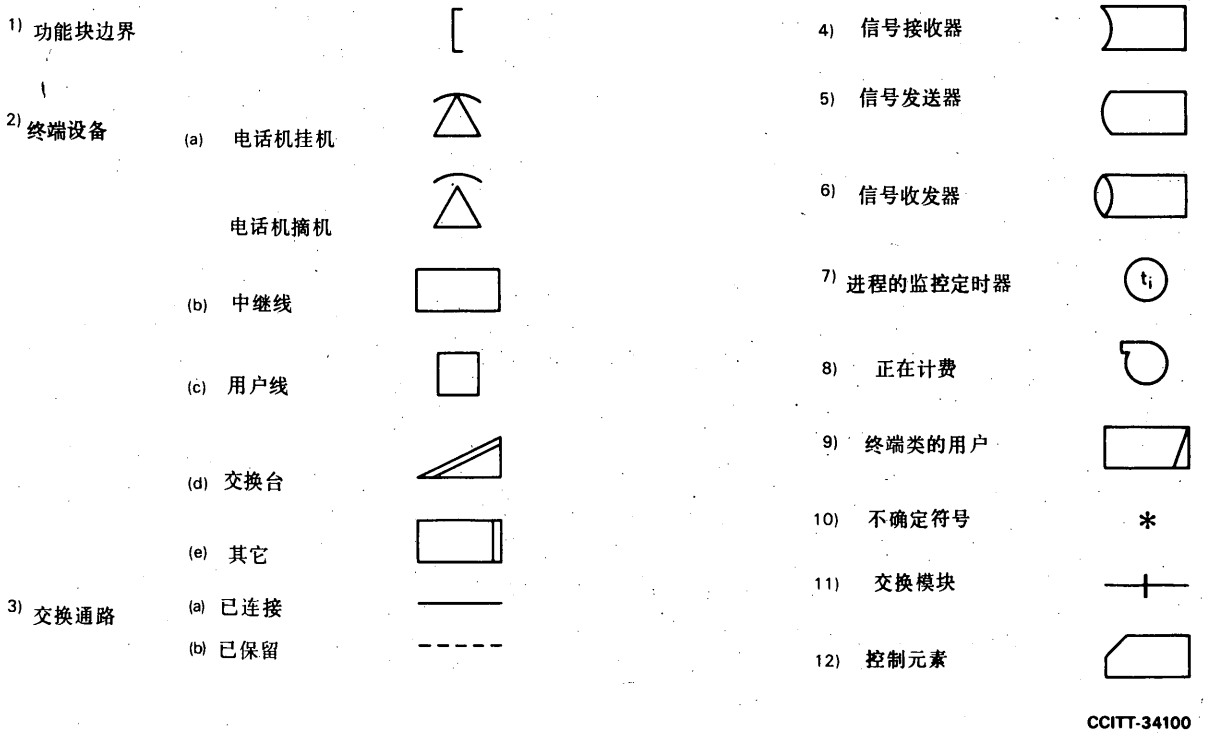


图 E-6

图形概念之基本集合的推荐符号

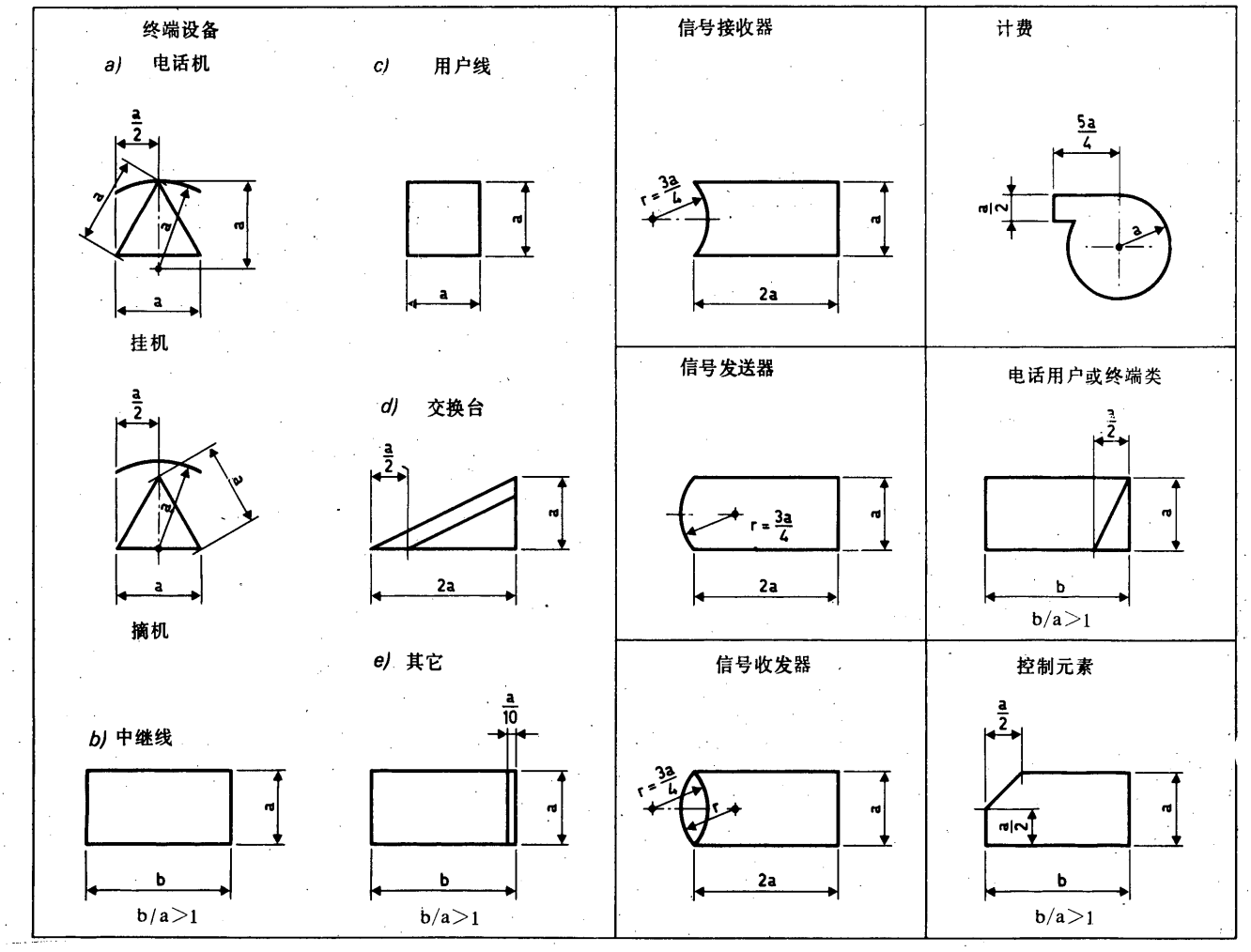
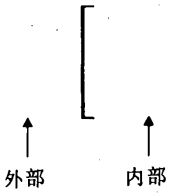
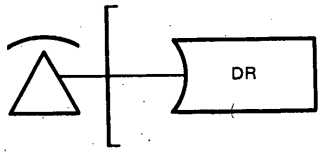
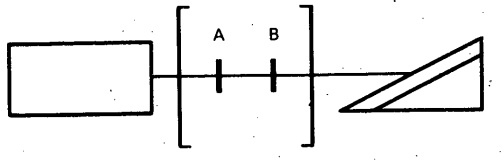
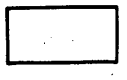
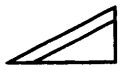
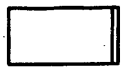
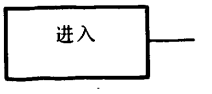
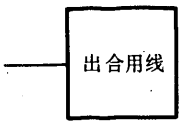
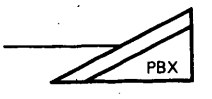
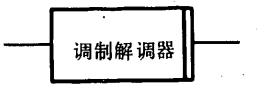


图 E-7

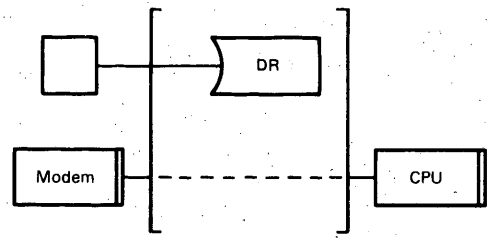
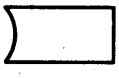
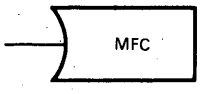
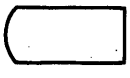
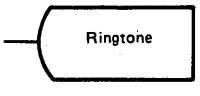
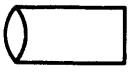
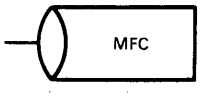
图形元素基本集的推荐比例

序号	图形元素	注释	举例
1.	功能块（FB）边界  	用来区别FB 边界内和边界外的事物。只有边界内的事物的状态可以直接由本进程直接改变。	1.1 FB 边界内的数字接收器连到边界外的电话机。  1.2 FB 边界外的中继线经两级交换单元连接到边界外的控制台。 
2.	终端设备 a) 电话机 挂机  摘机  b) 中继线  c) 用户线（除a）以外]  d) 交换台  e) 其它 	指出FB 边界外的终端设备（例如电话机和交换机设备）、对于增进对处理工作的理解可能是有利的。	2.1 A 挂机  2.2 B 摘机  2.3 入中继线连接器（来自空分交换机）  2.4 出用户线连往合用线  2.5 交换台  2.6 调制解调器 

CCITT-20880

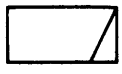
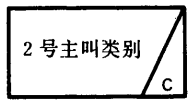
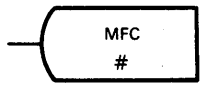
图 E-8

图形元素基本集的应用举例

序号	图形元素	注释	举例
3.	交换通路 a) 已连接 —————— b) 已保留 - - - - - -	表示进程中所涉及的终端设备与（或）信号设备之间的连接	3.1 用户线连到数字接收器及调制解调器以一条保留通路连到中央处理器（CPU） 
4.	信号接收器 	指所接收信号、特别是那些跨越功能块边界的信号的性质	4.1 多频编码信号接收器 
5.	信号发送器 	规定信号发送过程，并指明所发信号、特别是那些需要跨越功能块边界的信号的性质。	5.1 回铃音发送器 
6.	信号收发器 	这便于把信号发送器和信号接收器的功能结合起来	6.1 MFC 收发器 
7.	监视进程定时器 	用来表示在状态中运行的定时器	7.1 定时器t_3 正在运行  7.2 一般定时器t_3 正在运行  这里S = 1, 2, ……n，规定不同的服务音。

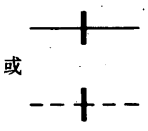
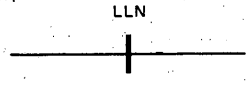
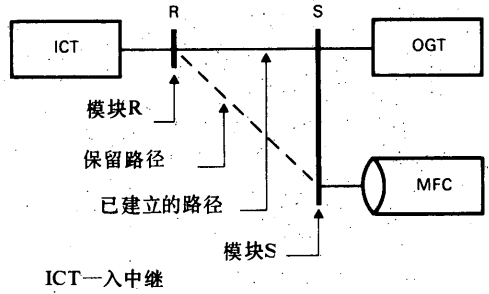
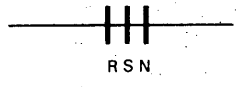
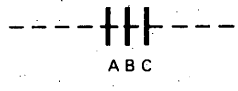
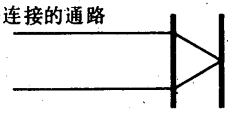
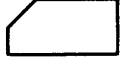
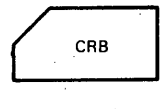
CCITT-20890

图 E-8（续）

序号	图形元素	注释	举例
8.	正在计费 	此元素中的限定正文指明正被计费的那个用户	8.1 当前正对用户A 计费 
9.	用户或终端类别 （及识别信息） 	对多分机呼叫中的每一个分机，此元素便于表明用户或终端类别的变化。	9.1 C 分机有 2 号主叫类别 
10.	不确定符号 # 	此符号代替在其它状态图中明确表示出的、故意不定义的信息。在某些情况下，通过应用不确定符号可以把两个或多个状态可靠地合成一个状态，使图可容易理解。	10.1 挂机或摘机的电话机  10.2 在此状态下，正送出一个未定义的MFC 信号 

CCITT-20900

图 E-8（续）

序号	图形元素	注释	举例
11	交换模块  或 	表示在进程中涉及哪些交换模块。 注：水平线是交换通路的图形元素，它可以是已连接或已保留的。垂直线可用来表示一个完整的交换模块（当不需要模块的内部结构时）、或表示一个交换模块中的一个交换级。	11.1 通过一个交换模块LLN (Line link network) 连接的一条通路 LLN = 线路链接网络  11.2 通过两个交换模块连接和保留的通路  ICT—入中继 OGT—出中继 MFC—多频编码 注—此例中，ICT 连接到OGT，而ICT 没有连到MFC 收发器 11.3 通过三级交换模块RSM连接的通路  11.4 通过三级交换模块ABC 保留的通路  11.5 通过一个折叠网络连接的通路 
12.	控制元素 （分配给一进程） 	表示在进程中所涉及的控制设备（尤其是那些必须标明尺寸的模块）。此符号可以用来指明已分配给进程的特殊软件元素。	12.1 呼叫记发缓冲器 

CCITT-20910

图 E-8（完）

建　议　Z.110

关于形式描述方法的使用及其适用范围标准①

1　对形式描述方法（FDT）的支持

鉴于建议的复杂性并且使用广泛，迫使我们采用先进的方法来开发和实现这些建议。

形式描述方法提供了一个重要途径，使得我们可以采用这样的先进方法。

在某些领域，FDT 的使用仍是比较新的，为引入其应用，需要一个分阶段的发展步骤。本建议提出了完成这个任务的步骤。

2　FDT

2.1　定义

形式描述方法（FDT）是一种制订规格的方法，它以一种描述语言为基础，这种语言使用严格的、无二义的规则，既用于产生在此语言中的表达式（形式语法），又用于解释这些表达式的意义（形式语义）。应把 FDT 用于建议的开发、制定规格、实现和验证方面（或其中的一部分）。

非形式描述方法的一个例子是自然语言描述。它使用 CCITT 发表建议所用的语言之一。自然语言描述还可以用数学符号和其它可接受的符号、图形等作为补充。

2.2　FDT 的目标

FDT 的目标是使得人们能够准确地，无二义地制定规格。FDT 也打算用来满足下面的一些目的：

— 作为一个基础用于分析规格的正确性、效率等；
— 作为一个基础用于确定规格的完整性；
— 作为一个基础用于对照建议的要求来验证规格；
— 作为一个基础用于确定具体的实现与建议的一致性；
— 作为一个基础用于确定在多个建议之间规格的相容性；
— 作为一个基础用于支持实现。

在目前的技术状态下，某些领域中可能需要不止一种 FDT，以达到上述的全部目的。

2.3　采用 FDT 的好处

应用 FDT 可以得到一些好处，例如：

— 改进了建议的质量，这一点对 CCITT 和对建议的用户来说，又降低了维护的费用；
— 在多种语言的环境中，降低了交换技术概念对自然语言的依赖性；
— 通过使用以 FDT 的特性为基础的工具，降低了具体实现的开发时间；
— 使得实现工作更容易，并导致更好的产品。

---

① 本建议的内容也作为 ISO 决议 ISO/IEC JTC 1/N 145发表了。在 JTC1文件中包含有：在有几种描述的情况时，有关各描述优先次序的叙述。在本建议中略去了这个叙述。

2.4　FDT 的问题

FDT 是还没有得到广泛介绍的先进技术。此外，在 FDT 的开发和形式描述（FD）中，还缺乏资源，同时，在 CCITT 研究小组内，还缺少专家去评价这些形式地描述的建议的技术优点和去收集有关这些建议的意见。

2.5　解决办法

编制个别指导性的和大众化的教育材料将有助于人们对 FDT 的复杂内容获得广泛的理解。然而，对于其消化吸收，必须允许一定的时间。

3　FDT 的标准化和发展

避免 FDT 不必要的增加是重要的。在采用一种新的 FDT 之前，下面的标准必须满足：

— 应当论证对此 FDT 的需求；

— 必须提供证据说明新的 FDT 的基础模型与已有的 FDT 模型有很大的差别，并且

— 应当论证此 FDT 的实用性和有效性。

4　形式描述的开发和认可

4.1　将来在建议的形式描述中，应该只用标准的 FDT 或正在标准化过程中的 FDT。

4.2　应该认为任何具体建议的 FD 的开发都是研究小组的决定（对共同的标准与 ISO 磋商）。如果为了一个新的建议而开发一种 FD，则只要可能、此 FD 就应该依据与建议的其它部分一样的时间表来开发。

4.3　为了逐步地把 FD 引入建议，可以分为三个阶段。研究小组有责任决定哪一个阶段开始应用到每一 FD，以及此 FD 朝另一个阶段发展的可能性。对一个 FD 来说，不一定要经历下面所描述的三个阶段，而且，更一般地说，也不强制去发展一个 FD。

阶段1

这一阶段的特征是缺乏 FDT 的广泛知识，以及缺乏形式描述的经验，在有关研究小组内可能没有足够的资源来产生或审查形式描述。

建议的开发必须以传统的自然语言方法为基础，所得到的用自然语言描述的建议是权威性的建议。

要鼓励研究小组去开发他们的建议的 FD，这些努力可以发觉缺陷从而提高建议的质量，可以为读者提供进一步的理解，并将支持 FDT 逐步进入使用。

如果由一个研究小组所研究出的形式描述，可以认为是忠实地表述了该建议的一个显著的部分，或表达了完整的建议，那么应该把此形式描述作为该建议的附件而发表。

同时，研究小组应当编制和提供 FDT 的教育材料，以支持 FDT 在 CCITT 和协作组织中的广泛介绍。

阶段2

此阶段的特征是已经可以较广泛地获得 FDT 的知识和形式描述方面的经验；研究小组可以提供足够的资源来支持形式描述的产品。然而，不能保证有足够多的 CCITT 成员能够审阅形式描述、以使他们能够赞同一个推荐的形式地描述的建议。

建议的开发仍应以传统的自然语言方法为基础，所得到的建议中，自然语言的描述仍是权威性的标准。然而，伴随这些开发工作，应该同时开发这些标准的形式描述，并在形式描述的支持下改进和支持自然语言描述的结构、一致性及正确性。

如果由研究小组研究出的形式描述、被认为是忠实地表达了建议的一个显著的部分、或表达了全部的建议，则应当作为建议的一个附件发表。

同时，教育工作应当继续。

阶段3

这一阶段的特征是：可以期望有了 FDT 的广泛知识；CCITT 成员可以提供充足的资源，既用于产生形式描述，也用于审查形式描述，并且可以保证采用了 FDT 并没有不必要地限制了实现方面的自由。

各研究小组应当常规性地使用 FDT 来开发他们的建议，并且这些 FD 与自然语言描述一起成为建议的一部分。

无论何时发现了自然语言描述与形式描述之间、或两种形式描述之间有矛盾，则要修改或改进自然语言描述或 FD 描述以解决这个矛盾，而不要偏爱任何一方。

4.4　提出上面对 FD 分阶段开发的步骤的意图是帮助 FD 的发展处于标准的步骤中，而不是妨碍它们的发展。然而，由于一直没有或很少关于这些步骤的实际经验，因此迫切要求任何应用 FD 的研究小组去认明一个或多个指导性的情况，并在分阶段开发的框架中认真地监察每一个 FD 的进展。如果出现发展步骤方面的问题，应该通知负责形式描述技术的研究小组，并且只要有可能就应当提出建议修改发展的步骤，以使问题得到缓解。

中国印刷 ISBN 92-61-03755-0