国 际 电 信 联 盟

CCITT

国 际 电 报 电 话 咨 询 委 员 会

蓝 皮 书

卷 X.2

建议Z.100的附件D
SDL用户指南

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1990年 北京

国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

蓝 皮 书

卷 X.2

建议Z.100的附件D
SDL用户指南

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1990年 北京

ISBN 92-61-03765-8

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 — 综合数字网及模拟-数字混合网中的数字市内局、转接局、综合局及国际交换局　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-网路管理　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F1：SDL形式定义 介绍（第X研究组）

卷X.4 — 建议Z.100的附件F2：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F3：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.2 目录

建议 Z.100 的附件 D

SDL 用户指南

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. 2

建议 Z. 100 的附件 D

SDL 用户指南

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

附 件 D

(附于建议 Z.100)

SDL 用户指南

目 录

D.1 序

CCITT 规格和描述语言通称 SDL，首先于1976年由建议 Z.101至 Z.103（桔皮书，卷 Ⅵ.4）给出定义，后来于1980年在建议 Z.101至 Z.104（黄皮书）中加以扩充，于1984年进一步得到扩充和承认，成为建议 Z.100至 Z.104（红皮书）。在1985－1988研究期中，再进一步对该语言进行了扩充和调整，原有的几个建议已被合并成一个，并且提供了数学定义。

为了便于把 SDL 广泛地应用于电信系统，需要有用户指南。用户指南的目的是为了帮助用户了解 SDL 建议及这些建议在不同领域的应用。

SDL 正被 CCITT 及其成员组织广泛地使用着，且其使用范围不断扩大。本用户指南旨在通过对 SDL 建议补充以具体的意见和有用的例子，来帮助正在考虑使用或已开始使用 SDL 的人们。本用户指南和建议本身在内容上有某些重复，这样做可使用户指南本身自成体系并且易于阅读。但建议依然是主导文件。

D.2 引言

D.2.1 SDL概述

SDL既可用于规定系统所应具有的行为，又可用于描述系统的实际行为。设计SDL时，主要着眼于用它来规定电信交换系统的行为，但它也适用于其他一些应用领域。实际上，对于系统行为能用扩展的有限态自动机（§D.2.1.1）来有效地模拟，且其重点在交互作用方面的所有系统，SDL都是很适用的。

SDL也可用作文件编制方法的基础，以便完整地表达一个系统规格或系统描述。在这个上下文中规格和描述的意义与它们在系统生命期中的应用有关。两者都抽象地定义一个系统的功能特点。描述通常包括某些与设计有关的方面（例如差错处理），对功能细节讲得比较完全。两者在涉及具体系统的设计时应协调一致，并且以后都用作为系统的文件。

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

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

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

一般说来，一份描述是由设备供应者为响应一份规格而写出的（虽然一份描述也可以是描述设备供应者企图出售的系统）。一份描述通常比规格具有较多的细节层次，因为它需要描述系统的具体行为。为简单起见，在以后各节中，SDL表示将专指规格而言。

还要指出，SDL提供了以不同的形式化程度描述一个系统的手段。

第一，可以用与自然语言有关的SDL构件来描述系统。这样形成的描述只将信息传递给具有上下文知识的阅读者，而不是传给计算机。计算机仅能自动地完成极其有限的校验工作。

第二，有可能与SDL构件形式语句有关，它由已定义的各种类型的元素和作用于这些元素的运算（或操作）符组成。这些元素的性质不是规范化的：例如“连接A—B”，这里A和B属于用户类型，而连接是该类型允许的一种操作。这样形成的规格将信息传递给了解所用操作符意义的阅读者，计算机能在一定程度上懂得这种规格，并能进行校验，但它不能进行完全的校验，也不能“实现”该系统，因巍—病≠作符的性质没有完全规定。

第三，有可能也提供全部运算符（或操作符）的所有性质。在这种情况下，规格是完全形式化的，计算机能进行全部校验，且能从概念上实现所描述的系统。

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

下文中规格一词将既用来表示所要求的行为，也用来表示实际行为。

D.2.1.1 SDL 的基础是扩展的有限态自动机（Extended Finite—State Machine）模型

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

1） 系统结构的定义，由抽象自动机及其相互连接来表述；

2） 每个自动机的动态行为，由它对其它自动机和环境的交互作用来表述；

3） 对交互作用数据的运算。

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

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

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

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

D.2.2 SDL 的语法形式

SDL 是一种语言，它具有两种不同的形式，两者都建立在同样的语义模型基础上。一种称为 SDL/GR（SDL 图形表示法），它的基础是一套标准化了的图形符号；另一种称为 SDL/PR（SDL 正文短语表示法），它的基础是类似程序的语句。两种表示法表示相同的 SDL 概念。

图形语言的优点是能够清晰地显示系统的结构并使人们易于看清控制流；而正文短语表示法则最适合于计算机使用。

作为一种设计工具，SDL 应当具有一种使得用户能清晰而简明地表示其思想的形式，

SDL/GR 体现了这一点并且更符合扩展的有限态自动机的惯例表示法。

SDL/GR 是 SDL 的最初形式，它是在1973－1976年期间产生的，并首次在 Z.100系列建议的1976年版本中出现。

SDL/GR 来源于一些图形语言，这些语言是各个组织机构为它们各自的需要而研制的。

SDL 的正文短语表示法（SDL/PR）是在1977－1980年研究周期期间创造的，但在把它作为建议之前需要作些改进。这一改进已在下一研究周期完成，从1984年起，SDL/PR 已成为 SDL 建议的具体语法之一。

起先，创造 SDL/PR 的目的是把它用作为一种易于将 SDL 文件输入计算机的方法，因为 GR 形式较难输入（需要图形外部设备来处理）。由于这个原因，重点是放在 PR 和 GR 之间一对一的映射上。此后随着图形终端的进展（性能的提高和成本的降低），GR 也适合于输入计算机了。而这并不降低 PR 的重要性和使用，因为有些用户发现它更合他们的心意，尤其是那些惯用编程语言工作的用户。

这种进展导致了 GR 和 PR 之间不严格的对应关系，所以我们还可以（容易地）把 PR 映射为 GR 或相反，不过每种形式具有它自己的特色。初看上去，PR 非常象编程语言（见图 D－2.2.1）。

```
–
–
STATE aw_off_hook;
INPUT off_hook;
TASK '激活计费';
TASK '连接';
OUTPUT reset_timer;
NEXTSTATE conversation;
–
–
```

图 D-2.2.1

SDL/PR 举例

事实上，关键在于根据什么可以把一段正文看成是一段编程语言。

如果定义一程序为“计算机能解释的信息”，则不仅 PR 而且 GR 都是“程序”。

但 SDL 规格和真正的程序之间有一些差别。首先，SDL 规格并不要求一定能为计算机所执行（虽然不反对这样做），而所要求的是它能够把准确的信息从一个人传递给另一个人。

若把一个 SDL 规格看成是程序，则可能认为该 SDL 规格是“错的 SDL 规格”（因为非形式正文的不完全性），若将它看作是系统功能要求的一种表示，则它就可以是完全有效的 SDL。

和程序的通常表示法的其它差别是在 SDL 规格的“风格”方面。

因为SDL的目的是用来支持人与人之间的通信，所以务必允许有不同的SDL布局，以便能用某种布局来指导读者集中注意力于某些被认为是更重要的方面。当然，这对程序来说是不重要的，因为已假定程序由计算机解释，计算机并不集中注意力于任一特定的方面，而是必须同等地考虑所有方面，它也不试图去“理解”程序。

由于PR与程序的类似性，使得一些很可能用CHILL来实现要求的程序员更喜欢PR。因此就有强大的引诱力来寻求从PR到CHILL的一对一映射，以便用PR表述的要求能自动地转换成CHILL代码。反过来也是令人感兴趣的，因为这样能从一个CHILL程序推导出一个PR规格。

在§D.9中，举例说明了把SDL映射到CHILL的几种可能的方法。

D.2.3 SDL的应用领域

图D—2.3.1示出可以使用SDL的范围，这是就电信交换系统的供求意义上来说的。

在图中，矩形框表示典型的功能组，他们的具体名称可随不同的机构而不同，但是他们的活动对许多管理机构和制造商来说都是典型的。每条有向线（流线）代表从一个功能组交给另一个功能组的一组文件，SDL可用来作为每一组文件的组成部分。此图仅用来作图解说明，既不是定义性的，也不是完善的。

凡是能够由相互联系的扩展有限态自动机有效地模拟的领域都可以应用SDL，例如电话、用户电报、数据交换、信令系统（如7号信令系统），信令系统与数据协议的互通，用户接口（MML）等。

当专门考虑SPC交换系统时，能使用SDL来作说明的功能的例子是：呼叫过程（如呼叫处理、路由选择、发信号、计费等），维护和故障处理（如报警、自动排除故障、系统构成、例行测试等），系统控制（如过载控制）和人机接口。

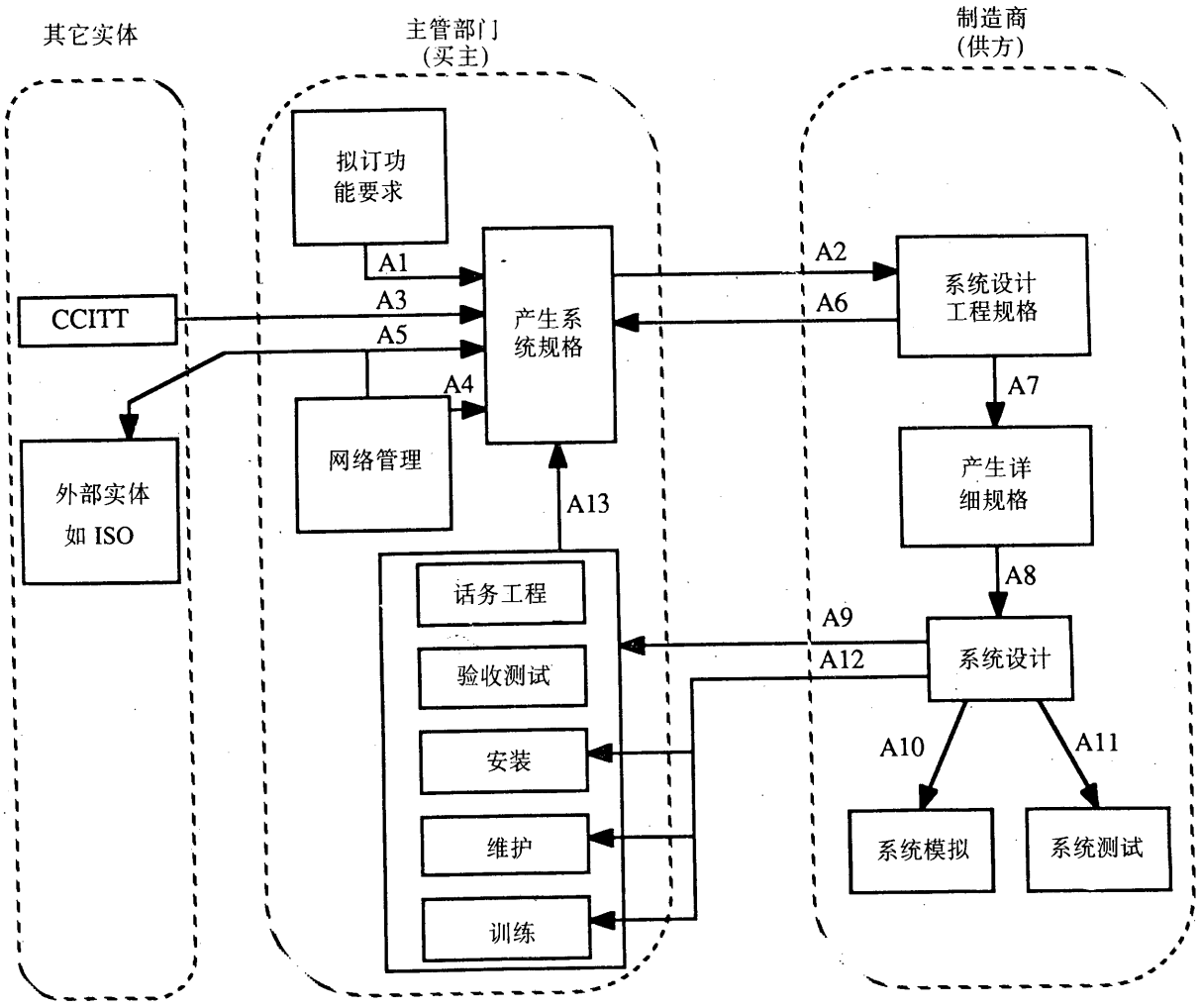
SDL的应用例子见§D.10。

用SDL来说明协议的规格已在CCITT X系列建议中介绍。

D.3 SDL的基本概念

D.3.1 系统

已如前述，SDL是用来构造系统模型的，因此系统就是由一个SDL规格所定义的。这就是说，一个SDL系统可以是下列设施的模型：一个电话系统（或交换局）的一部分；电话系统的一个完整网络；或者多个电话交换局的某些部分（例如在一条中继线两端的中继线控制器）。关键在于从SDL的观点看来，SDL系统包含有规格所试图定义的一切。诸如双份系统或多个等效节点等在实现方面的特点则不能由SDL显式地加以模拟。环境在规格以外，不在SDL中定义。



A1 一种功能或特性的规格要求，它与实现方法无关，并与网络无关

A2 一份与实现方法无关但与网络有关的系统规格，包括对系统的环境描述

A3 CCITT 建议和用户指南

A4 对系统规格提出有关网络管理和运行方面的要求

A5 有关的其它建议

A6 如何实现的建议及其描述

A7 一个工程项目的规格说明

A8 一份详细的设计规格说明

A9 一份完整的系统描述

A10 系统和环境的适当的描述文件，供系统模拟之用

A11 系统和环境的适当的描述，供系统测试之用

A12 安装和操作手册

A13 来自主管部门内部的专门功能小组对系统规格方面的意见形成的文件

注1 — 在所有层次上均可能反复多次。

注2 — 在某些情况下，此图中看作是一个机构内部的 SDL 文件如 A1、A7、A8，可以提供给其它机构。

图 D-2. 3. 1

使用 SDL 的一般情况

系统通过信道与环境连接。从理论上说，只需要一条双向信道来与环境连接，实际上，通

常为环境的每个逻辑接口规定一条信道。

每个系统由许多用信道连接起来的功能块组成。每个功能块相对于其它功能块而言是独立的。在两个不同功能块中的进程之间，通信的唯一手段是靠发送信号，信号通过信道来传递。需要将系统适当地分成几个功能块的依据可以是：把部件定义得其大小规模便于处理；能与实际的软件（硬件）划分相适应；与自然的功能划分相一致；把交互作用减到最少；以及其它。

对于一些大的 SDL 系统，有某些 SDL 构件可以规定系统部件的子结构，以便从系统的总体概貌出发，逐步提供愈来愈多的细节。在这种情况下，该系统可在不同层次上表示，而各层次的详细程度不同。这些构件将在§ D. 4中说明。

在第一个层次上，系统的 SDL 规格描述该系统的结构，并含有下列项目。这些项目将在下列段落中加以说明。

— 系统名字。
— 信号定义：规定在系统的各个功能块之间或功能块和环境之间相互交换的信号的类型。包括规定由信号传递的值的类型（类别表）。
— 信号表定义：规定一些标识符，用以把几个信号组合起来，以及（或）把其它信号表组合起来。采用这些标识符可以节省空间，并可使规格更加清晰。
— 信道定义：规定系统各功能块间互相连接以及功能块和环境间连接的信道。信道定义中要规定由该信道传输的信号的标识符。
— 数据定义：规定在所有功能块中可见到的、由用户定义的新类型、同义类型和生成程序。
— 功能块定义：规定把系统划分成诸功能块。
— 宏定义：使用宏的指南在§ D. 5. 1中给出。

按照 SDL 建议，有一种预定义数据类型，它是每个系统都可使用的。不需要去定义它们，就可通过它们的预定义名字来使用，即：INTEGER，REAL，CHARACTER，STRING，CHARSTRING，BOOLEAN，PID，TIME，DURATION。预定义数据类型在系统定义的任一层次上都是可见的，可认为它们是在系统数据库中隐式地定义的，可在规格中的任一位置使用它们。

在系统层次上，信号定义中所用的类别名字必须由系统层次上可见的部分类型定义来引入，即：预定义数据类型或在该层次上由用户定义的新类型或同义类型。

使用数据类型的进一步说明见§ D. 3. 10及§ D. 6。

系统定义的 SDL/PR 由一组语句组成，每个语句用“;”（分号）结束。系统结构的定义以语句“SYSTEM 名字;”开始，而以语句“ENDSYSTEM 名字;”结束。在结束语句中的名字可以写出也可不写，但只要写出，它就必须与关键字 SYSTEM 后面的名字相同。建议总是在结束语句中写上名字，因为这样可增加文件的可读性。

系统结构定义的 SDL/PR 概貌示于图 D-3. 1. 1。

为了使系统结构表示得更加清楚和简洁，同时也为了有一种自顶向下的系统规格，SDL 提供了一种通用的引用机制。在系统层次上引用机制可以用于功能块定义。SDL 语言的这一特点

使用户可以在系统结构定义之内仅只规定功能块名字，而真正的功能块定义可另行给出(见图D-3.1.2)。

这一引用机制在SDL/GR中特别有用，因为大多数图都必须容纳在一页上，往往没有足够的地方来画出嵌套的图形规格。

有关系统定义的SDL/GR例子可在§D.3.6中找到。

```
SYSTEM ...;
    ...     信号定义     ...
    ...   信号表定义       ...
    ...     信道定义       ...
    ...     数据定义   ...
    ...   功能块定义   ...
    ...     宏定义     ...
ENDSYSTEM ...;
```

图D-3.1.1

系统结构定义的SDL/PR概貌

```
SYSTEM s;
    ...
    BLOCK b1 REFERENCED;
    BLOCK b2 REFERENCED;
    ...
ENDSYSTEM s;
```

图D-3.1.2

功能块定义引用的例子

D.3.2 功能块

在功能块内部，进程间可通过信号或共享值来互相通信，这样，功能块不仅提供了一种把进程组合起来的合适机构，也提供了数据可见性的边界。由于这个缘故，在定义功能块时应小心，以保证归入一功能块中的一些进程是根据功能合理地组合到一起的。在大多数情况下，应先把系统（或功能块）分成功能单位，然后再定义功能块里的诸进程。

在功能块内部，能够（任选地）定义诸进程之间的通信路径，或者进程与功能块的环境

(即功能块边界）之间的通信路径，这种通信路径称为信号路由。

对于大的 SDL 系统，能够把一个功能块的子结构用另外一些功能块和信道来描述，好像该功能块本身就是一个系统。§ D. 4对这种机制做了说明。

功能块结构的定义可包含下列项目：

— 功能块名字。

— 信号定义：规定功能块内部相互交换的信号的类型。包括规定由信号传递的值的类型（类别表）。

— 信号表定义：规定与信号表相对应的标识符和（或）信号表的其它标识符。这些标识符把几个信号组合成群，可用来节省空间，并使规格更加清晰。

— 信号路由定义：规定功能块的诸进程间互相连接以及进程和功能块环境相连接的通信路径。定义中还规定了由该信号路由传送的信号的标识符。

— 信道到路由的连接：规定功能块外面的信道和功能块内部的信号路由之间的连接。

— 进程定义：规定进程的类型。这些进程描述了功能块的行为。若功能块不用其子结构来描述，则在该功能块内至少必须有一个进程类型定义。为进程定义提供了一种引用机制，这与 § D. 3. 1中所述的功能块的情况是类似的。

— 数据定义：规定由用户定义的新类型、同义类型和生成程序。它们在功能块所有已定义的进程中和（或）在功能块子结构中是可见的。

— 宏定义：使用宏的指南在 § D. 5. 1中给出。

若是有功能块的子结构，上述有些项目可以任选。（参阅 § D. 4，对系统构成的说明。）

在功能块中下列类型是可见的：

— 预定义数据类型；

— 由用户定义的、在功能块本身内定义的数据类型；

— 由用户定义的、在父本功能块内可见的数据类型（在功能块被划分的情况下）。

在 SDL/PR 中，用关键字 BLOCK 和 ENDBLOCK 来限定一个功能块定义的边界，功能块定义的 SDL/GR 例子可在 § D. 3. 6中找到。

D. 3. 3 信道

信道是系统的不同功能块之间或功能块和环境之间进行通信的手段。一条信道可以单方向地（单向信道）或双方向地（双向信道）将一个功能块连接到另一个功能块或者连接到环境。通常信道是一种功能性实体，可用来表示特定的信息通路。事实上通过划分信道（在 § D. 4. 5 中描述），可以形式地规定每个信道的行为。

一个信道规格为每个指定的方向（通信路径）给出一个信号表，列出能由该信道在该方向上传递的全部信号标识符。此信号表作为一种手段，用以保证由信道一端某一进程发送的每个信号，都能被位于信道另一端的功能块中的进程所接收。这样，信道规格就成为每个功能块接口规格的一部分。在有多人参加的大工程项目中，就信道中有哪些信号以及这些信号的规格方面及早达成一致，可减少两个进程不能按要求互相通信的概率。

信道的定义含有下列项目：

— 信道名字。

— 一条或两条通信路径：通信路径规定信号表中信号的始发端和终点。在这里的上下文中可使用功能块标识符或关键字“ENV”（环境）。

— 一个或两个信号表：对于每条通信路径都必须有一个信号表，规定在该方向上传递的信号。此表可包含信号标识符，或者也可包含其它信号表的标识符。

— 一个任选的信道子结构定义（或对它的引用）：参阅§D.4.5。

在SDL/PR中，信道定义用关键字CHANNEL和ENDCHANNEL括起来。关键字FROM和TO用来指明通信路径的两端，而关键字WITH则用来指明信号表。在图D-3.3.1中有一个SDL/PR的信道定义的例子。

```
SYSTEM so_and_so;
    . . .
    SIGNALLIST s_list_id_1 = sig_b, sig_c, sig_d;
    . . .
    CHANNEL c1
        FROM block_a TO block_b WITH signal_1,signal_2,signal_3;
    ENDCHANNEL c1;

    CHANNEL chan_2
        FROM ENV TO block_c WITH ext_sig_1,ext_sig_2;
        FROM block_c TO ENV WITH int_sig_1;
    ENDCHANNEL chan_2;

    CHANNEL chan.name.3
        FROM bx TO by WITH sig_a,(s_list_id_1),sig_e;
    ENDCHANNEL chan.name.3;
    . . .
ENDSYSTEM so_and_so;
```

图D-3.3.1

SDL/PR中的信道定义举例

在SDL/GR中，信道定义由连接通信双方的一条线来表示。信道名字必须较其它符号更靠近这条线，通信路径由箭头表示，而信号表必须用方括弧括起来，如图D-3.3.2中的例子所示。箭头不得放在线条两端的任一端，以避免信道和信号路由相混淆（见§D.3.5）。

在双向信道中，每个信号表必须最靠近与它相对应的箭头。

D.3.4 信号

信号能在系统层次、功能块层次上，或者在进程定义的内部来定义。在某个层次上定义的

信号可以在该层次上使用，或者也可在一些较低的层次上使用，然而为了使每个层次简化起见，建议将信号尽可能定义在局部的范围内。在进程定义内部规定的信号可在同一个进程类型的诸实例间互相交换（参阅§D.3.8），或者在进程内的诸服务之间互相交换（§D.5.3）。

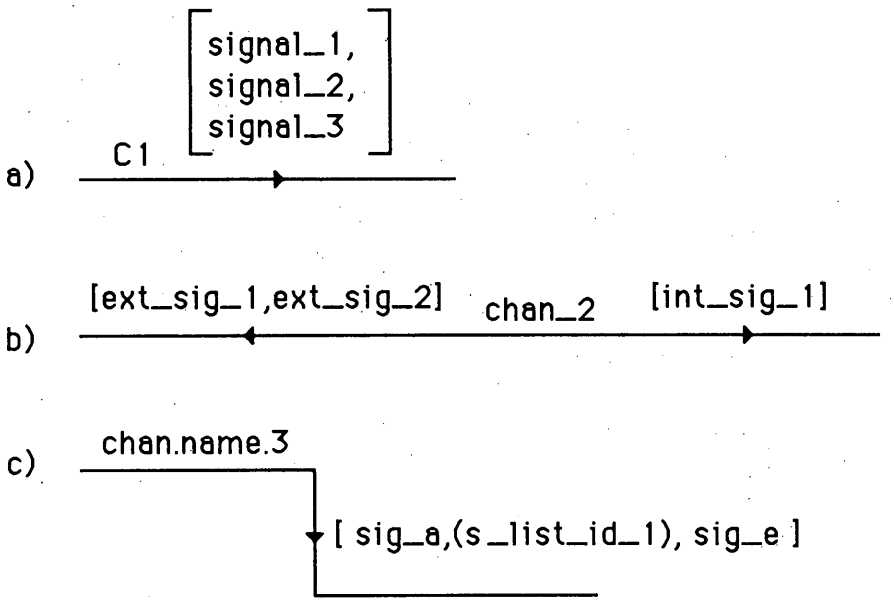


图 D-3.3.2
SDL/GR 中的信道定义举例

信号定义包含下列项目：

— 信号名字；

— 类别表（任选）：表示由该信号传递的值的类型的表；

— 信号具体化（任选）：在§D.4.7中说明。

在 SDL/PR 中，信号定义由关键字 SIGNAL 规定。在此构件中可定义几个信号（不经具体化），同时提供信号名和类别表。SDL/PR 中的信号定义例子在图 D-3.4.1中给出。

在 SDL/GR 中，信号定义由包围在正文符号内的写成一行一行的语句来规定。如图 D-3.4.2所示。

```
SIGNAL  s1,s2,s3;

SIGNAL  sig_a (sort1,sort2),
        sig_b (sort3,sort4);
```

图 D-3.4.1

SDL/PR 中的信号定义举例

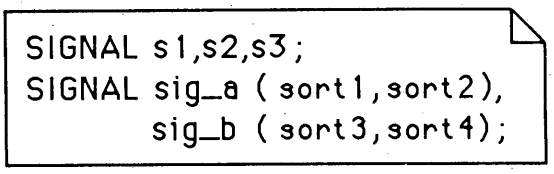


图 D-3.4.2

SDL/GR 中的信号定义举例

D.3.5 信号路由

与信道相类似，信号路由也用来表示通信路径。它们可使用在功能块层次上，也可使用在进程层次上。和信道一样，信号路由可以是单向的或双向的，但它们不能被划分。

在功能块层次上，它们表示一种在功能块诸进程之间或在进程和功能块环境之间通信的手段，这里功能块环境也就是通向或离开该功能块的一条信道。

在进程层次上，信号路由可用在将进程分解成服务子结构的场合（参见§D.5.3）。在这种情况下它们把服务互相连接起来，或者把服务连接到进程的信号路由上。

当一个信号被传到通向功能块边界的一条信号路由时，该信号即被交付给连接到该信号路由的信道。当一个信号从一条信道到达功能块，且该信道连接到一条或多条信号路由时，该信号即传向能够传递该信号的信号路由。

在 SDL/PR 中，信号路由的定义用关键字 SIGNALROUTE 开始。通信路径和信号表的语法和信道的情况一样。

在 SDL/GR 中，信号路由和信道之间的唯一差别是信号路由的箭头必须画在线的两端；箭头附近必须有适当的信号表。在下一节的图 D-3.6.3和 D-3.6.5中可找到信号路由的例子。

D.3.6 系统图和功能块图

在 SDL/GR 中，系统定义由一组图来表示。由信道和功能块组成的系统结构由系统图来

表示。

系统图包含有：

— 框符号：是一个包围所有其它符号的矩形符号。它表示系统边界：这个框的外面是系统环境。

— 系统标题：关键字 SYSTEM 后面紧跟系统名字（写在框的左上角）。

— 一个任选的页码编号（写在框的右上角）。

— 正文符号：这样一个符号可把一行一行的语句围起来。它通常用来给出图内的信号、信号表和数据的定义。

— 功能块交互作用区：这包括系统的功能块、信道以及由信道传输的信号表的规格。

— 宏图：使用宏的指南在§D.5.1中给出。

在系统图中，功能块的规格可为下列两者之一：

— 一个功能块引用标记：是一个功能块符号，其中仅含有功能块名字。

— 一个功能块图：是一个框，其中规定了功能块的结构（由它的诸进程及其相互作用来表示）。如果功能块分解成子功能块，则在功能块框内必须给出这些子结构的规格或其引用标记（§D.4.3）。

系统图和功能块图中所用的符号的形状，可在 SDL/GR 概要中找到。符号的尺寸请参阅§D.7.1.4。

图 D-3.6.1中有一个系统“s”的系统图的例子。在此例中，系统 s 被分成两个功能块 B1和 B2，它们由信道 C1、C2、C3和 C4互相连接并连接到环境。本例中对功能块 B1和 B2只给出了引用标记，对信道符号和信号表符号的进一步说明在后文给出。

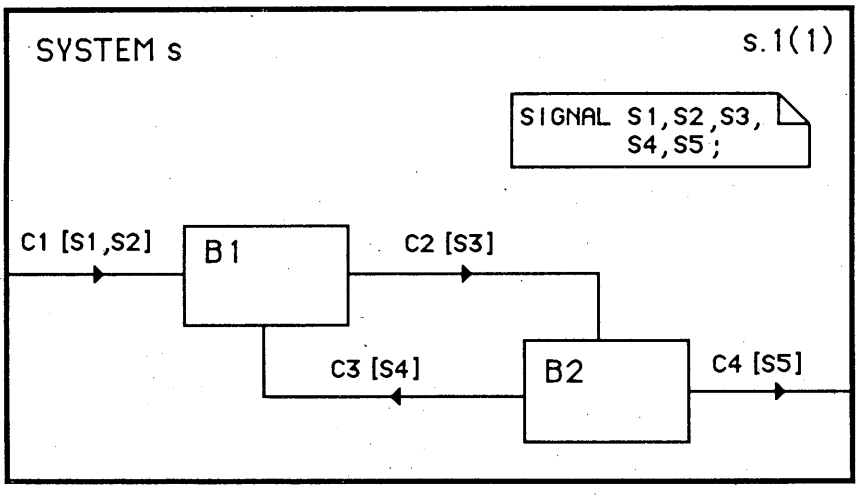


图 D-3.6.1

系统图举例

用 SDL/PR 表示的同一例子示于图 D-3. 6. 2。

用进程和信号路由表示的功能块结构在 SDL/GR 中用功能块图来表示。

```
SYSTEM s;
    SIGNAL S1,S2,S3,S4,S5;
    CHANNEL C1 FROM ENV TO B1 WITH S1,S2;
    ENDCHANNEL C1;
    CHANNEL C2 FROM B1 TO B2 WITH S3;
    ENDCHANNEL C2;
    CHANNEL C3 FROM B2 TO B1 WITH S4;
    ENDCHANNEL C3;
    CHANNEL C4 FROM B2 TO ENV WITH S5;
    ENDCHANNEL C4;
    BLOCK B1 REFERENCED;
    BLOCK B2 REFERENCED;
ENDSYSTEM s;
```

图 D-3. 6. 2

SDL/PR 中的系统结构定义举例

功能块图的组成是：

— 框符号：一个包围所有其它符号的矩形符号。它表示功能块的边界：在这个框的外面是功能块环境。

— 功能块标题：关键字 BLOCK 后面紧跟功能块名字（写在框的左上角）。

— 一个任选的页码编号（写在框的右上角）。

— 正文符号：这种符号可把一行一行的语句围起来。它通常用来给出信号、信号表和数据的定义。

— 进程交互作用区：这包括功能块诸进程的规格，也可能包括信号路由及由这些路由传输的信号表的规格。在这个区域内也可能示出创建其它进程的进程，这个特性在§D. 3. 8. 1中描述。

— 信道标识符：若图中标出通向或离开功能块环境的信号路由，则必须在框外标明连接这些信号路由的信道的标识符，与信号路由线相对应。

— 宏图：使用宏的指南在§D. 5. 1中给出。

在功能块图中，进程的规格可为下列两者之一：

— 进程引用标记：是一个含有进程名字的进程符号，且任选地也可含有进程实例的规格。这样一种进程实例的规格由一对整数组成，彼此用逗号隔开，并括在圆括号内（参阅§D. 3. 8）。

— 进程图：是一个框，它含有由一些符号连接起来的流图，这些符号表示状态、输入、

输出、动作等，以此来描述进程的行为（参阅§D.3.8）。如果进程分解成子结构—服务，则进程图应包含服务交互作用区（§D.5.3）。

若功能块有子结构，上述有些项目可以任选（参阅§D.4，对系统构成的说明）。

图D-3.6.3中有一个功能块图的例子，即图D-3.6.1的例子中所介绍的功能块“B1”。功能

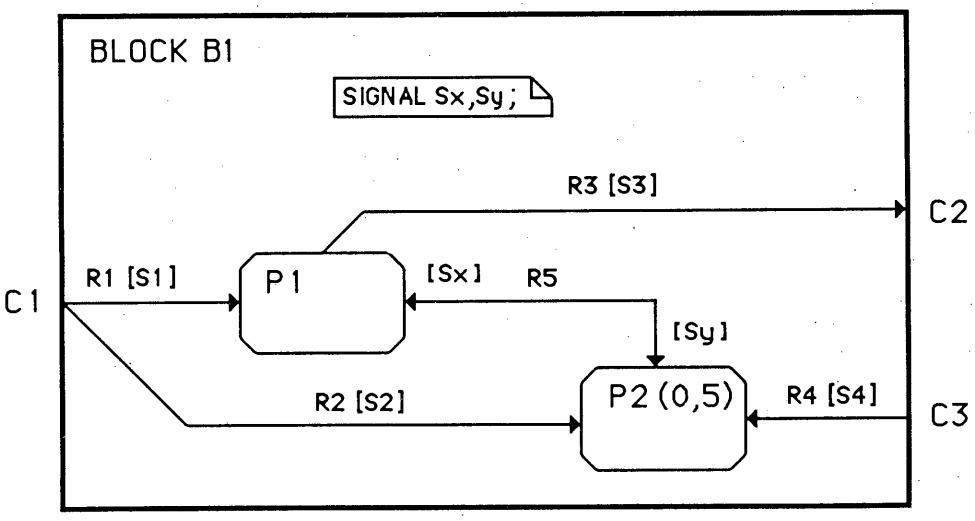


图 D-3.6.3

功能块图举例

块 B1被描述成与信号路由 R1、R2、R3、R4和 R5相连接的两个进程 P1和 P2。在本例中进程 P1和 P2只给出了引用标记，在框外还标出了信道标识符 C1、C2和 C3。

在 SDL/PR 中，同一例子示于图 D-3.6.4。

```
BLOCK B1;

    SIGNAL Sx,Sy;

    SIGNALROUTE R1 FROM ENV TO P1 WITH S1;
    SIGNALROUTE R2 FROM ENV TO P2 WITH S2;
    SIGNALROUTE R3 FROM P1 TO ENV WITH S3;
    SIGNALROUTE R4 FROM ENV TO P2 WITH S4;
    SIGNALROUTE R5 FROM P1 TO P2 WITH Sy;
                   FROM P2 TO P1 WITH Sx;

    CONNECT C1 AND R1,R2;
    CONNECT C2 AND R3;
    CONNECT C3 AND R4;

    PROCESS P1 REFERENCED;
    PROCESS P2 REFERENCED;

ENDBLOCK B1;
```

图 D-3.6.4

SDL/PR 中的功能块结构定义举例

如前所述，功能块图可以包括在系统图之内，来代替用引用标记，例如图 D-3. 6. 5中的例子，已把图 D-3. 6. 1和 D-3. 6. 3两个图合并成一个系统图。

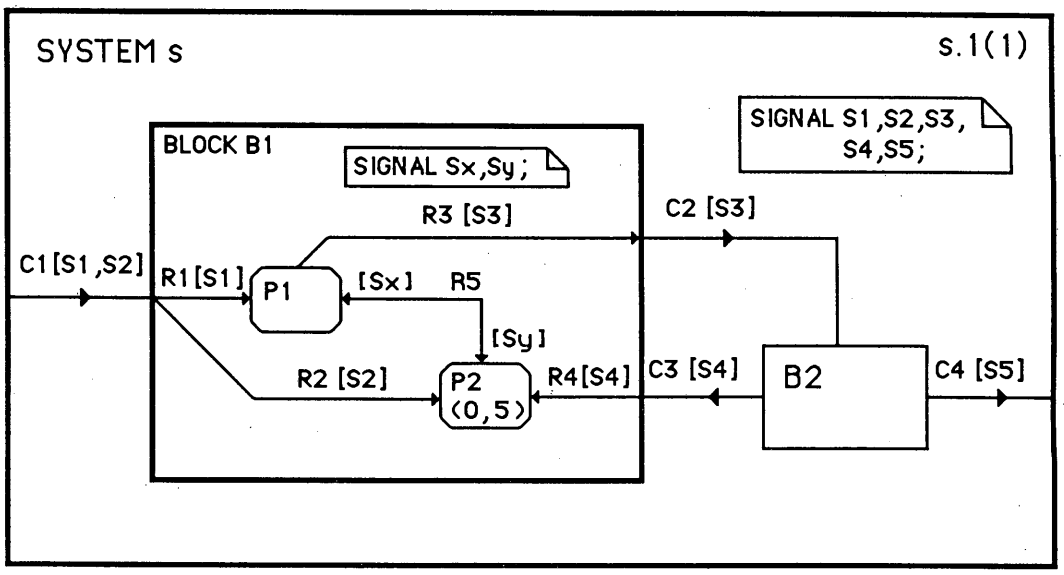


图 D-3. 6. 5

含有功能块图的系统图举例

画这类图的一般准则是不应使图太复杂，以便于阅读，并使它能容纳在一页之内。

D. 3. 7　注释和正文扩展

D. 3. 7. 1　注释

可以把注释加在 SDL 规格上，以帮助读者并使某些部分易于理解。SDL 有两种注释类型来适应 SDL/PR 及 SDL/GR。

第一种类型称为“注”，特别适用于 SDL/PR。它以“/*”开头、以“*/”结尾来定界。

在 SDL/PR 中，这种注释可在出现空格的任何地方插入。注释必须不含专用序列“*/”。

在 SDL/GR 中，这种注释可放在语句行内出现空格的任何地方。

图 D-3. 7. 1和图 D-3. 7. 2中示出这种注释形式的一些例子，有在 SDL/PR 中的，也有在 SDL/GR 中的。

注释的第二种形式可以在 SDL/PR 和 SDL/GR 之间作出一对一的映射，较适合于进行自动翻译的应用场合。

在 SDL/PR 中，这种注释由关键字 COMMENT 后面跟一字符串组成；它作为一个语句（即

后面跟一个“;”),可以在凡可插入任务语句的任何地方插入。此外,它还可在任何语句的结尾、符号“;”(分号)之前插入。

```
SYSTEM NSS;

/*国家交换系统的定义

  系统名字: NSS*/

/*****************************/

  ...
  SIGNAL s1; /*有关使用信号 s1 的说明*/
  ...
  BLOCK MAI REFERENCED; /*维护*/
  ...
ENDSYSTEM NSS;
```

图 D-3.7.1

SDL/PR 中注释的第一种形式举例

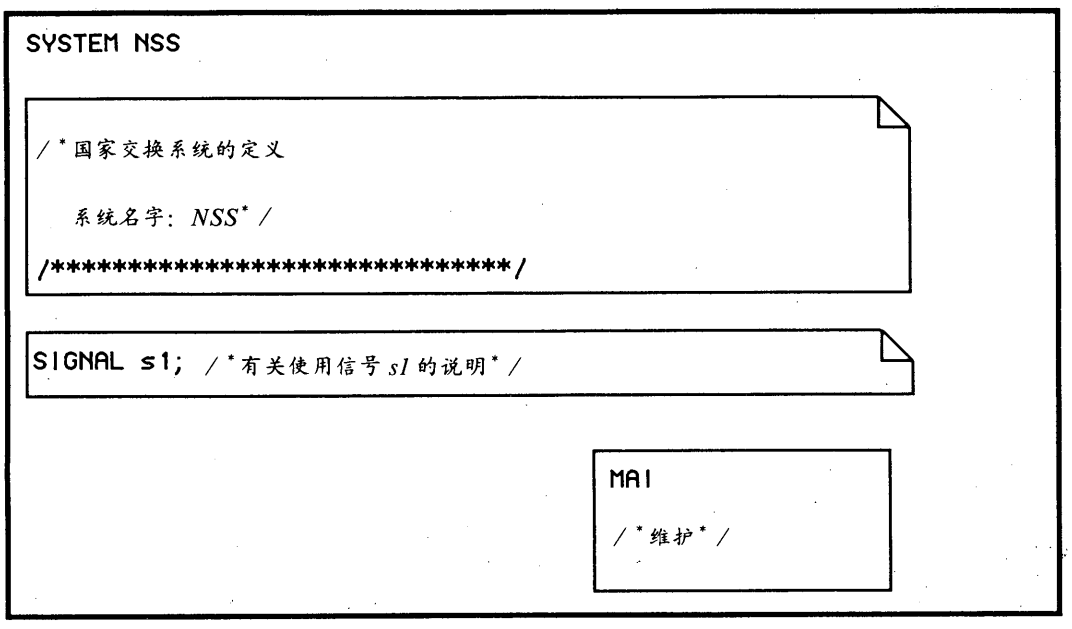


图 D-3.7.2

SDL/GR 中注释的第一种形式举例

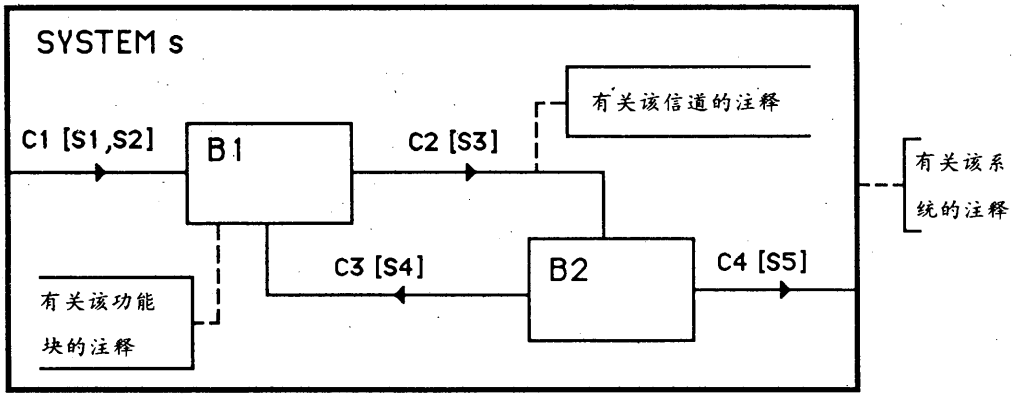
在SDL/GR中，这种注释形式由一含有注释正文的注释符号表示。注释符号是一个矩形符号，但缺少左方或右方的边，它必须在水平和垂直两个方向上延伸到足以包含全部正文。可把它连接到任何SDL/GR符号或流线上，连接线必须用虚线。如果注释正文和注释符号间的对应是明确的，则注释符号可简单地画成一个方括号。

在图D-3.7.3和D-3.7.4中示出SDL/PR和SDL/GR中这种注释形式的若干例子。

```
SYSTEM s COMMENT '有关该系统的注释'
    CHANNEL C2
        FROM B1 TO B2
        WITH s3          COMMENT '有关该信道的注释'
    ENDCHANNEL C2;
    BLOCK B1
    COMMENT '有关该功能块的注释'
        . . .
        PROCESS p1;
            . . .
            TASK 't1';
            TASK 't2';
            . . .
        ENDPROCESS p1;
    ENDBLOCK B1;
ENDSYSTEM s;
```

图D-3.7.3

SDL/PR中注释的第二种形式举例



图D-3.7.4

SDL/GR中注释的第二种形式举例

D.3.7.2 正文扩展

通常与一图形符号有关的正文应置于该符号的内部，但这常常是不可能或不实际的。代替的办法是把正文写在正文扩展符号内，并把扩展符号连接到有关的符号。正文扩展符号和注释符号相似，唯一的不同是连接线是实线而不是虚线，参见图 D-3.7.5。

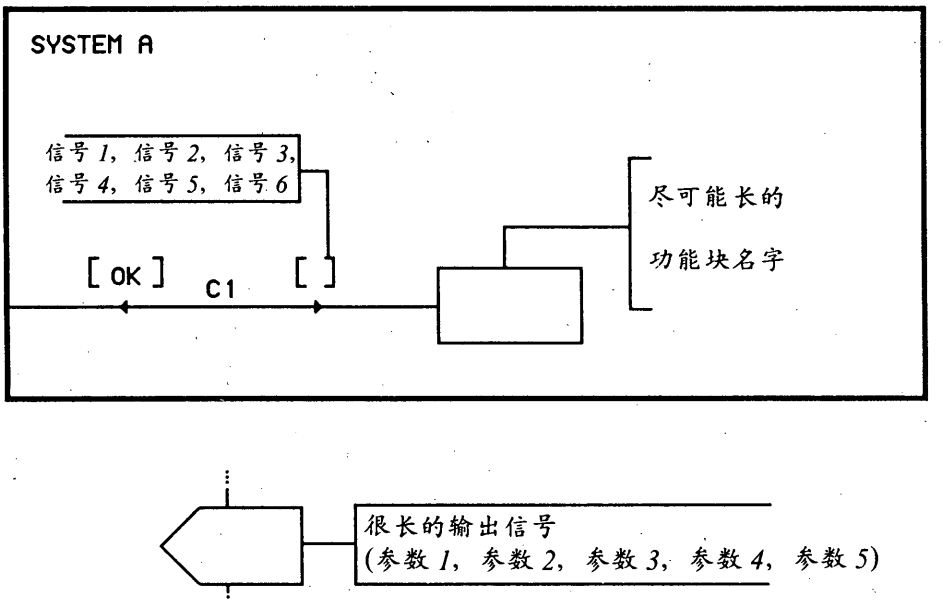


图 D-3.7.5

正文扩展符号使用举例

一行正文的末尾 可用一个底线符号（“—”）来作为连续符，这时同一行的余下的空格不看作是正文的一部分。关于名字的语法的更具体的规则见 § D.3.13。

D.3.8 进程

进程是一种扩展的有限态自动机，它规定一个系统的动态行为。进程基本上是处于等待信号的状态。当收到一个信号时，进程作出响应，执行特定的动作。应根据该进程所能接收的每个信号类型来规定相应的动作。进程含有许多不同的状态，使它在收到一个信号时能执行不同的动作。这些状态提供了早先已出现过的动作的记忆。根据收到的具体信号作完相应的全部动作以后，就进入下一个状态，这时进程又等待下一个信号。

进程可以在系统创建时就存在，或者作为另一进程发出创建请求的结果而被创建。另外，进程能够一直生存下去，或者能通过执行一个停止动作而停止。

一个进程定义规定了一类进程，可以创建同一进程类型的多个实例，这些实例可以同时存在，可以各自独立地以及并发地执行。进程定义由下列各项组成（其中有些是任选的）：

— 进程名字。

— 一对整数：第一个整数规定在系统创建时所创建的进程实例的数目，如果它被省略，就具有隐含值1；第二个整数规定同时存在的进程实例的最大数目，若它被省略，隐含的最大值不受限制。

— 形式参数：一张附有变量类别的变量标识符表，用来在进程创建时刻传递信息。为此目的，在进程创建请求中可提供一张实在参数表。在系统创建时创建的进程其形式参数的值是未定义的。

— 有效输入信号集：一张信号标识符表，规定进程所能接收的信号。

— 信号定义：规定能在同一进程的诸实例之间或者在进程内的诸服务之间相互交换的信号（§D.5.3）。

— 过程定义：规定能被进程调用的过程。在此上下文中可使用过程引用标记（过程在§D.3.9中说明）。

— 数据定义：规定局限于该进程的由用户定义的新类型、同义类型和生成程序。

— 变量定义：声明进程变量。一个变量可任选地声明为可在同一功能块内为其它进程所共享（REVEALED 变量），或者也可输出到其它功能块中的进程（EXPORTED 变量）。对每个被声明的变量必须规定其类别的标识符。可以任选地规定变量的初值。

— 视见定义：所声明的变量的标识符，可用来取得由其它进程实例所拥有的变量的值。对每个变量标识符必须规定变量类别。

— 进口定义：规定本进程要想输入的、为其它进程所拥有的变量的标识符。变量类别必须加以规定。

— 计时器定义：在§D.3.11中说明。

— 宏定义：使用宏的准则在§D.5.1中给出。

— 进程体：用状态、输入、输出、任务等来规定进程的实在行为。若进程由子部件（服务）构成，则进程定义应包含一节服务划分来代替进程体。这一特性的规则在§D.5.3中给出。

有关数据、变量、视见定义和输入定义的例子及说明在§D.3.10中给出。

在 SDL/PR 中进程定义的一个不完整例子示于图 D-3.8.1（SDL/PR 的关键字用大写字母标出）中。

进程体表示有限态自动机的实际流图。它在 SDL/PR 中由一系列排好次序的语句组成，而在 SDL/GR 中，它是由一些有向弧连接起来的符号序列（类似流程图）。在规定进程体时必须总是以 START 开始，下接一组动作（跃迁）。当一个进程实例创建时，就开始解释该进程实例。

```
PROCESS p1 (2,100);
   FPAR var1,var2 sort1,
        var3 sort2, var4 sort3;     } 形式参数

   SIGNALSET s1,s2,s3;              } 有效输入信号集

   SIGNAL s4,s5;                    } 信号定义
   SIGNAL s6 (sort6,sort7);

   PROCEDURE ...                    } 过程定义

   ...      数据类型定义      ...   } 数据类型定义

   DCL ...                          } 变量定义

   ... 进程体          ...          } 进程体

ENDPROCESS p1;
```

图 D-3. 8. 1

SDL/PR 中进程定义的不完整例子

在一次跃迁中可能执行的动作为：

— 任务：变量赋值（或非形式正文）。

— 输出：变量输出。

— 置位：请求激活一个计时器。

— 复位：将一个计时器复原。

— 输出：向其它进程发送一个信号。

— 创建请求：创建一个所指定的进程类型的实例。

— 判定：根据对问题的应答，选择一组动作。

— 过程调用：要求执行一组独立的、自成体系的动作（达到编程语言的要求）。

— 汇接：规定“跳”到另外一组动作上去。

一次跃迁可以用下列动作之一结束：

— 下一个状态：规定该进程实例将要达到的状态。

— 停止：进程实例立即停止。

在规定了启动动作以及任选的启动跃迁以后，进程体就包括该进程所有可能的状态的定义。每个状态定义首先规定进程在该状态等待的可能的激励。可能的激励是：

— 输入：可以接收的信号。
— 保存：需要保存的信号，留待将来处理。
— 允许条件：在§D.3.8.5中说明。
— 连续信号：在§D.3.8.5中说明。

除了保存以外，对应于上述每一种激励都必须规定一个跃迁，这样一种跃迁表示若该激励出现时进程将执行的动作序列。

如进程执行停止动作而尚有待处理的信号，其中这些信号已被送出但尚未被该进程收到，则这些信号被丢弃。

在SDL/GR中，进程定义由进程图来表示。进程图包含下列项目：

— 框符号：是一个围住所有其它符号的矩形符号。如果没有信号路由连接到框符号，则它可省略。
— 进程标题：关键字PROCESS后面紧跟进程标识符，然后依次为可能的进程实例规格和形式参数规格。进程标题写在框的左上角。
— 一个任选的页码编号（写在框的右上角）。
— 正文符号：在进程图中，正文符号可用来放置信号、变量、视见、进口、数据及计时器等的定义。
— 过程引用标记：含有过程名字的一个过程符号，表示该进程的一个单独定义的过程。
— 过程图：进程中未加引用的局部进程的过程，每个过程都有一张过程图。
— 进程流图区：规定进程的行为，它包括有起动、状态、输入、输出、任务……等以及有向弧。若进程由服务构成，则进程流图区应含有这些服务的规格或它们的引用标记（见§D.5.3）。进程体所使用的GR符号可从SDL/GR概要中找到。
— 宏图：使用宏的规则在§D.5.1中给出。

图D-3.8.2中有一个SDL/GR的进程定义例子。

有关进程流图的更多解说和例子将出现在以后各章中。

如果一页的页面容纳不下一个进程流图，则该图可画成几页，并注意：

— 应提供页码编号，编号应具有页数和总页数，如：1（9）。
— 可用汇接符或下一状态符来表示进程流图不同部分之间的连接。

把一个进程流图分成几部分的一个好准则是在每一页上表示一个状态定义。若一个状态定义在一页中容纳不下，可用汇接符来连接画在另一页上的图的另一部分。

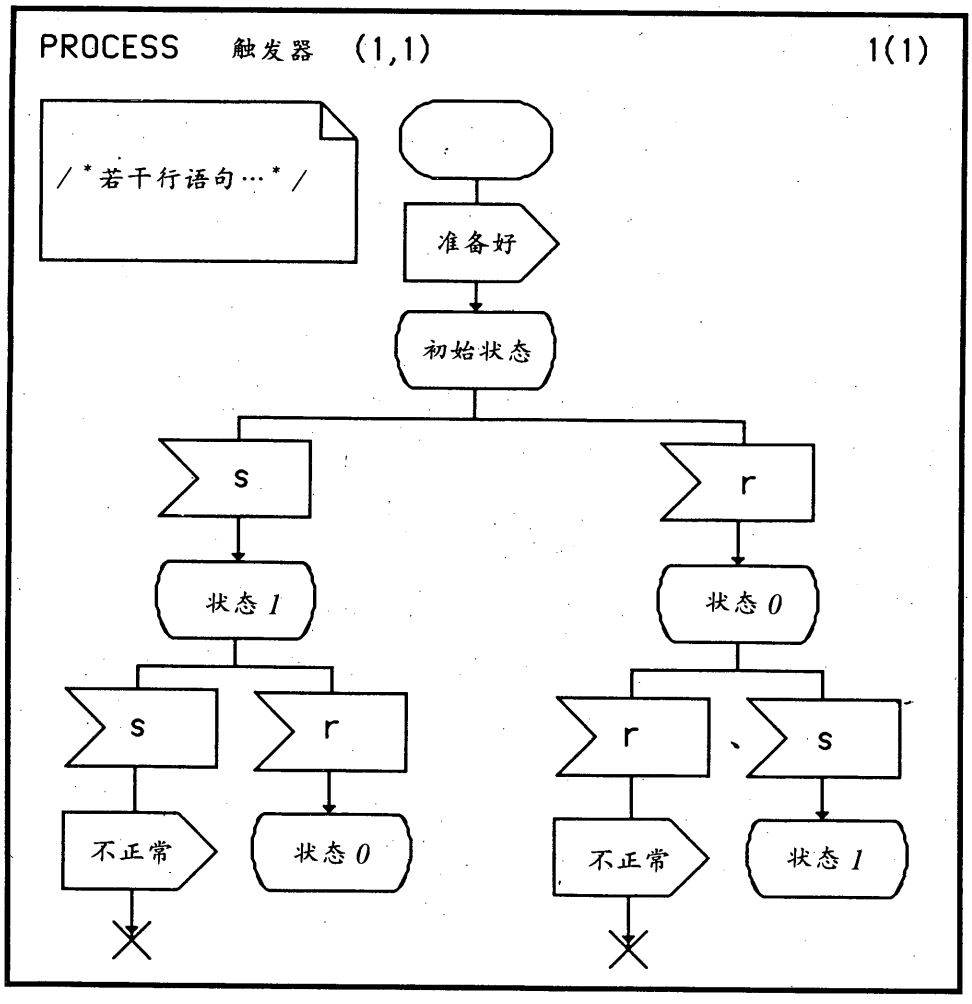
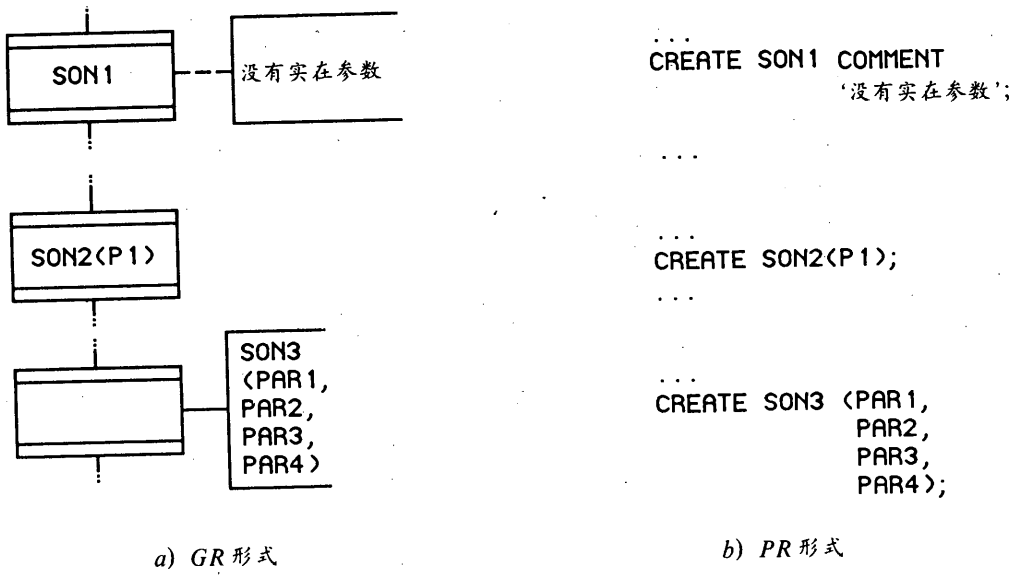


图 D-3. 8. 2
进程图的例子

D. 3. 8. 1 进程创建

如前所述，多个进程（即多个进程实例）可作为一次显式请求的结果而创建，或者可在系统创建时创建。

显式的创建请求只能由与被创建进程在同一功能块内的另一进程提出，并规定实在参数，向被创建的新进程传递信息。用 PR 及 GR 形式举出的创建例子见图 D-3. 8. 3。



a) GR 形式　　*b) PR 形式*

图 D-3.8.3

用 SDL/GR 及 SDL/PR 表示创建请求的例子

如果在系统创建时创建一给定进程类型的一个或多个进程实例，则该进程的定义必须不去访问形式参数，因为它们还未定义。从而，具有形式参数的每个进程定义，或者必须保证其形式参数在未赋值前不被访问，或者必须显式地含有在系统创建时进程实例数为零的规格(见图 D-3.8.4)。

```
PROCESS explicitly_created_proc_1 (0,...);
  FPAR ...
  ...
ENDPROCESS explicitly_created_proc_1;
```

图 D-3.8.4

具有形式参数规格的进程定义

在进程创建时，进程实例的值可通过下列预定义表达式确定：

OFFSPRING：返回本进程最近创建的实例的 PId 值。

SELF：返回进程实例本身的 PId 值。

PARENT：返回创建本实例的进程实例的 PId 值。

SENDER：返回发送最近一次被消耗的信号的进程的 PId 值。

当进程是作为系统创建的结果而创建时，只有表达式 SELF 返回一个 PId 值，而表达式 OFFSPRING 和 PARENT 则给出 NULL 值。

当同一进程类型具有较多的进程实例时，这些值非常重要，因为它们是使诸信号准确地访问这些实例的唯一途径。事实上，如 § D. 3. 8. 6中所述，当进程发送信号时，必须规定目的地实例，除非目的地实例可以明确地确定。

用户必须小心地保证所创建的进程实例在需要时能互相通信。为做到这一点，往往必须提供若干种初始化进程。这类进程在系统创建时被创建，它们必须会创建其它进程，并能够把适当的 PId 值通知所有需要知道这些 PId 值的进程。

有关进程创建的其它重要注意事项是：

1） 创建系统以后，进程只能由同一功能块内的另一进程来创建。允许在别的功能块里创建进程的一个方法，是让每个功能块里都有一个专用的进程，当此进程收到来自另一功能块的某个进程的一个信号时，就会创建一个进程。

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

创建进程和被创建进程之间的关系可在功能块图内用创建线符号来表示，如图D-3. 8. 5所示。

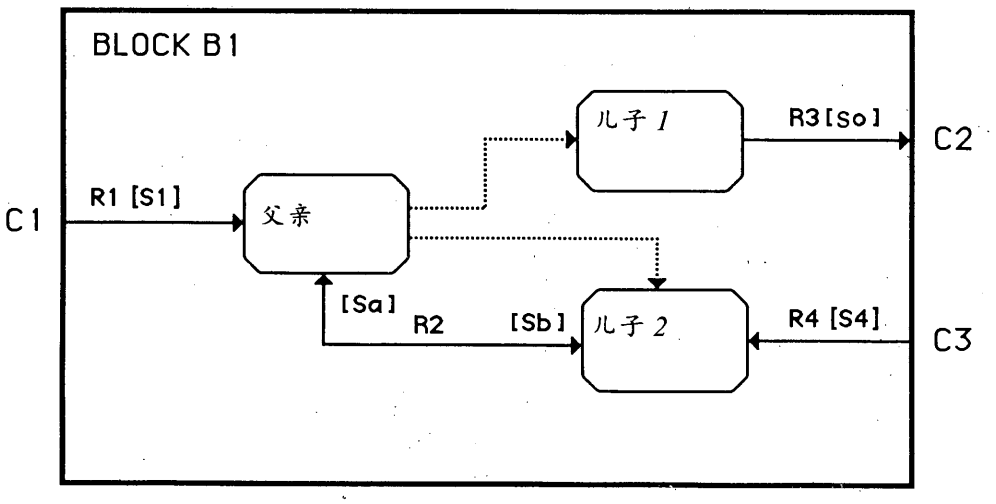


图 D-3. 8. 5

具有创建线符号的功能块图举例

D. 3. 8. 2　状态

状态是进程中的一个点，在该处没有动作正在执行，但要监视输入队列，看有没有外入信号到达。根据输入信号中给出的标识符，该信号的到达会使进程离开该状态而执行一特定的动作序列。一个已经到达且已引起一次跃迁的信号于是就被“消耗掉”而不再存在。

在 SDL/PR 中，状态是用关键字 STATE 后跟状态名字来表示的。此状态的规格或者在下一个状态语句开始处结束，或者在进程结尾处结束，或者用显式关键字 ENDSTATE 结束。在状态语句中可用一个星号（*）来代替状态名字；这是一个缩写符号，它表示后面所跟的输入或保存信号及相应的跃迁在所有的状态上都能够被解释。

关键字 NEXTSTATE 后面跟状态名字，用来指明下一个状态。在下一状态语句中可以用一个短划（dash）来代替状态名字，以表示下一状态和发出当前跃迁的状态是同一个状态。

图 D-3. 8. 6中有一个不完整的 SDL/PR 例子。

```
SYSTEM s;
  ...
  BLOCK b;
    ...
    PROCESS p;
      ...
      START;
        ...
      STATE st1;
        ...
      STATE st2;
        ...
        NEXTSTATE st1;
      ...
      ENDSTATE st2;
      STATE st3;
        ...
    ENDPROCESS p;
  ENDBLOCK b;
ENDSYSTEM s;
```

图 D-3. 8. 6

SDL/PR 中关于状态定义的不完整例子

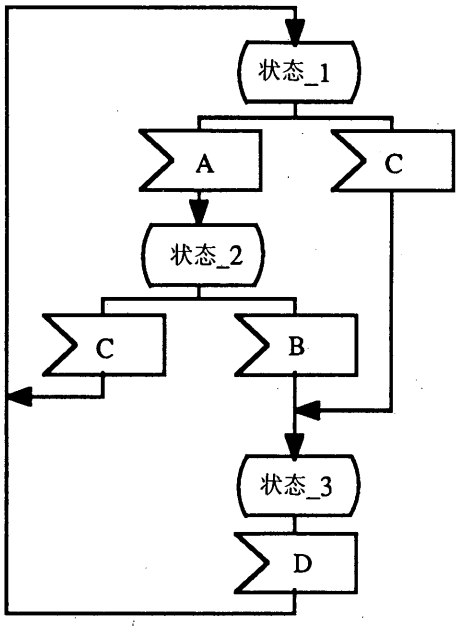


图 D-3. 8. 7a

一个状态只出现一次

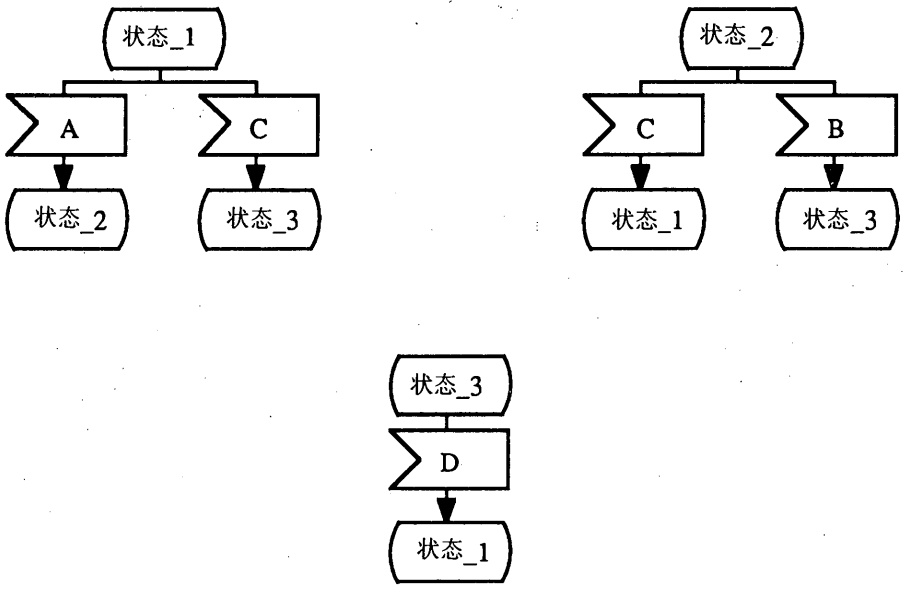


图 D-3. 8. 7b

将图 a）改画成具有几个主状态，
而把后继状态用作到这些状态的连接符

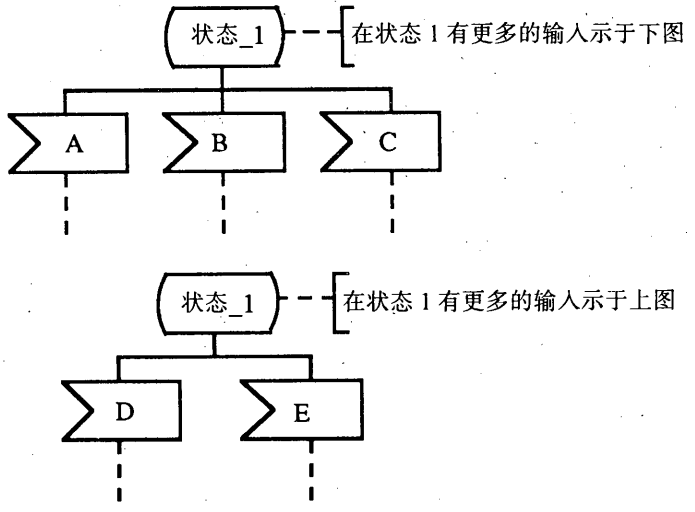
SDL 建议 - 附件D：用户指南 - § D. 3

在SDL/GR中，状态用状态符号表示。状态符号含有状态名字，并且接有输入符号或保存符号。具有一个外入箭头的状态符号可用来表示下一状态语句。

为了便于简化作图或有助于更好的理解，在一个SDL图中相同的状态可以出现若干次。在这种场合，可认为该图与合并同一状态的所有重复出现而得到的图是完全等效的。图D-3.8.7和图D-3.8.8给出了这样的例子。在图D-3.8.7b）中，当一个状态符号处在下一状态符号的地位时，被用作为连接符连接到具有同一名字的主状态。在图D—3.8.8中，一个状态用重复的几个符号表示，而每个符号仅具有输入（或保存）的一个子集。

在图D-3.8.7中，a）和b）逻辑上是等效的。图a）中每个状态只出现一次，而图b）中每个状态重复出现几次。图b）中各状态有一个主状态符号出现，在那里指明了所有与它有关的输入（和保存）符号。图中把能从其它地点到达该状态处（作为一个跃迁的终点）的状态都显示为一不带任何相关输入或保存的状态。关于某一状态符号的注解，在下一状态符号的注释中将会改善清晰度，特别当各个重复出现的状态符号出现在不同的页上的时候。

图D-3.8.8运用一个状态的重复出现来建立输入（和保存）的全集。图中示出该状态的每次出现仅附有其中的一个子集。



图D-3.8.8

当仅用一个状态符号不能清楚地画出
全部输入时，同一状态符号可以重复出现

在状态有较多的输入或保存时，使用这种方法是有效的，但如果读者未意识到同一状态有重复出现，就可能有对图形产生误解的危险。要避免这种误解，应当给只示出输入（或保存）的

子集的那些状态加上注释，指出具有有关输入（或保存）的同一状态的另外一些状态符号，如图 D-3. 8. 8所示。

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

在跃迁期间，进程并不明确地知道跃迁是哪个输入信号引起的，这只能从上下文关系来推断（即：这个跃迁只能在接收到某一特定信号时才出现）。在图 D-3. 8. 9中，只有收到 I1后才执行任务 T1，但若收到 I2或 I3，都将执行任务 T2，如对 T2来说知道收到的是哪个输入很重要，则按图 D-3. 8. 10所示来设计进程较好。

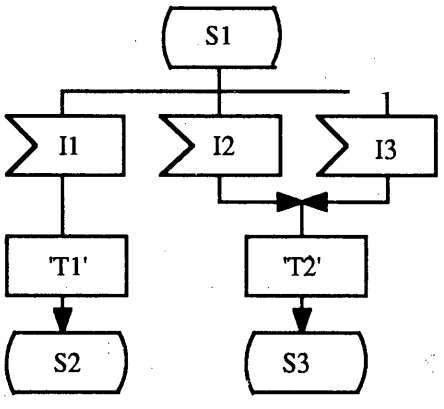


图 D-3. 8. 9

执行任务取决于三个收到的信号中的两个，但不取决于收到的是两个中的哪一个

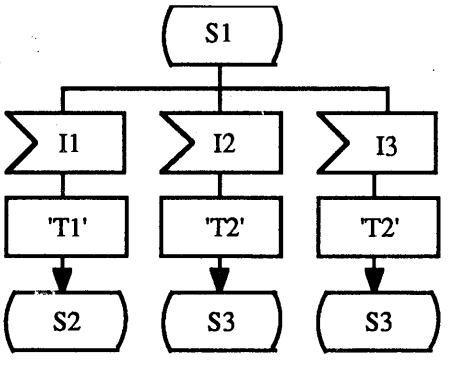


图 D-3. 8. 10

执行任务取决于所收到的信号

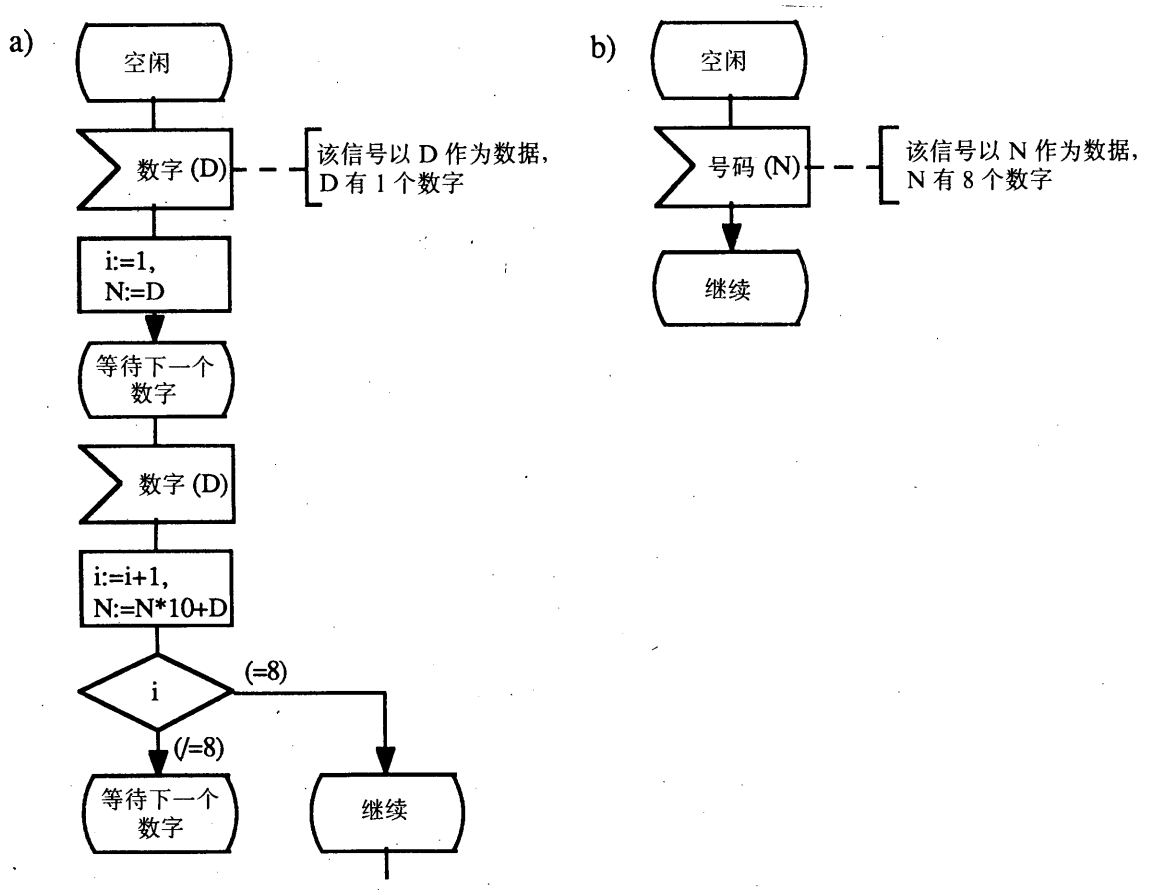
D. 3. 8. 2. 1　决定需要哪些状态

当定义一个进程的诸状态时，SDL 图的作者通常可有一些方法上的灵活性，他可能需要一

个能使他确定进程诸状态的策略，这个策略可以是非形式的，也可以是形式化的。为了使所产生的 SDL 图不至于不必要地过于复杂(确定过多的不同状态)，但也还要发挥 SDL 所固有的优点（不去人为地过分减少状态的数目），这就需要作出好的判断（通过实践取得）。在作者开始制图之前，必要的准备工作（在§D.3.2中已讨论过）必须已经完成，例如：

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

所有上述因素对决定每个进程的状态都有重大影响。在图 D-3.8.11中用两个例子来说明“选择”输入信号对 SDL 图中状态数目的影响。



注–例 a)和例 b)用不同的详细程度来表示相同的功能，结果是状态数目不同。

图 D-3.8.11

接收一个8位数字的电话号码

D.3.8.2.2 减少状态数目

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

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

D.3.8.2.3 分离公共功能

在规划 SDL 图时，可以理解：如果要定义一进程的某一特定而又重复出现的特性，就要求表示重复的状态。图 D-3.8.12示出了线路信号进程的 SDL 图的一部分，它说明了这样的要求，即线路信号音必须在一特定的时间长度内持续出现，才能认为已检测到线路信号。

为了对此作出规定，需要在状态“未检测到线路信号”和状态“通话”之间有一个中间状态。假设在一个完整的图中，这一公共功能在信号受检测的任一点都必须重复。另一个办法就是定义一个独立的进程，它负责在规定的识别时间内监视线路信号音并检测线路信号。有了这个新的进程，就能够把图 D-3.8.12中的图改画成图 D-3.8.13所示。（在一给定的上下文关系中，只要引进一个新的信号“有效线路信号”，即可使两个图等效。）

应注意图 D-3.8.12中所示的进程和图 D-3.8.13中的两个进程之间有细小的差别。

图 D-3.8.12中的进程一收到 T1就开始收费计时，而呼叫处理进程（图 D-3.8.13中的进程 b））要在收到有效线路信号后才开始计时，而有效线路信号是由图 D-3.8.13中的进程 a）在收到 T1后产生和发送的。这意味着在第二种情况下，计时开始要经过 T1加信号产生、发送和接收的时间。此外，在有效线路信号产生和发送这段时间里，还可能有其它信号已经排上了队。

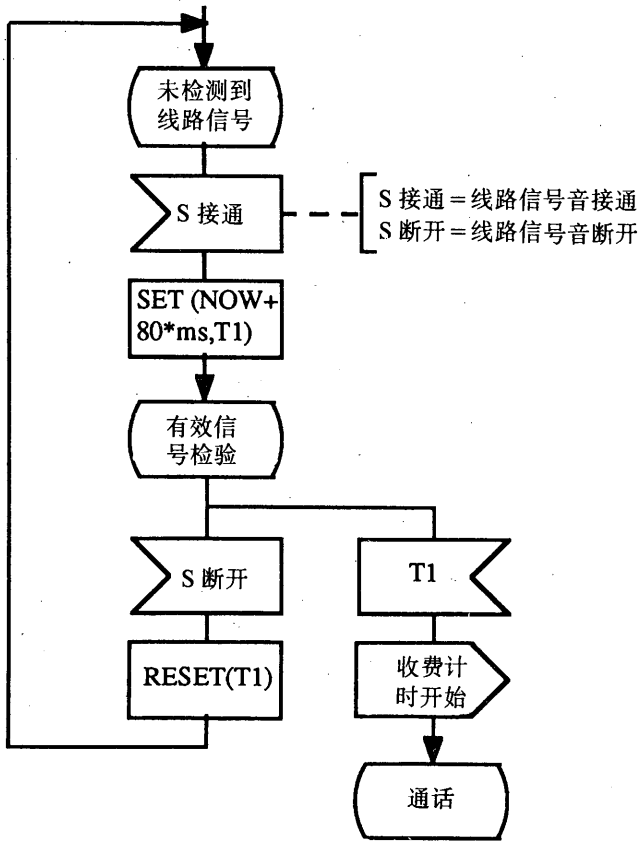
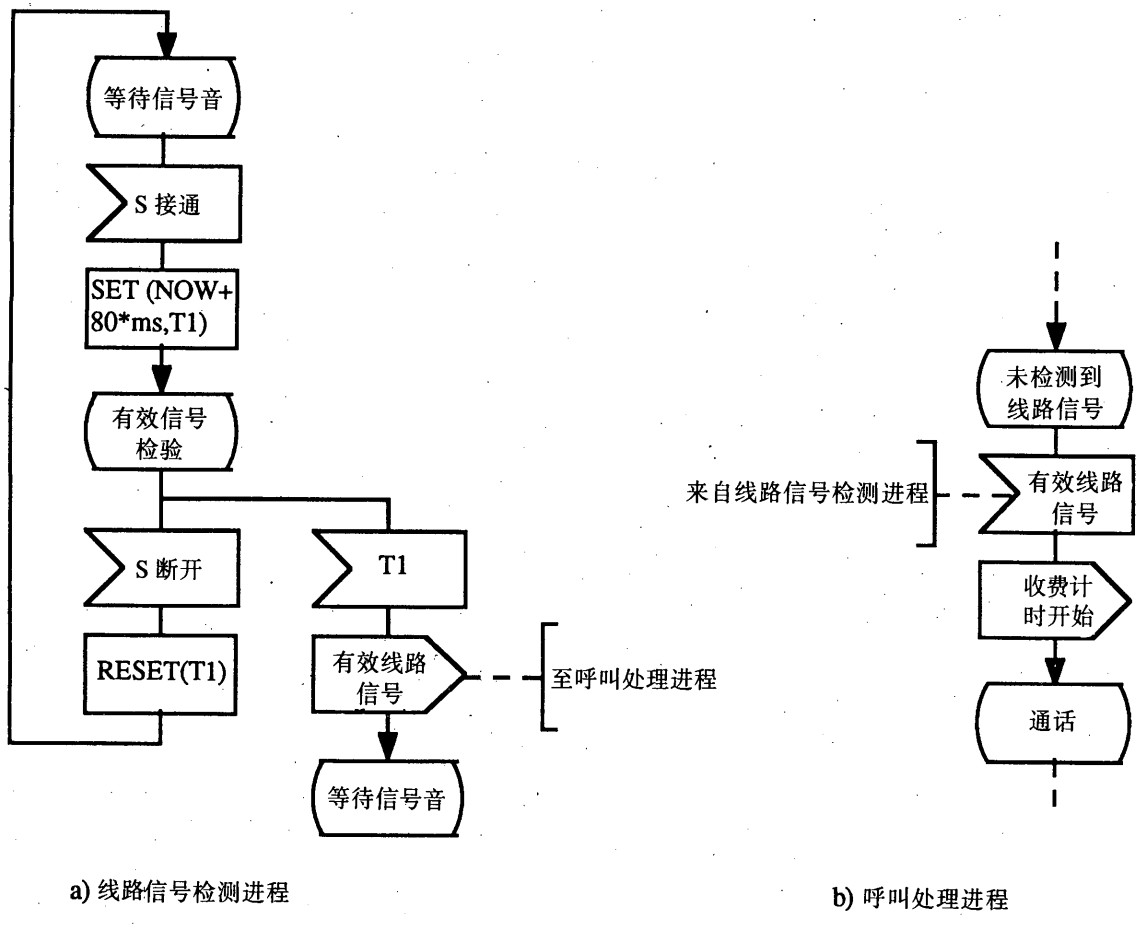


图 D-3. 8. 12

一个 SDL 图的例子：呼叫处理和线路信号检测组合图



a) 线路信号检测进程

b) 呼叫处理进程

图 D-3.8.13

分离一公共功能（线路信号检测）的例子，
以避免诸如“有效信号检验”这类重复状态

应注意的第二种情况是：如果被一子功能接收的信号必须也同样被主功能所接收，那么就不可能把此子功能分离开来，因为一个信号总是只通向一个进程。若在此例中，同一个信号“S 断开”必须在另一个状态中被接收，而这时它不需要 T1的证实时间，那么也不可能把此证实子功能分离到另一进程中去。

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

解决这个问题的另一种方法是使用服务概念，这将在§D. 5. 3中说明。

这个问题的一种不同解决办法是使用 MACRO 符号，如§D. 5. 1中所述。在这种情况下，我们可以得到比较简洁的图，而丝毫不变动原来的图的语义。此外，如果该进程的逻辑对 MACRO 有需要，则该 MACRO 可以被几个状态调用。

D. 3. 8. 2. 4　状态归并

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

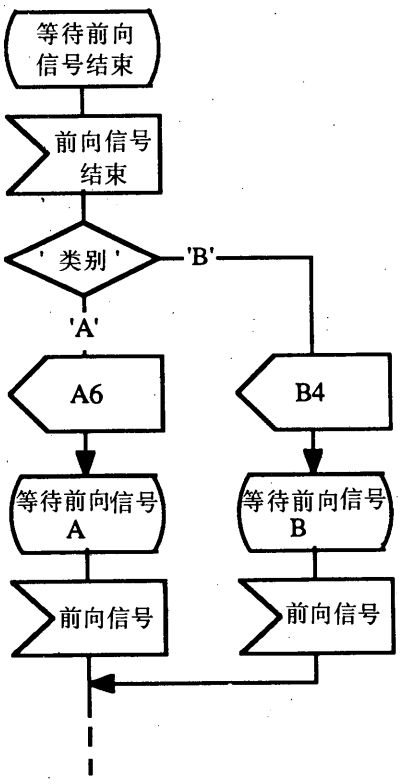


图 D-3. 8. 14

例：状态归并前 SDL 图的一部分

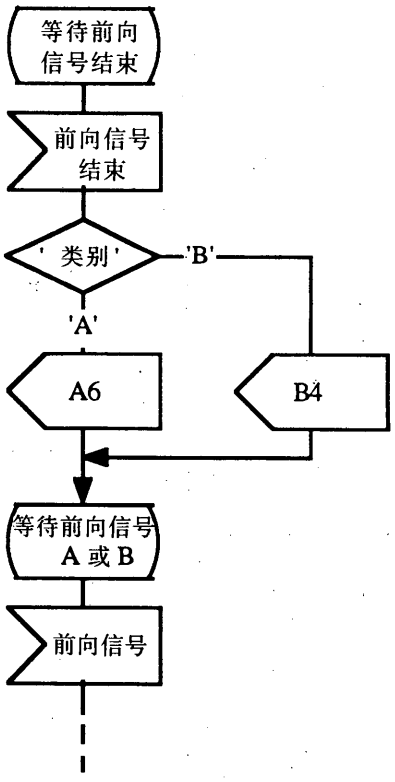


图 D-3. 8. 15

例：状态归并后的 SDL 图

图 D-3. 8. 14示出一 SDL 图的一部分，它具有两个状态，它们的历史不同，但未来却“完全一样”。在图 D-3. 8. 15中，这两个状态已合并成一个状态。这是一个相当平凡的例子，其复杂程度降低不多，但此技术可以用来获得显著的简化效果。SDL 语义不允许在某一状态之后跟有判定，来确定该状态以前该进程的历史（在此例中即：所送出的是 A6呢还是 B4?），除非这个

信息在进入该状态以前就已显式地存储起来。应注意：在一输入中命名数据就使得该值被存储起来。

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

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

D.3.8.3 输入

D.3.8.3小节说明输入的概念，以及在SDL图中不带保存概念的输入的用法。保存的概念以及SDL图中输入和保存合在一起的用法在§D.3.8.4中说明。

D.3.8.3.1 概述

连接到某一状态的一个输入符号表明：若在输入符号中给出信号名的信号到达时此进程正好在此状态上等待，那么就应执行跟在该输入信号后的跃迁。当一个信号已经触发了一个跃迁的执行，则该信号不再存在，我们说它已被消耗掉了。

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

在SDL/PR中，INPUT语句包含一信号标识符表，信号中所包含的值都是用变量标识符命名的，变量标识符必须具有在信号定义中指明的类别，因此它们的位置是很重要的。这些变量标识符括在圆括号内，并用逗号隔开（见图D-3.8.16）。如果丢弃一个或多个信号值，那么相应的变量也就失去了，这时用连续两个或多个逗号来表示（见图D-3.8.17）。

```
...
SIGNAL sig1 (INTEGER,BOOLEAN,INTEGER);
...
PROCESS ...
    ...
    DCL a INTEGER, b BOOLEAN, c INTEGER;  /*声明*/
    ...
    STATE st1;
        INPUT sig1(a,b,c);  /*一个正确的输入*/
        ...
    STATE st2;
        INPUT sig1(a,c,b);  /*一个不正确的输入*/
        ...
ENDPROCESS ...
```

图D-3.8.16
INPUT语句

INPUT a(var1,var2,,var4);

注–在这个语句中，丢弃了信号的第三个值

图 D-3. 8. 17

信号 a 有4个定义了的参数，这里仅输入 a 的3个值

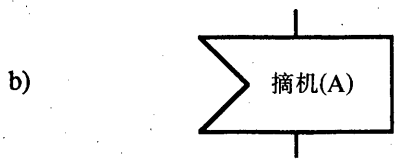
在 SDL/GR 中，输入由输入符号来表示。输入符号含有信号标识符表及与所传递的值相对应的变量标识符。为了使这些值能为进程所利用，它们必须在输入符号内命名，并括在括号内。

从输入中接收值的例子示于图 D-3. 8. 18、D-3. 8. 19及 D-3. 8. 20。

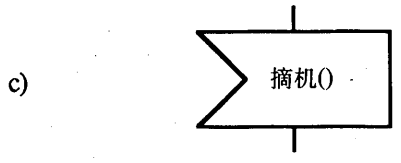
当该输入被机器解释时，接收进程就能利用已命名的数据。

图 D-. 8. 18示出“摘机”信号的接收。“摘机”信号具有相关联的值9269（用户号码）。这个信号可通过三种方法接收，如图中 a-c 所示。

注–值 9269 存储在名为″用户号码″的变量中



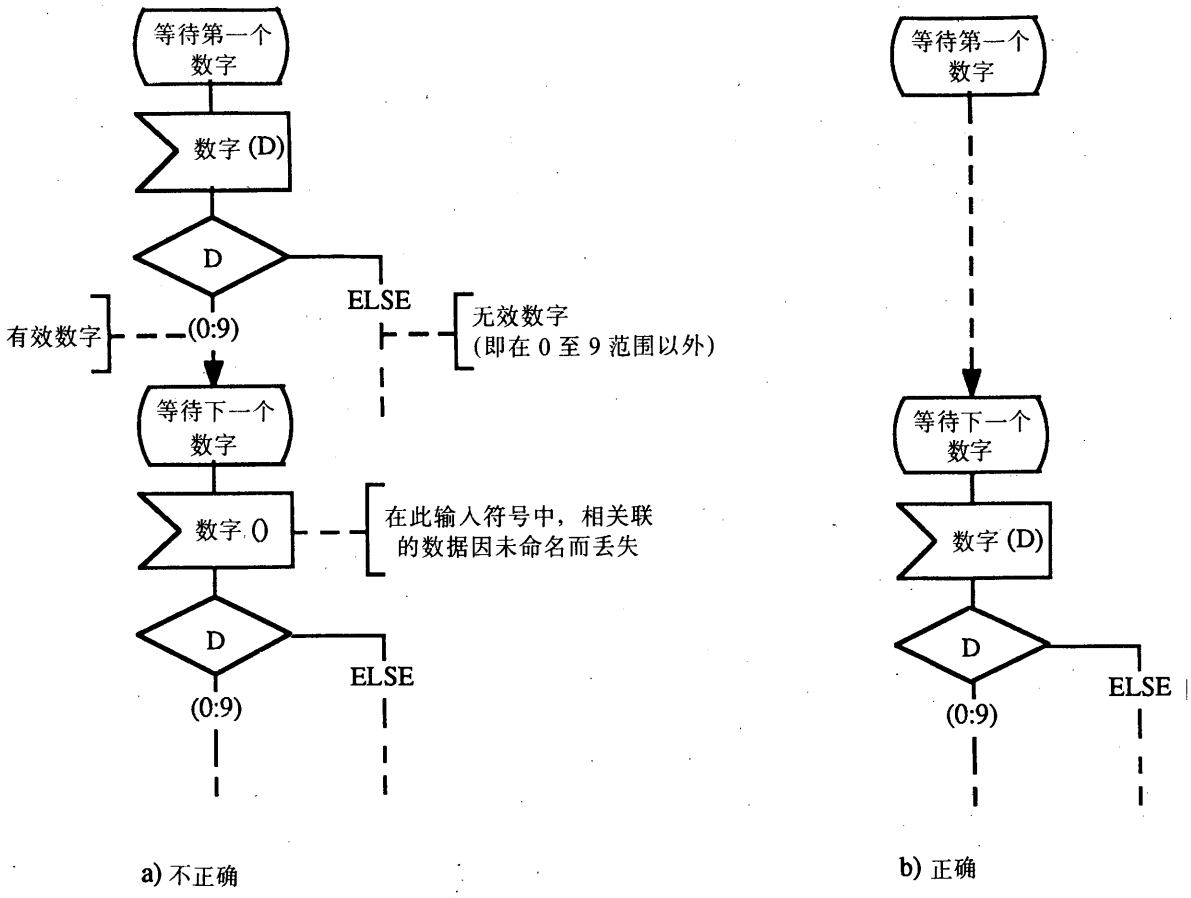
注–值 9269 存储在名为″A″的变量中



注1–没有数据名，故值9269丢失，不能为接收进程所利用。
注2–信号名（摘机）必须与输出的信号名一致，但数据名不一定相一致

图 D-3. 8. 18

一个进程接收数值的例子



注1–在a）中，第二个判定将使用得到的第一个信号作为D值。
注2–在b）中，D值将是当前数字的值，以前诸数字的值均被冲掉。

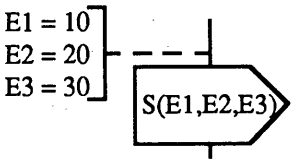
图 D-3. 8. 19
一个数字接收进程的一部分

图 D-3. 8. 20示出如何用一个信号来发送和接收几个值，每一项必须用逗号和它后面的项隔开。图 D-3. 8. 20c）示出如何通过在类别表中留空白的办法略去不需要的值。

应注意：在输出信号中我们可以写 E1、E2或 E3的表达式，但在输入信号中我们必须用变量来接收发来的值。

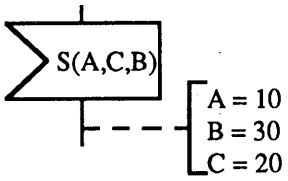
在 SDL 中，不需要画出输入符号来表示（因其到达而）起动零跃迁的信号（零跃迁即不含动作且导致返回到原来状态的跃迁）。习惯上，在某一状态上不用一显式输入符号表示的任何信号，必存在一个在该状态下的隐式输入符号和零跃迁。根据这个惯例，示于图 D-3. 8. 21的两个图在逻辑上是等效的，可任用其一。

a)



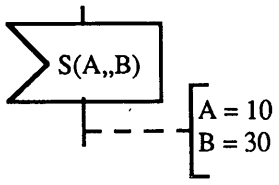
注–输出信号S有三个变量，称为E1、E2和E3。这些项有三个值，例如当前为10、20和30。

b)



注–在接收进程中，对应的输入信号S分别把这些项命名为A、C和B。

c)



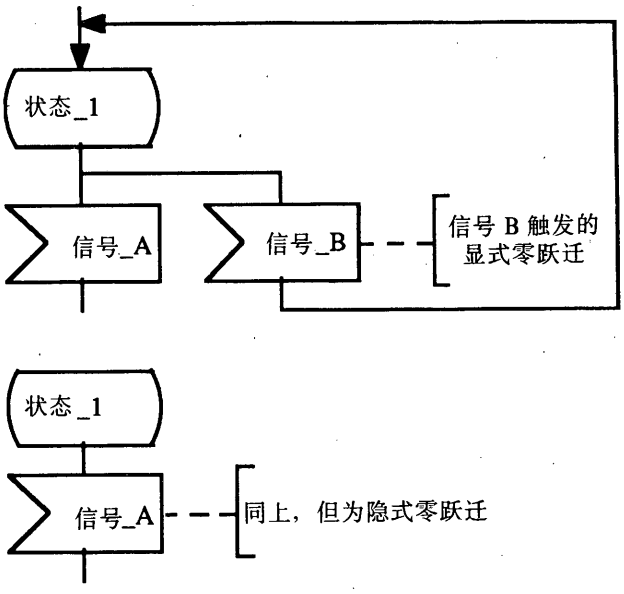
注–这个输入信号只命名两个变量，中间的值被丢失。

图 D-3. 8. 20

带有若干个值的信号

a) 显式“零”跃迁

b) 隐式“零”跃迁

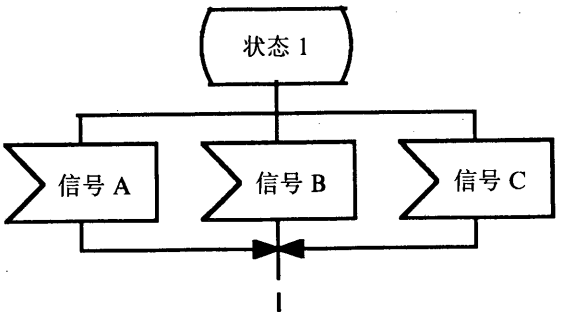


注–如信号 B 有数据关联，则在两种情况下数据均丢失。然而如在情况 a）中示出数据名（如：情况 a 中信号 B（x）），则数据值将保留。在这种情况下这两个例子不再完全等同。

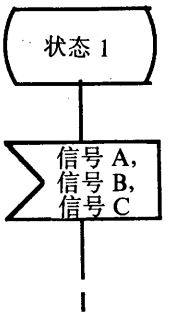
图 D-3. 8. 21

“零”跃迁的显式表示和隐式表示

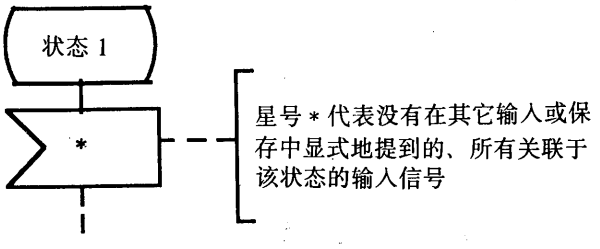
在多个输入导致同一跃迁的场合，所有有关的信号标识符均可置于一个输入符号内。在SDL 中提供了一个简化符号来表示所有信号（对该进程有效）的输入，而不显式地在该状态上提到。图 D-3. 8. 22说明了这一点，其中的几个图在逻辑上都是等效的。如在一个输入符号中提到所有的信号标识符，它们必须用逗号隔开。



a) 使用各自输入符号的多个输入信号举例



b) 使用一个输入符号的多个输入信号举例



c) 使用星号符号的多个输入信号举例

图 D-3. 8. 22

多个输入的各种表示法

D. 3. 8. 3. 2　隐式排队机制

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

图 D-3. 8. 23用这种队列概念来说明所画的 SDL 进程的操作，其中跃迁时间不为零。应注意：

— 没有用保存概念，因此诸信号按照它们到达的次序被消耗掉。
— 信号到达的顺序很重要。如果信号“C”在状态1和状态2间的跃迁期间先于“B”到达，则状态序列将成为1、2、3，而不是1、2、4。
— 因为当进程进到状态2和状态4时队列都不空，该进程在这两个状态上都不等待。
— 不可能为一信号指定优先权。为了进行服务间的互相通信，提供了一种专门的机制，使服务间的信号交换先于其它信号得到处理（见§D.5.3）。

若跃迁时间为零，则每一信号在其到达进程的当时就被消耗，除非采用保存操作（§D.3.8.4）。

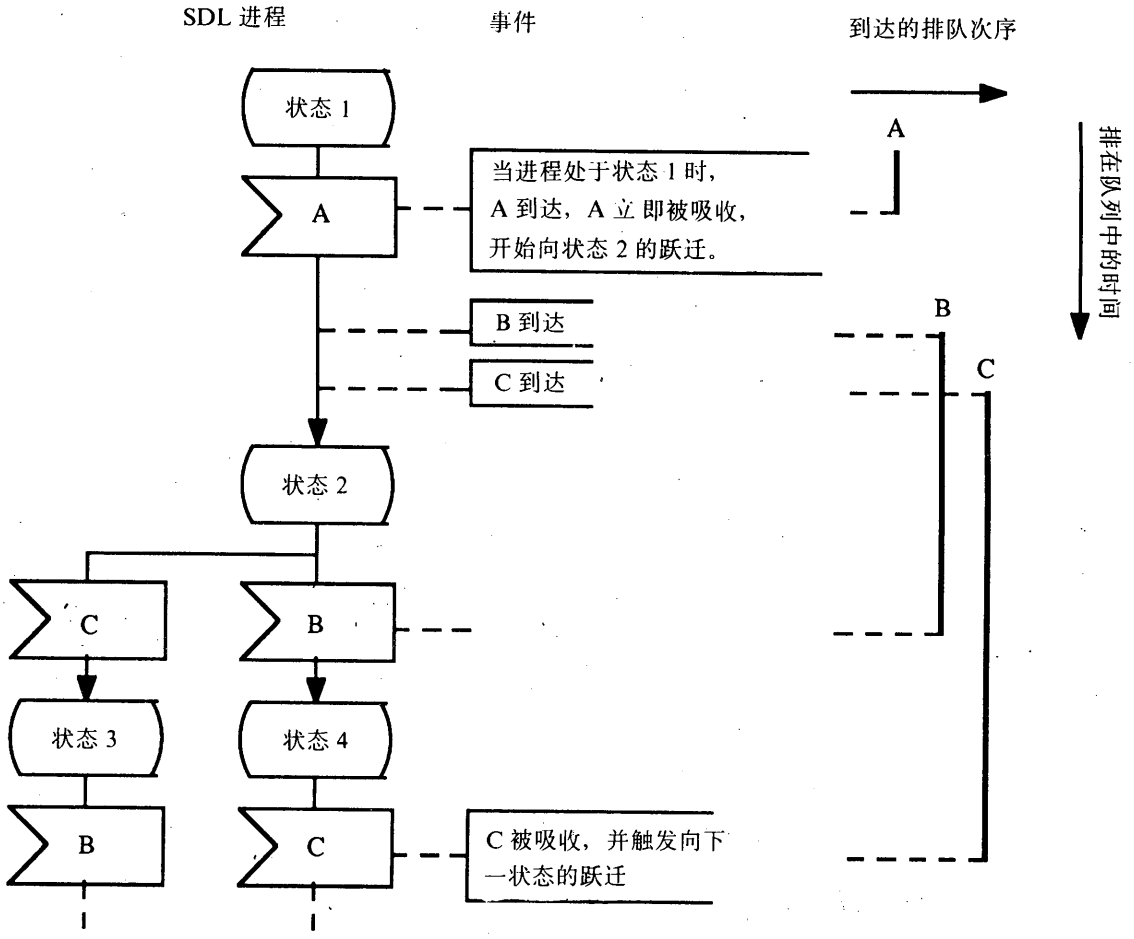


图 D-3.8.23
隐式排队机制的操作举例

D.3.8.3.3 非正常出现的信号的接收

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

另一种可能办法是采用一个状态的多次出现和“全部”符号（*）。例如，如果要使信号 A_ON_HOOK（A 挂机）能在所有状态上被接收并且产生完全相同的动作，则可采用图 D-3.8.24 中所示的方法。

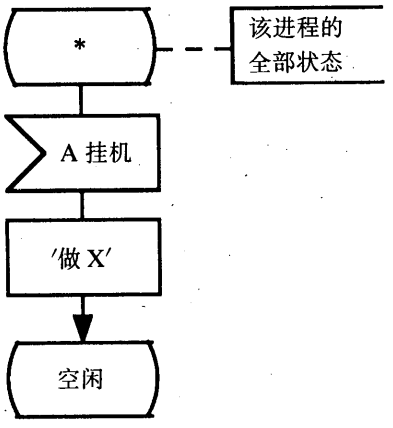


图 D-3.8.24

处理在几个状态上出现的信号的例子

D.3.8.3.4 信号的同时到达

在 SDL 建议的 §2中，曾提到信号能同时到达一个进程，这时可以任意安排它们的次序。

如果用户要设计一个能得到同时到达的信号的进程，就应小心不论到达的次序安排如何，都不应打乱该进程的预期操作。

SDL 不定义信号的优先权，即：信号的同时到达意味着随便选用其中之一。不过必须注意；用于服务间通信的信号总是首先被处理（见 § D.5.3）。

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

D.3.8.3.5 确定发送进程的标识值

每个信号带有其发送进程的进程实例值（PId）。当信号被消耗掉时，通过表达式 SENDER 可得到发送进程的 PId 值。图 D-3.8.25示出使用这种方法的一个例子。

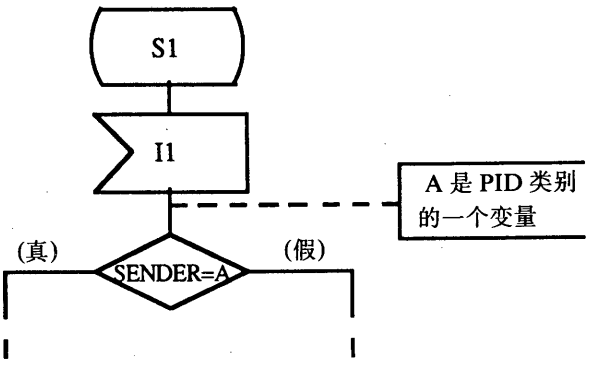


图 D-3.8.25

SENDER 表达式的用法

D.3.8.4 保存

保存的概念使得信号的消耗可以延迟，直到随后到达的一个或多个其它信号已被消耗为止。在§D.3.8.3中已讨论过：除非使用保存的概念，信号是按它们到达的先后次序消耗的。

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

在每个状态上，每个信号可按下列方式之一来处理：

— 把它表示为一个输入；

— 把它表示为一个保存；

— 包含在一隐式输入信号中，该信号可以导致一次隐式的零跃迁。

在§D.3.8.3中介绍的、隐式排队机制的操作，也可应用于保存的概念。信号一到达就进入队列，且当进程到达一个状态时，队列中的信号按它们到达的先后次序每次一个地受到检查。包含在显式或隐式输入中的信号被消耗了，并且相应的跃迁被执行。在保存中出现的信号则不被消耗，仍留在队列中原来的序列位置上，而考虑队列中的下一个信号。保存后面不跟跃迁。

在 SDL/PR 中，保存的构造用关键字 SAVE 并在后面跟一信号标识符表来表达。SAVE 语句的一个简单例子在图 D-3.8.26中给出。

```
...
STATE State_31;
      SAVE s;
      INPUT r;
      NEXTSTATE State_32;
STATE State_32;
      INPUT s;
      ...
```

图 D-3.8.26

使用保存的例子

在 SDL/GR 中，保存概念用一个含有信号标识符的保存符号来表示。

和输入的情况相类似，可以用一个“星号”来表示对所有未显式地在该状态中提到的信号（对该进程有效）的保存。

图 D-3.8.27示出一个带有保存符号的 SDL 进程例子。应注意信号 S 和 R 被消耗的次序是先 R 后 S—和它们被接收的次序相反。一个保存符号能对信号进行保存，只限于进程处于（保存符号所在的）这个状态期间，并且保存到跃迁到下一状态为止。在下一状态，该信号将通过一显式的或隐式的输入被消耗掉（如图 D-3.8.27所示），除非标有该信号名的保存符号再次出现，或者在该隐式队列中，在它之前碰巧有另一个保存的信号要被消耗（如图 D-3.8.28所示）。

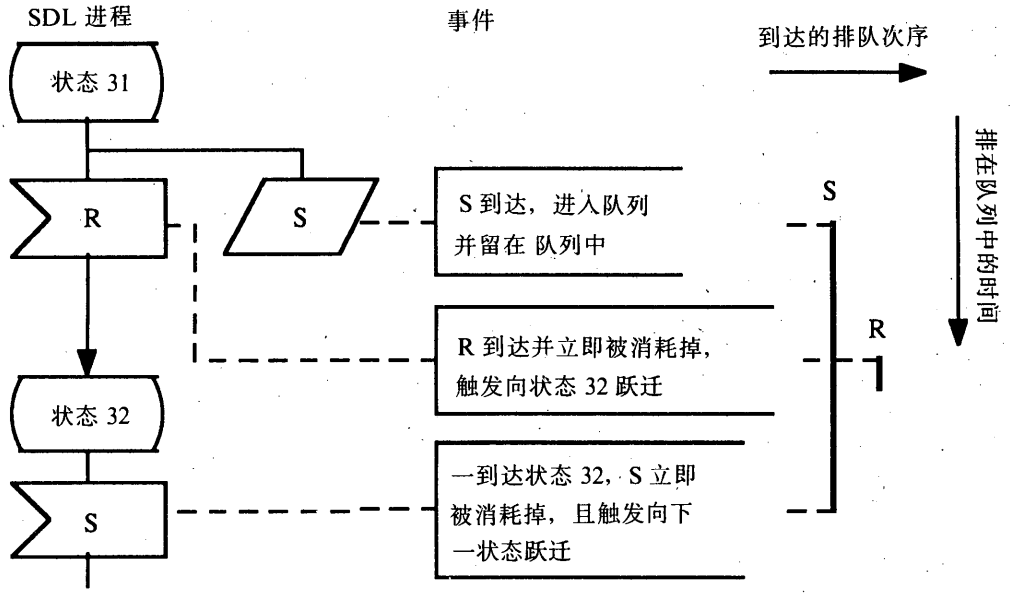


图 D-3.8.27

一个带有保存符号的 SDL 图例子，示出隐式排队机制的操作

保存信号只有通过相应的输入符号（显式的或隐式的）才能被进程所利用。特别是在它通过输入被消耗掉之前，在判定中不能提出关于它的问题，也不能使用它所关联的值。

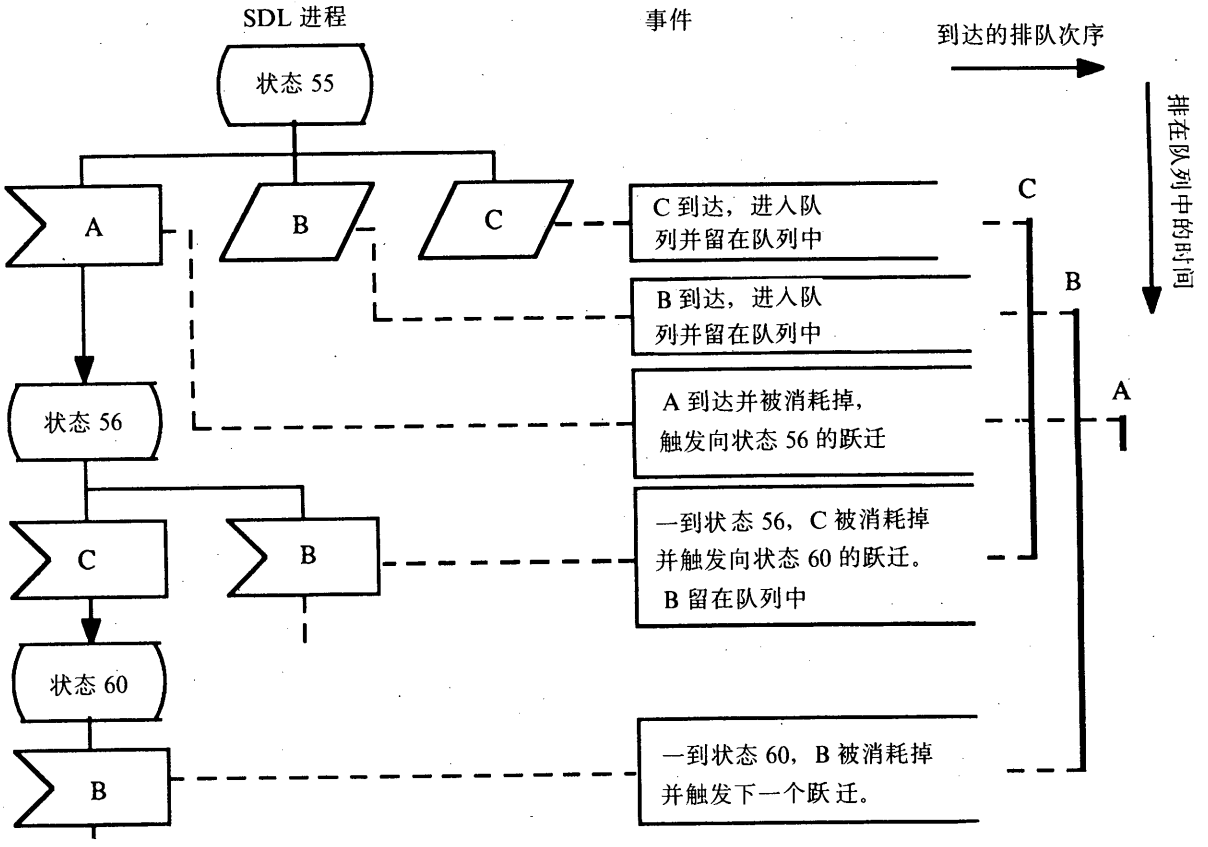


图 D-3. 8. 28

使用保存符号的第二个例子

在某个状态上如果要保存多于一个的信号，可以给每个信号一个保存符号，或者把所有信号都写在一个保存符号中。如要保存多个信号，保存符号的语义隐含了要保留它们到达的先后次序。

在图 D-3. 8. 29中给出了使用保存的第三个例子，在图 D-3. 8. 30中描述了其概念性的排队机制的操作过程。

保存能用来使图简化。例如，通过对一个信号进行保存，就可以避免接收它但又不得不存储其关联值直到下一个状态。

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

如果保存使用得不小心，保存的信号队列会变得很长，或者会使保留信号的时间拖得太长，以致于在需要吸收一个新信号的地方却去吸收了一个旧的信号。

SDL 对队列的长度不作限制，这样会引起实现方面的问题。

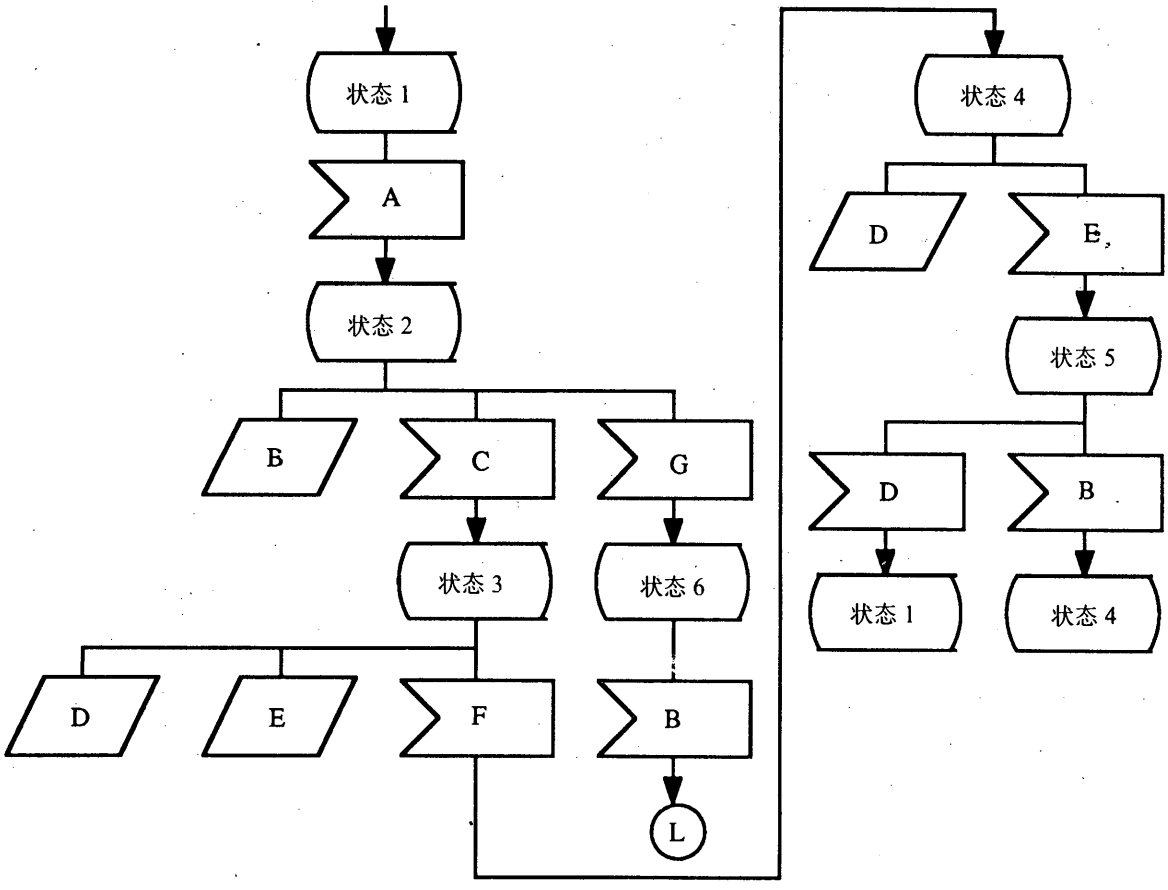


图 D-3. 8. 29

一个较复杂的 SDL 图的例子

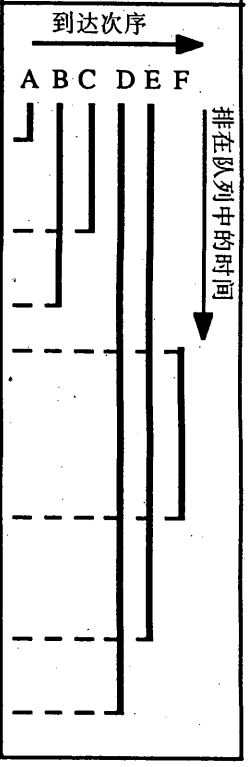
当前状态	事件	排队
状态 1	(进程到达状态 1，队列中有信号 A、B、C、D、E) 队列中的第一个信号 A 被消耗掉，触发向状态 2 跃迁。	
状态 2	队列中的第一个信号 B 出现在保存符号中，故留在队列里。	
状态 2	第二个信号 C 被消耗掉（显式输入），触发向状态 3 跃迁。	
状态 3	队列中的第一个信号 B 被消耗掉（隐式输入）。	
	信号 F 到达并进入队列。	
状态 3	(在再次到达状态 3 时) 队列中的第一个信号 D 出现在一个保存符号中，故留在队列里。	
状态 3	第二个信号 E 出现在一个保存符号中，故留在队列里。	
状态 3	第三个信号 F 被消耗掉（显式输入），触发向状态 4 跃迁。	
状态 4	队列中的第一个信号 D 出现在保存符号中，故留在队列里。	
状态 4	第二个信号 E 被消耗掉（显式输入），触发向状态 5 跃迁。	
状态 5	队列中的第一个(也是唯一的一 个)信号 D 被消耗掉（显式输入），触发向状态 1 跃迁。	

图 D-3. 8. 30

排队机制的操作过程

D. 3. 8. 5 允许条件和连续信号

允许条件根据规定的允许条件有条件地接收信号。若条件为真则信号被吸收，且跃迁被执行；若条件为假，则信号被保存，进程仍停留在原来状态，直到另一信号到达或条件由假变成真为止。这可由图 D-3. 8. 31的例子来说明：当进程 P1进入状态 S1时，就对条件（即 IMPORT (X，P2）是否等于10）进行检查，若条件为真，则信号 B 可以被接收，否则，信号 B 就被保存起来。在本例中，A 到达并引起向 S2的跃迁。在该跃迁期间 X 变为11，而 P2输出其新的值；因此现在，在状态 S2，附于信号 B 的条件为真。由于信号 B 是队列的第一个信号，故执行它所引起的跃迁，而进程在状态 S3结束。

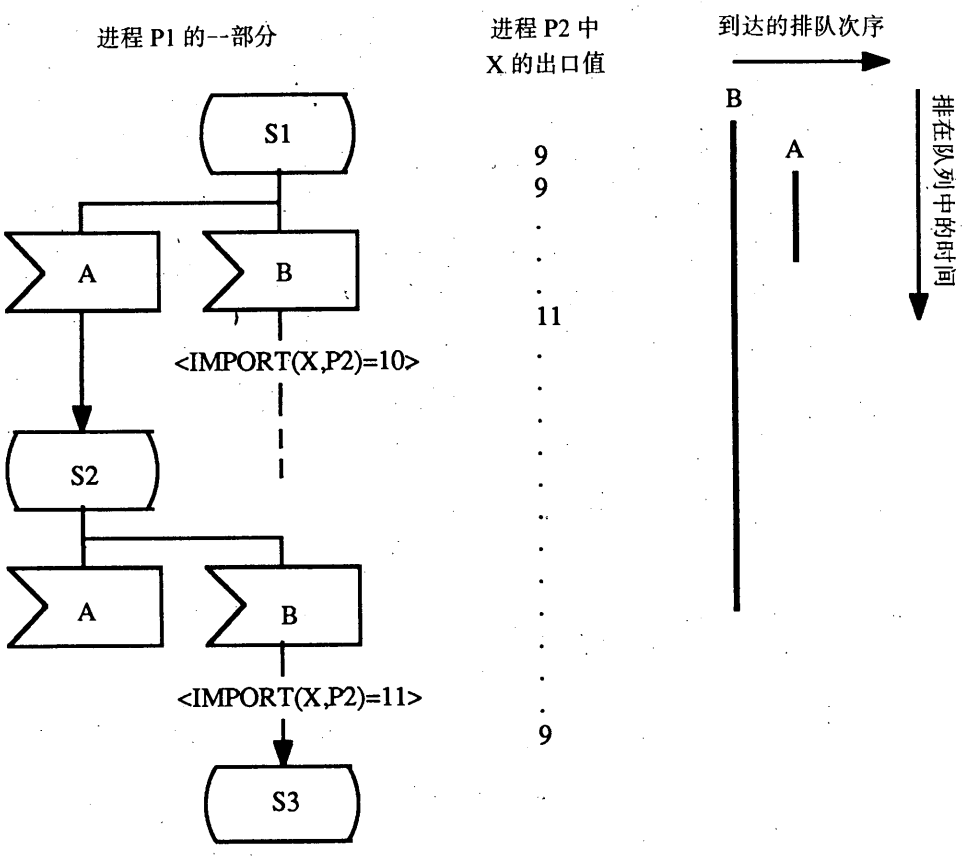


图 D-3. 8. 31

允许条件

允许条件的一些重要属性是：

1) 当进程到达一个状态时，就检查允许条件，并在之后进程停留在该状态期间连续不断地监视允许条件。因此，如果在收到 A 后执行跃迁时，X 的输出值已从9变到11再变到12，则进程将仍停留在状态 S2。

2) 允许条件的基础可以是局部变量和（或）可包含在一个表达式中的任何语言构件（例如 IMPORT、VIEW、NOW）。

3) 虽然对每个状态可能使用一个以上的允许条件，但对同一个信号却不允许用多于一个的允许条件。因此，图 D-3. 8. 32中所示的条件是不许可的。若对一给定信号要求有多个条件，则它们应能组合成一个布尔表达式，如图 D-3. 8. 33所示。

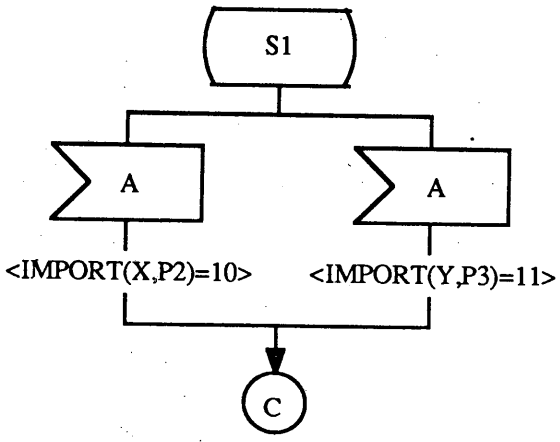


图 D-3. 8. 32

错误的允许条件

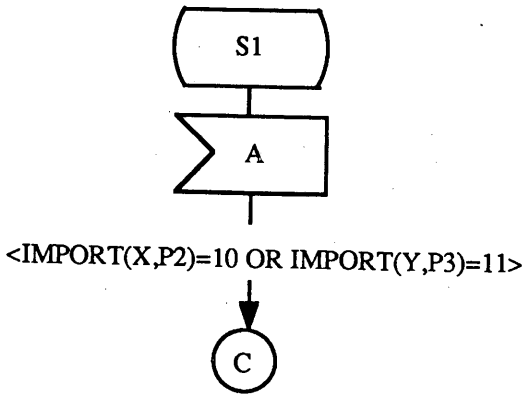


图 D-3. 8. 33

图 D-3. 8. 32的正确解答

允许条件可被检查多次，并可按任何次序进行检查。所以用户必须小心，看表达式相互间是否有副作用（例如把 IMPORT 和 SENDER 相组合）。

还必须注意：在允许条件中规定的信号不能影响该条件的布尔值，因为在信号被吸收之前，它所传递的值尚未指定。例如语句：

```
INPUT x (A)PROVIDED (A=5); …
INPUT y     PROVIDED (SENDER=pid1);
```

这一语句是令人费解的，因为条件中 A 和 SENDER 的值相当于信号被吸收之前的情况。

连续信号的基本性质和允许条件相同，只是它不附带信号。因此，当它进入状态而在队列

中没有能够引起跃迁的信号时，这些连续信号将受到检查，如果有一个信号为真，就执行跟随它的跃迁，这可由图 D-3. 8. 34的例子来说明。起初进程处于状态 S1，X 的出口值为9，当信号 A 到达时，它引起向 S2的跃迁；在跃迁期间，X 变到11且出口其新的值，由于队列中没有其它信号，所以执行向 S3的跃迁。

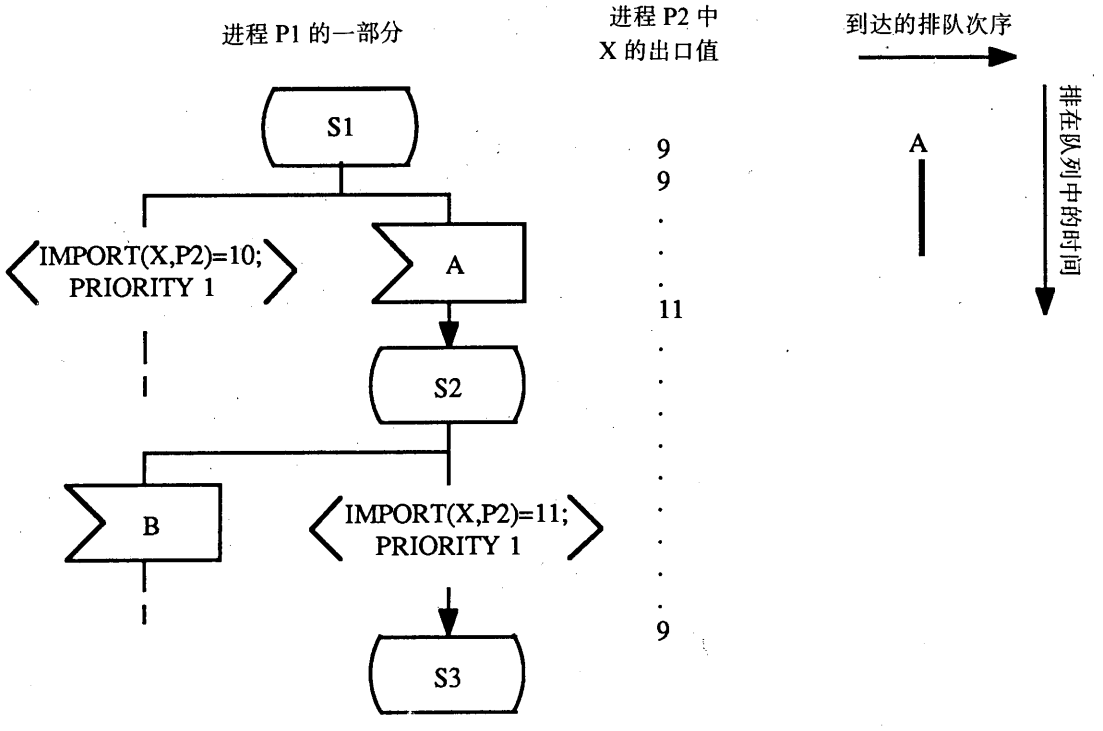


图 D-3. 8. 34
连续信号

连续信号的一些重要属性是：

1） 和允许条件一样，条件的值只在进程到达这个状态或处于该状态中时才受检查；

2） 每个状态允许有多个连续信号。当一状态附有多个连续信号时，优先权较高（优先数最低）的连续信号最先受检查。不能有两个连续信号具有相同的优先数。在所有情况下，连续信号的优先权低于任何其它信号，这又是由 SDL 的基础系统所引起的；不过在 SDL 中，是用出口变量时所发送的信号来作为连续信号，这一方法允许连续信

号采用优先权，而且事实上当出现多个连续信号时，为了避免混乱也需要有优先权。这在图 D-3.8.35中作了说明。起初进程处于状态 S1，其进口表达式给出 X 的值为10，Y 的值为11，由于两个连续信号皆为真，具有较高优先权（较小的优先数）者被选中，从而执行向 S2的跃迁；在 S2上，由于 Y 的条件不再为真，所以即使 X 的连续信号优先权低于 Y 的，但仍执行它后面的跃迁，而进程到达状态 S3。

3） 当执行连续信号之后的跃迁时，表达式 SENDER 的返回值与 SELF 的返回值相同。

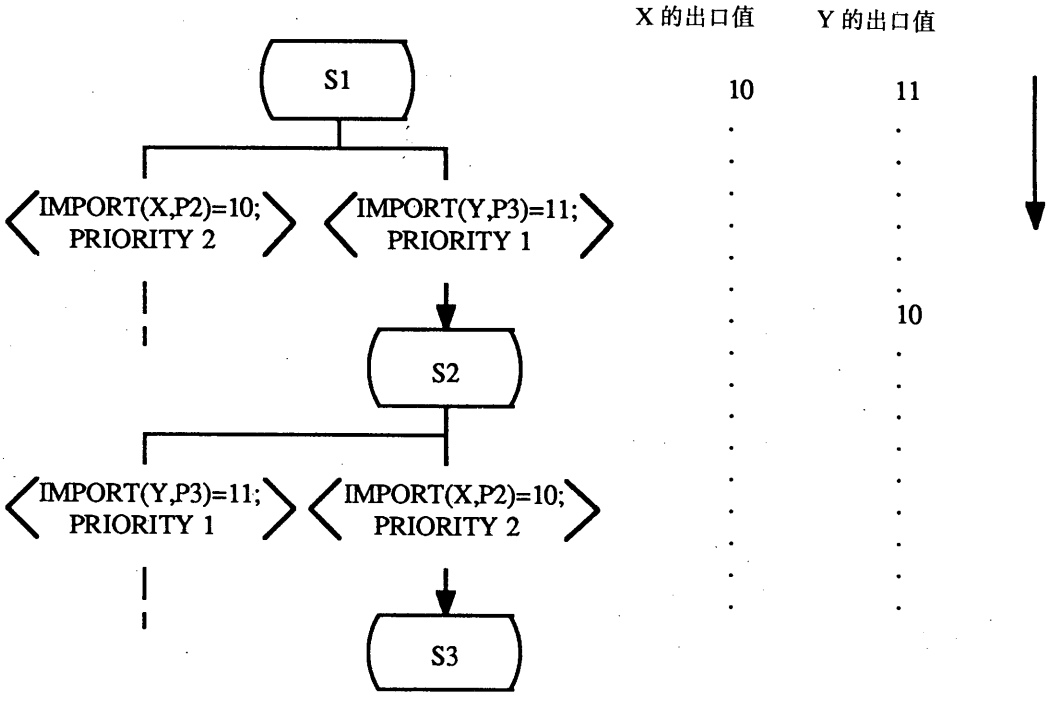


图 D-3.8.35
带优先权的连续信号

D.3.8.6 输出

输出就是将一个信号从一个进程发送到另一进程（或发送到本进程）。由于对信号消耗的控制与接收进程（见§D.3.8.3）有关，且与输出直接有关的语义比较简单，所以从输出进程的观点看来，一个输出常可被看作一次瞬时动作，动作一旦完成，就对该发送进程不再进一步产生直接影响，该进程也不直接地知道该信号的命运。

输出动作表示对一信号及其关联值（如有的话）进行发送。将一个值关联于一输出信号的方法是：将该值放在括号内，或将具有该值的表达式放在括号内（见图 D-3. 8. 37）。

在 SDL/PR 中，输出动作用关键字 OUTPUT 后面跟一信号标识符表来表示，每一个信号标识符可以带一个括在圆括号内的实在参数表。若在输出中没有与信号定义中某一类别相应的实在参数，则在进行接收的输入中，相应的变量将具有值“未定义”。

输出语句中的目的进程实例必须通过关键字 TO 后面跟一进程实例表达式来表述。若从上下文来看该目的进程实例能够唯一地予以确定，则 TO 子句可以省略。在输出语句中还可附加一种寻址条件，办法是在关键字 VIA 后面跟一个信号路由或信道标识符表。

图 D-3. 8. 36示出一些正确的输出语句例子：

```
OUTPUT sig1(2,true,10);
OUTPUT sig2, sig3(par1,par2) TO process_a;
OUTPUT sig4 TO process_b VIA channel_x,channel_y;
OUTPUT sig5 VIA signal_route_z;
```

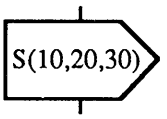
图 D-3. 8. 36

输出语句举例

在 SDL/GR 中，输出由输出符号来表示，其中含有信号、实在参数，以及任选的目的地址和/或 VIA 构造等规定。

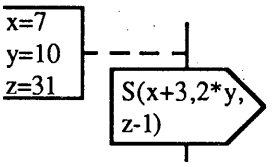
每一输出必须指向一具体的进程实例。由于在产生一个规格之初通常不可能知道任何进程的进程实例，信号寻址的正常方法是在关键字 TO 中采用一个变量或表达式，图 D-3. 8. 38、D-3. 8. 39和 D-3. 8. 40给出了几个例子。图 D-3. 8. 38中，在进程创建时把一进程实例的值赋给进程参数 out-to，然后在该进程内部把 out-to 作为这个进程和所连进程之间的链接。在进行系统设计时，应小心地保证由 out-to 指示的进程类型能接收所发送的信号。在图 D-3. 8. 39中，用内部的表达式 SENDER 来把一个信号送回给发送该信号的那个进程。在图 D-3. 8. 40中，把信号发送给本进程最近创建的子孙进程。

a)



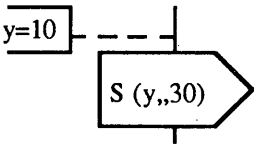
注–信号S带有三个关联值：10、20和30。

b)



注–在解释该输出时，x、y和z（在本例中）分别具有7、10、31。该输出发送的值为10、20、30。

c)



注–在解释该输出时，y（在本例中）具有值10，该输出发送的值为10、一个未定义值和30。

图 D-3. 8. 37

带有关联数据的输出信号

```
PROCESS X (0,5);
    FPAR (out_to PID);
    . . .
    OUTPUT sig TO out_to;
    . . .
```

图 D-3. 8. 38

用形式参数指出信号发往的地址

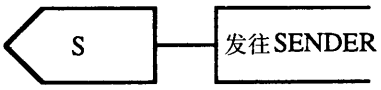


图 D-3. 8. 39

把信号发送回 SENDER

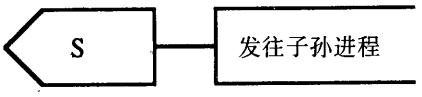


图 D-3. 8. 40

把信号发送到一个子孙进程

D. 3. 8. 7 任务

任务用来在跃迁中表示对变量的操作，或通过非形式正文来表示特定的操作。

在 SDL/PR 中，任务用关键字 TASK 后面跟一序列语句或用非形式正文来表示，各语句或正文彼此用逗号隔开，由分号终结。任务中的语句可以只是一个赋值语句，而非形式正文则由一用单引号括起来的短语组成。图 D-3. 8. 41中给出一些 SDL/PR 的有效的任务的例子。

```
TASK a:=b;

TASK '连接主叫用户和被叫用户';

TASK c:=d+e;

TASK var1:=var2*var3,
     var4:=var5 MOD var6;
```

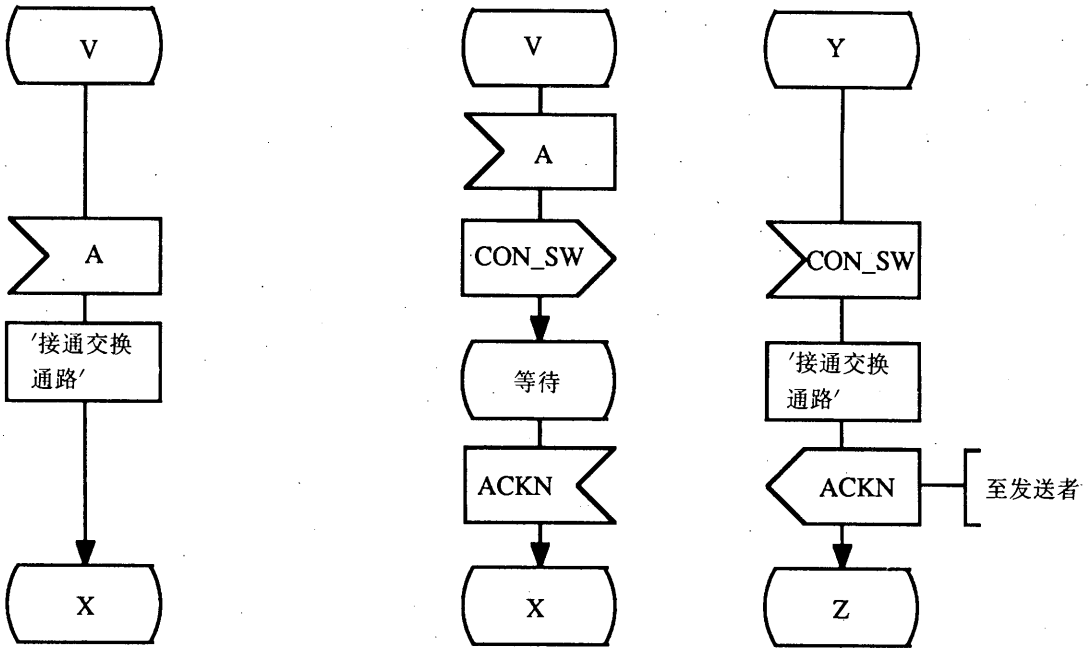
图 D-3. 8. 41

任务的例子

在 SDL/GR 中，任务由一个含有序列语句或非形式正文的任务符号组成。

SDL 用户们在决定用一个任务还是用一个独立的进程来表示所定义的系统的某些方面

时，有时会感到困难。考虑图 D-3. 8. 42中所示的进程，“接通交换通路”应表示为一个任务呢还是表示为一个独立的进程呢?如果尚未确定一个单独的交换通路控制进程，则任务符号将是适宜的（见图 D-3. 8. 42a)）；若已确定一个单独的交换通路控制进程，则必须使用与该控制进程通信的信号（见图 D-3. 8. 42b)）。



a）用一个进程的解　　b）用两个进程的解

图 D-3. 8. 42
关于“接通交换通路”的两个解

D. 3. 8. 8　判定

判定是在跃迁内部的动作，它提出一个问题，涉及在执行该动作的瞬间一个表达式的值。根据对问题的回答，进程沿着判定后面所跟的两条或多条路径中的一条前进。SDL 图的作者们应保证诸进程是这样确定的：即它们不会去执行得不到回答的判定。这种判定将使该图无效，并会引起相当大的混乱。

判定的问题可以是一个表达式或一段非形式正文。判定的回答可以是一个或多个从问题的表达式得出的值，或者是一段或多段非形式正文。如果问题或诸回答中的一个回答是非形式的，那么整个判定就是非形式的。不同的回答彼此间用逗号隔开。这些值可由多个常量表达式来表示，或由带有一运算符作为前缀的一些常量表达式来表示，或者由一些范围来表示，这些范围的上限和下限均为常量表达式。

回答的值必须与问题中所含的表达式属于同一类别。

可以显式地指明若干回答，而把所有可能的其它回答用关键字 ELSE 组合在一起。

在 SDL/PR 中，判定的表示是用关键字 DECISION 开头，接着规定问题，然后是多个可能的回答，每个回答与相应的跃迁有关，回答都在圆括号内指明。最后用关键字 ENDDECISION 来结束这组跃迁（见图 D-3. 8. 43）。

```
DECISION question ;
    (answer_a): . . .
    (answer_b): . . .
    (answer_c): . . .
    ELSE: . . .
ENDDECISION;
```

图 D-3. 8. 43

判定的框架

判定的一些例子示于图 D-3. 8. 44。

```
. . .
DCL x INTEGER,a BOOLEAN;
. . .
DECISION x;
    (2): . . . ;
    (2+5): . . . ;
    (6+8,10+9): . . . ;
    (=3): . . . ;
    (20:30): . . . ;
    (>=100): . . . ;
    ELSE: . . . ;
ENDDECISION;
. . .
DECISION a;
    (TRUE) : NEXTSTATE s1;
    (FALSE) : NEXTSTATE s2;
ENDDECISION;
. . .
DECISION '用户类别';
    ( '国际', '国内' ): . . .
    ( '本地' ): . . .
ENDDECISION;
. . .
```

图 D-3. 8. 44

SDL/PR 中判定的一些例子

全部跃迁在关键字 ENDDECISION 处结束。有的跃迁在结尾处不是一个终止语句（即 JOIN，NEXTSTATE，STOP），这些跃迁将继续执行在 ENDDECISION 后面的语句，如图 D-3. 8. 45中两种等效的分支所示。

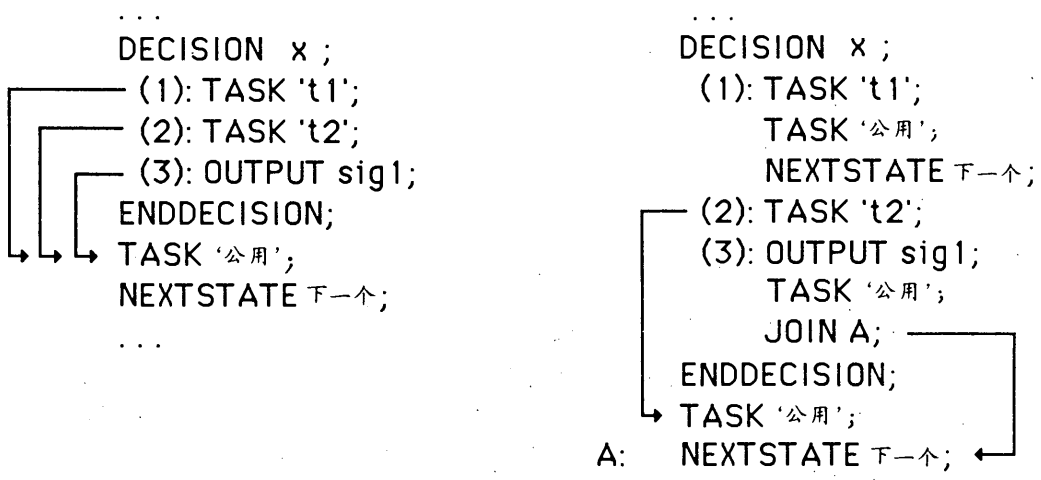


图 D-3. 8. 45

来自判定的两种等效的转移

判定语句也可以用来模拟 IF-THEN 结构、DO-WHILE 结构及 LOOP-UNTIL 结构，就像在结构化编程中那样。

在 SDL/GR 中，判定由一个内部含有问题正文的判定符号表示。判定符号必须有两个或两个以上的分支与同样数目的相应的回答相关联，每个回答必须安排在相应分支的右侧或顶部，或者也可写在分支中间而中断流线。在 SDL/GR 中，回答是否要加括号是任选的，但为了避免误解，建议加括号。

SDL/GR 中的一些判定例子示于图 D-3. 8. 46。

如果一个回答产生的跃迁返回到该判定，则中途必须执行一些动作来影响判定中的问题。然而，即使用了这条规则，还可能形成无穷循环，如图 D-3. 8. 47所示。因此当有的回答产生的跃迁返回到原判定时，总要十分谨慎。

可使用进程当前可用的任何值来作出判定，包括：

— 通过一次输入接收到的数值；

— 进程创建时作为实在参数传递给进程的值；

— 共享值。

问题中的表达式可包括常量和任何上述各类的值。

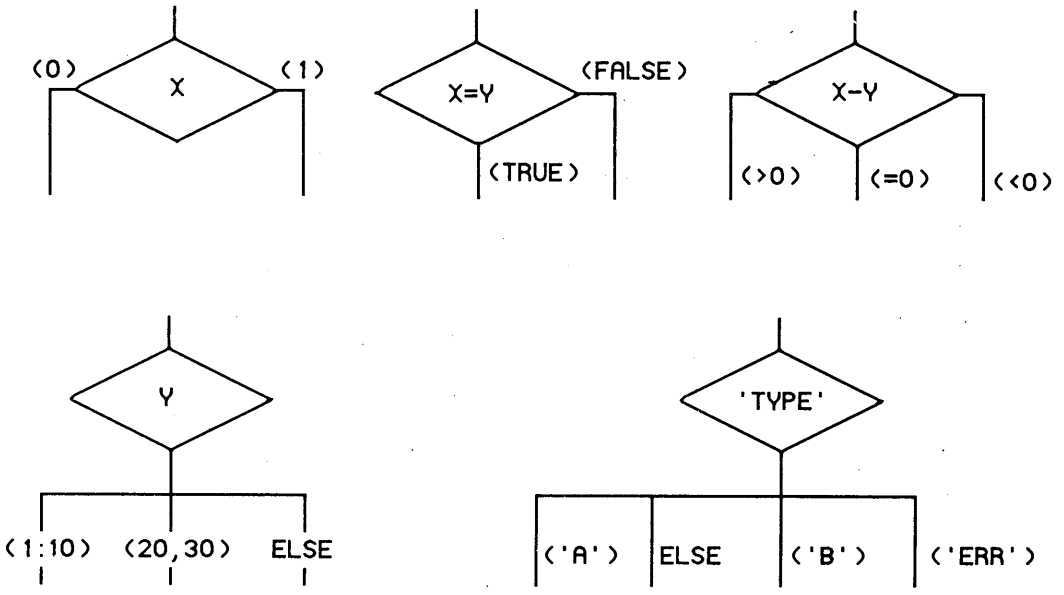


图 D-3. 8. 46

SDL/GR 中一些判定的例子

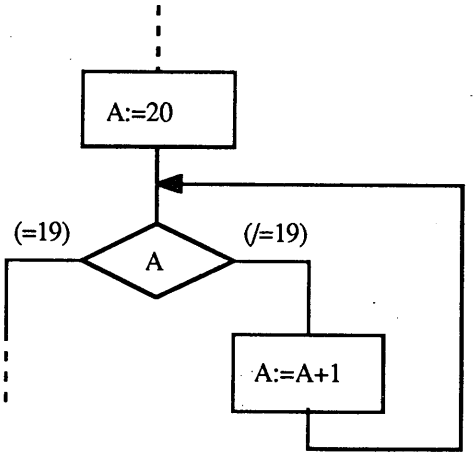


图 D-3. 8. 47

合法使用判定的例子，该判定形成了无穷循环

D.3.8.9 汇接和连接符

汇接可使控制从一个进程体（也可从一个过程体内或一个服务体内）的某一点转到另一点。

在 SDL/PR 中，它们与“GO TO”语句等效。使用标号来作为与语句相关联的入口点，如图 D-3.8.48所示。在进程体（或过程体）内部，不允许把控制转移到（因而也不允许把标号联到）如图 D-3.8.49所示的各种类型的语句。标号总是局部于一个进程的，因此不可能借助一个汇接把控制从一进程转移到另一进程。

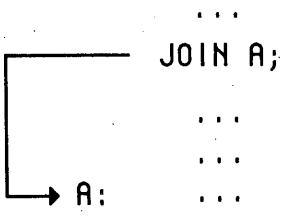


图 D-3.8.48
标　号

STATE,
ENDSTATE,
INPUT,
SAVE,
ENDDECISION

图 D-3.8.49
不允许用标号的地点

在 SDL/GR 中，汇接相当于连接符（出连接符和入连接符）。它们可用来（由于地方不够）把图进行分解，或者也可用来避免流线的交叉，流线的交叉会使图多少有点不清晰。此外，通常 SDL 图的可取画法是从一页的顶部画到底部。

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

在 GR 中，最好为出连接符指明相应的入连接符所在的参考页码，还应为入连接符给出相应的一个或多个出连接符的参考页码（见图 D-3.8.50中的例子）。

D.3.9 过程

SDL 中的过程类似 CHILL 和其它编程语言中的过程。引入过程的目的是：

a） 使进程的结构构成可进入一些细节层次；

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

c） 允许定义常用的项的组合，便于重复地使用。

过程定义只可以包含在进程定义、服务定义或过程定义中，因此一个过程只能为定义它的那个进程或过程所见到。

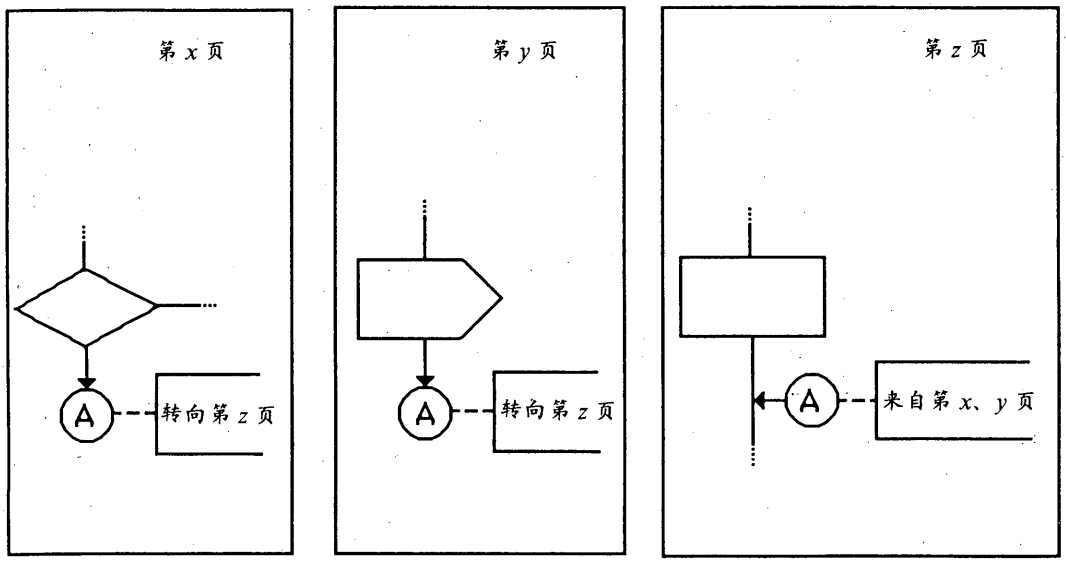


图 D-3.8.50

连接符的参考页码

过程定义由下列各项组成（其中有些是可任选的）：

— 过程名字。

— 过程形式参数：一张与各变量类别相关的变量名字表。它们用来在过程调用时向（或从）该过程传递信息。过程参数可以按值（IN 参数）传递，或者按引用（IN/OUT 参数）传递。若参数是按值传递的，则所规定的形式参数定义了一个局部于本过程的变量；若参数是按引用传递的，则所规定的形式参数为该变量定义了一个同义词。

— 过程定义：只能由本过程调用的过程。

— 数据定义：规定局部于本过程的数据类型。

— 变量定义：本过程内部的局部变量。

— 过程体：用状态和动作来规定本过程的实在行为（类似于进程体）。

在图 D-3.9.1中给出了一个 SDL/PR 的不完整的过程定义例子（关键字用的是大写字母）。注意不带显式属性的形式参数具有隐式的 IN 属性（图中的变量5(var5)）。

D.3.9.1 过程体

过程体很像进程体，但有下列不同

— 过程用“返回”来终止对它的执行，以代替进程的“停止”。在SDL/PR中，返回语句用关键字RETURN来表示。

```
PROCEDURE prcd1;
   FPAR IN/OUT var1,var2 sort1,          } 过程形式参数
        IN var3 sort2,                   }
        IN/OUT var4 sort3, var5 sort4;   }

   PROCEDURE ...                         } 过程定义
   PROCEDURE ...                         }

   ... data definitions ...              } 数据定义

   DCL ...                               } 变量定义

   ... procedure body ...                } 过程体

ENDPROCEDURE prcd1;
```

图 D-3.9.1

SDL/PR 中不完整的过程定义举例

— 在SDL/GR中，过程起动符号与进程起动符号略有差别。

过程的起动符号和返回符号可以在SDL/GR概要中找到。

过程可使用汇接构件，但只能指向它本身内部的一个标号。汇接既不能用于从过程外面进入过程，也不能离开过程转出去。

在SDL/GR中，过程定义用过程图来表示。过程图很像进程图，它由下列各项组成：

— 一个任选的框符号：一个包围所有其它符号的矩形符号。

— 过程标题：关键字PROCEDURE后面跟过程名字和过程形式参数的规格。过程标题通常置于框的左上角，如果没有框，则置于图纸的左上角。

— 一个任选的页码编号（置于右上角）。

— 正文符号：过程图可以使用正文符号来围住形式参数、数据和变量定义的规格。

— 过程引用标记：即过程符号，每个符号含有一个过程名字，代表一个单独定义的局部过程。

— 过程图：用来直接定义局部过程。

— 过程流图区：用起动、状态、输入、输出、任务、…及有向弧等来规定过程的行为。

图D-3.9.2中有一个SDL/GR的过程定义例子。例子中被引用的过程“TERM_P”是局部于进行调用的过程的。

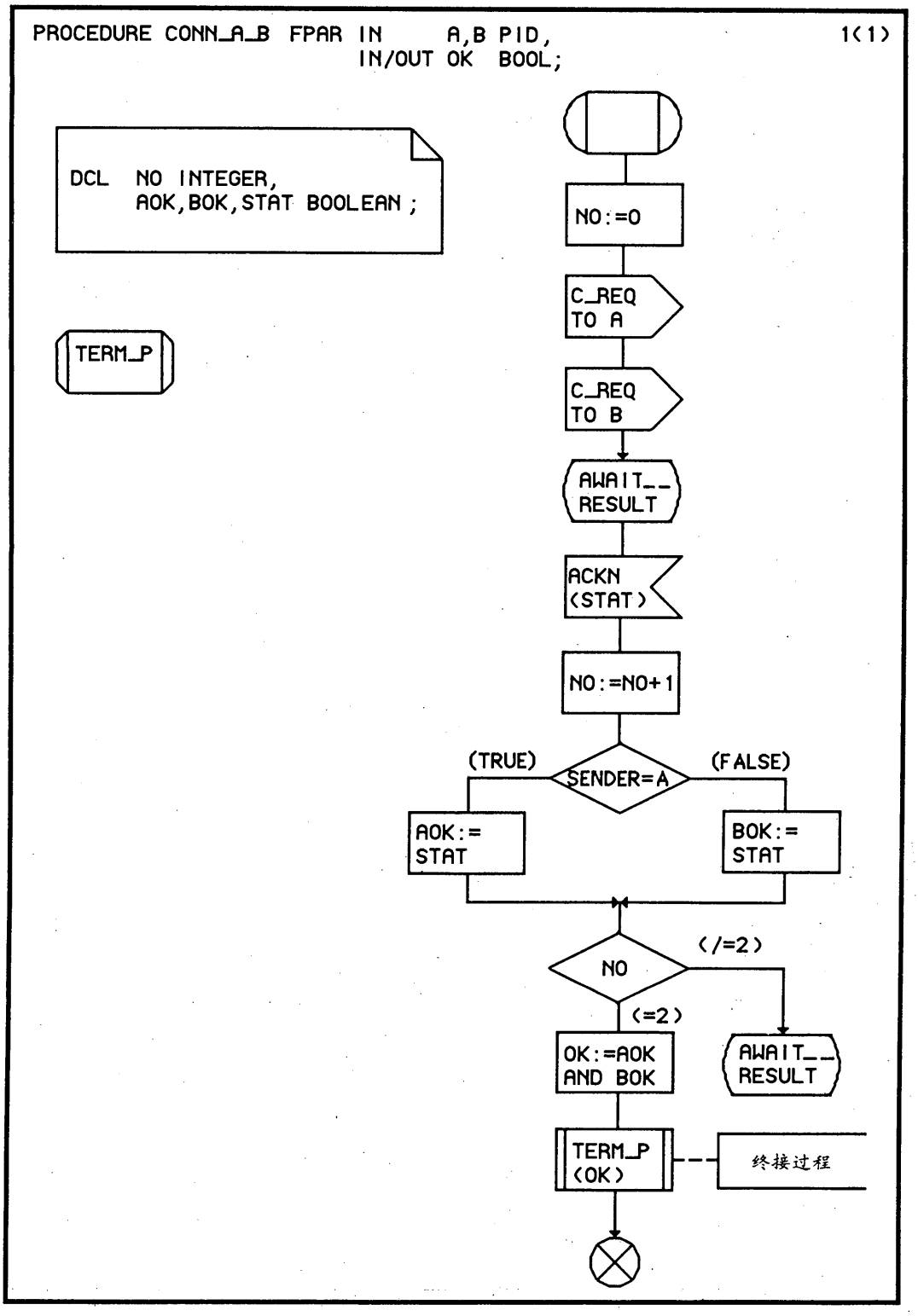


图 D-3.9.2
过程图的例子

如进程图中所述那样（§D.3.8），若一个过程图在一页中容纳不下，该图可分成几页，每页给出框符号、标题及页码。

D.3.9.2 过程调用

在进程或过程流图中，凡是允许有任务的地方都可以出现过程调用。在某种意义上讲，过程可作为任务来执行，但下述情况例外：

1） 一个过程可能含有状态；如果有状态，就要接收信号。

2） 过程可发送信号。发信号的进程实例就是调用该过程的进程实例。

当一个过程被调用时，就要创建过程环境，并且开始执行该过程。对过程的执行继续进行，直到到达 RETURN 为止。当正在执行过程时，所有传送给该进程的信号或者被隐式地保存，或者被该过程显式地加以处理。过程没有自己的输入队列，而是使用调用它的进程的输入队列。

在 SDL/PR 中，过程调用由关键字 CALL 后面跟过程标识符和括在圆括号内的实在参数表来表示。如果一个参数没有给出，就必须用两个连续的逗号来指明。在这种情况下对应的形式参数具有“不确定的”值。也要注意：在过程定义中对 IN 或 IN/OUT 进行了声明，所以就不允许在调用语句中重复声明。SDL/PR 中调用的一些例子示子图 D-3.9.3。

```
CALL proced1;
CALL proced2(var1,var2);
CALL proced3 (5,a+b,Pid1);
CALL proced4 (W,X,,Z);
```

图 D-3.9.3

SDL/PR 中调用语句举例

在 SDL/GR 中，过程调用由含有过程名字和实在参数表的过程调用符号来表示，实在参数括在圆括号内。SDL/GR 中调用的一个例子在图 D-3.9.2中给出。

D.3.10 数据处理

D.3.10.1 变量声明

变量是局限于进程实例的。这意味着所有的变量有一个且只有一个拥有它们的进程实例，只有拥有者进程实例可以改变这些变量的值。

在图 D-3.10.1中所声明的变量局限于进程 P 的每个实例，因此它们只能被进程 P 的各个实例来访问和修改（每个进程实例能访问或修改它自己的变量副本）。

```
SYSTEM S;
   . . .
   BLOCK B;
   . . .
      PROCESS P(3,10);
         DCL
          A Integer,
          D Integer;
         . . .
         . . .
      ENDPROCESS P;
   ENDBLOCK B;
   . . .
ENDSYSTEM S;
```

图 D-3.10.1

变量声明举例

变量可直接在声明后面置初始值，如图 D-3.10.2所示。

```
DCL A Integer :=1;
```

图 D-3.10.2

变量置初始值

SDL 允许用两种为变量置初值的办法。可以在数据类型定义中用 DEFAULT 语句为某一类别的所有变量声明一个初值（见§D.6.4.5），在这种情况下，赋初值对该类别的所有变量均有效。

另一方面，可以为某一类别的每个变量置一个初值，如图 D-3.10.2所示。如果声明中既有一个 DEFAULT 语句，又有一个初始值，则后者生效。若变量未置初值，则认为其初值在系统中是“不确定的”。

当然，示于图 D-3.10.2的变量置初值的简化写法只适用于简单的变量，或者适用于一些数据类型，其变量可以简洁具体地表示（另一个例子示于图 D-3.10.3）。

图 D-3.10.3表示：Struct 类别具有一个简洁具体的表示法来表示一个结构值。为了生成数组，则建议在进程的初始跃迁语句串中模仿一个“While 构件”，显式地为变量置初值（见图 D-3.10.4）。

```
PROCESS P1;
    NEWTYPE S Struct
        I Integer;
        B Boolean;
    ENDNEWTYPE;
    DCL
     A S := (. 1,True .);
    . . .
ENDPROCESS P1;
```

图 D-3. 10. 3

变量置初值的另一个例子

```
PROCESS P;
        NEWTYPE Arr1 Array (Nat1,Integer)
        ENDNEWTYPE Arr1;
        SYNTYPE Nat1 Natural CONSTANTS 1:3
        ENDSYNTYPE Nat1;
        DCL
         A Arr1,
         I Natural;
        START;
        TASK I:=1;
  Lab1: DECISION I<=3;
            (True): TASK A(I):=I;
                    TASK I:=I+1;
                    JOIN Lab1;
            (False): ;
        ENDDECISION;
        . . .
ENDPROCESS P;
```

图 D-3. 10. 4

变量置初值的一个较复杂的例子

D. 3. 10. 2 透露的/视见的变量

除了使用信号，两个进程还可用其它方法来交换信息。一个进程可以用 VIEW 操作来访问为另一进程所拥有的变量的值，但有几条规则要注意：

— 两个进程必须属于同一个功能块。

— 执行 VIEW 操作的进程必须在视见定义中规定被视见的变量的标识符。

— 透露变量的进程必须声明该变量具有 REVEALED 的属性。

— 在变量声明中的类别标识符（或同义类型标识符）必须和在视见定义中的一样。

执行视见的进程通过 VIEW 操作所得到的值和执行透露的进程通过一般访问得到的值是相同的。

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

在本节的余下部分中，假设进程 R（透露者）拥有变量同时又透露变量，而进程 V（视见者）在其视见定义中引用这些变量。

企图在进程 R 被创建之前就在进程 V 中视见变量会导致产生 SDL 错误。用户可以任择下列两种方法之一，来避免这个问题。

— 保证进行透露的进程实例 R 在进行视见的进程实例 V 之前就已创建，并已预置了有关变量的初值；或

— 保证 V 在 R 已被创建并预置了有关变量的初值之前不进入要使用透露的/视见的变量的跃迁。

获得前一种情况的简便方法是使 R 成为 V 的父亲（或先辈）（如图 D-3.10.5中的例子），或在创建系统的同时创建 R（隐式创建）。在后一种情况下，可安排让 V 中的有关跃迁只能被来自 R 的信号所触发。

进程 R 停止以后，R 的变量就不能被视见。此后要视见数据的任何企图都将是一次 SDL 出错。用户可用下列两种方法避免这个问题，任择其中之一：

— 根本不在 R 中使用停止符号；或

— 保证 V 获知 R 将停止，并不再有任何视见有关数据的打算。

对于实现者来说，第一种解决办法的缺点是 R 没有释放它所用的数据存储区。

一个使用透露的/视见的变量的例子，在图 D-3.10.5中给出。

D.3.10.3 出口的/进口的值

一进程可声明其一个或多个变量为“可出口的”，其效果是使所有其它进程（不论它们属于哪一个功能块）在申请时可进口该变量的值的副本。进行进口的进程必须在其进口定义中声明该变量。

当进行出口的进程执行一次出口时，变量的值被复制在一个隐式的变量中。进口的进程通过进口表达式获得这个副本的值。因此由进口表达式得到的值可以与由变量拥有者通过正常访问所得到的值不同，即使它们在同一时刻执行。

在图 D-3.10.5中，假定进程 P1是进程 P2和 P3的父亲，所以在 IMPORT 表达式和 VIEW 表达式中的进程实例标识符用标志 PARENT 来指明。

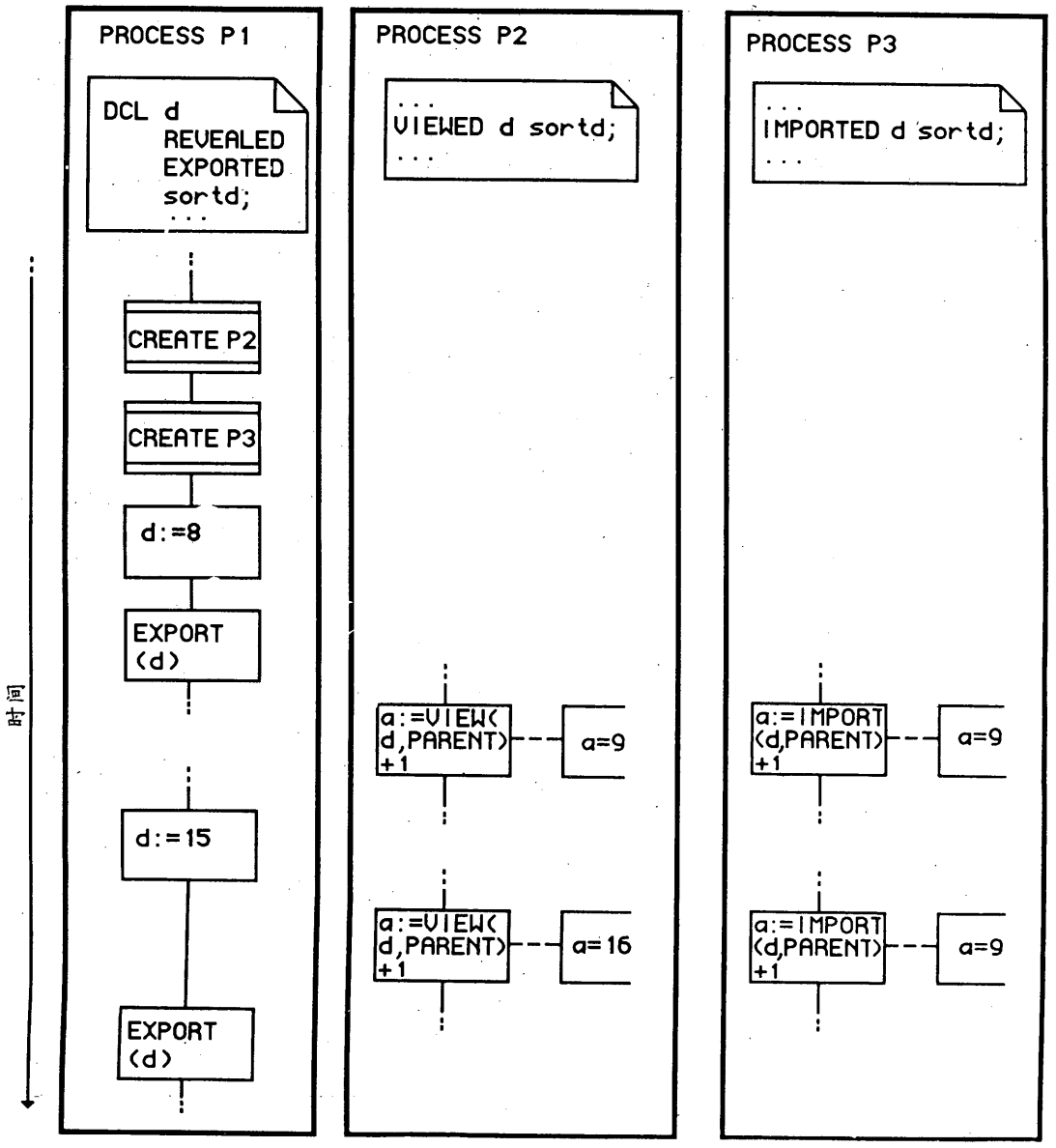


图 D-3.10.5

透露的/视见的变量与可出口的变量之间的用法区别举例

D.3.10.4 表达式

在一个SDL进程中，表达式可以用来作为判定、替换、选择、任务、连续信号、允许条件和设置构件的形式正文。表达式也可用作输出、过程调用和创建构件的实在参数。Pid表达式用在输出构件的TO部分内。替换和选择构件中的表达式（静态计算）应该是预定义数据类别。表达式中可有基本项（即只含有常量值的表示的项）和带有变量的项。

SDL有一组预定义的中缀运算符集合。这些运算符可用于任何数据类型，而且也只允许采用这些中缀运算符。对于这些运算符的优先规则也是预定义的，并且不能改变。预定义的中缀运算符是：

⇒，OR，XOR，AND，IN，/=，=，>，<，<=，>=，+，-，//，*，/，MOD，REM

SDL还提供预定义运算符：

－，NOT

它们是一元前缀运算符。

使用这些预定义运算符时，就优先规则而论应该小心谨慎，因为这些运算符是从不同的应用领域独立地予以预定义的，所以容易搞错。

例如，考察图D-3.10.6中的新类型和表达式，人们想到乘法和加法的优先规则，会把表达式按A⇒（（B*C）+D）进行计算，但这与AND和OR的运算优先规则不同，因此这样一变就会引起混淆。更有甚者，这种改变还会导致公理的不一致而使SDL规格无效。

```
NEWTYPE Newbool
    INHERITS Boolean
    OPERATORS ("+"="AND", "*"="OR", "=>")
ENDNEWTYPE Newbool;

...

A=>B*C+D
```

图 D-3.10.6

在表达式中采用容易混淆的符号的例子

由用户定义的所有其它运算符都是函数，并且必须采用前缀表示法。

前一节已经说明，VIEW和IMPORT这两个操作符具有特定的语义。无论如何，它们返回某一类别的一个值，该值可以是一个表达式内的另一个操作数。

D. 3. 11　SDL 中的时间表达

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

在 SDL 中，“计时器” 模型是一些元进程，它们在遇到请求时能向进程发送信号。要使用计时器必须先在进程定义内的计时器定义中加以声明。“SET” 操作和 “RESET” 操作是用来激活计时器的。SET 操作即置时操作请求在达到一特定时间时发生到时，而 RESET 操作即清零操作则取消该规定的到时。（应注意：在计时器未发生到时的时候再进行 SET 操作，就等效于先 RESET 再 SET。）

计时器的 SET 构件含有所请求的延时的时间表达式、所涉及的计时器名字、以及可任选的一些表达式组成的表。这些表达式按次序规定了计时器信号中将具有的值。

在 RESET 构件中可以规定一些表达式组成的表，用来取消计时器实例中具有相同值的实例。

计时器定义必须包含那些相应于 SET/RESET 表达中使用的类别的引用类别标识符表。

SET 语句的一个例子示于图 D-3. 11. 1。

在置时 SET 构件中必须规定一绝对时间。通过加上表示当前时间的简单函数 “NOW”，可以把相对时间转换成绝对时间。所请求的延时的表达式应该是一个时间表达式；时间是一个预定义类别，从实数类别里派生出来。

为了接收一次到时，要在一个输入符号中规定定时器名字，如图 D-3. 11. 1所示。

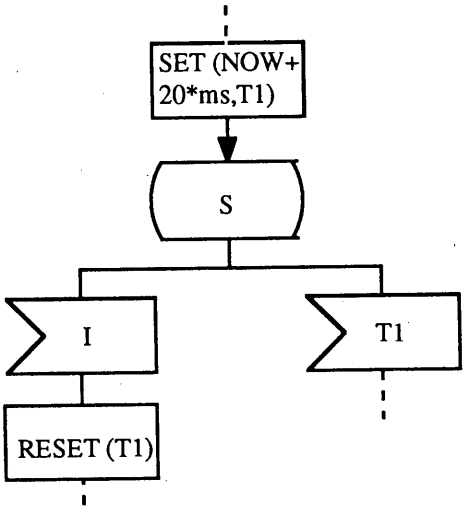


图 D-3. 11. 1
计时器使用举例

可以定义同义词来指明所要求的时间段。一经选定时间及时间段单位，用户就需定义所需的同义词来表示时间段，如图 D-3. 11. 2所示。

```
PROCESS p;
    TIMER T1 subscriber_id;
    . . .
    DCL Sa subscriber_id;
    . . .
    SYNONYM
        Sec  Duration = 1000.0,
        Min  Duration = 60000.0,
    . . .
    START;
        . . .
        SET  (NOW +20*Min +30*Sec , T1 (Sa));
        . . .
        RESET  (T1 (Sa));
        . . .
ENDPROCESS p;
```

图 D-3. 11. 2

表示时间段的同义词

SDL 建议中指出：允许将计时器设置为一已经“过去了的”时间。采取这一决定是为了便于系统的模拟，然而由于这样一种设置会形成不明确的规格，故应该避免这种用法。

D. 3. 12　限定符的用法

在 SDL 中，当一个规格中的项目名字不能唯一地确定该项目时，就用增加限定符的办法来指定这些项。当然，当定义项目时，只有名字是必须规定的；但当在其定义范围以外来引用它时，就需要一个由限定符和名字组成的标识符。

这也可应用于采用远方定义的场合：在定义范围内引用该定义时只要用该项目的名字；而远方定义则要用加了限定符的名字来规定它所在的范围。

图 D-3. 12. 1中有一个在进程中使用限定符的例子。该进程能接收两个不同的信号，它们具有相同的名字，但标识符却不同！第一个输入指的是在功能块层次上定义的信号类型；第二个输入是指在进程定义中定义的信号类型。第二个输入中的限定符可以省略，因为当标识符不加限定符时，指的是最内层的定义。

```
SYSTEM s;
   ...
   BLOCK b;
      SIGNAL x;
      ...
      PROCESS p;
         ...
         SIGNAL x;
         ...
         STATE wait;
            INPUT  SYSTEM s/BLOCK b  x;
               ...
            INPUT  PROCESS p  x;
               ...
         ENDSTATE wait;
         ...
      ENDPROCESS p;
   ENDBLOCK b;
ENDSYSTEM s;
```

图 D-3.12.1

限定符用法举例

D.3.13 名字的语法

在 SDL 中，名字可由一个字组成，或者由几个字组成，字与字之间用定界符（空格或控制字符）隔开。当采用长的名字时，第二种方法可使规格更加易读，特别是用 SDL/GR 时，因为图形符号尺寸大小有限。由几个字组成名字的例子示于图 D-3.13.1。

每当在一个名字中用几个字会产生二义性时（见图 D-3.13.2中的例子），两个字间的定界符必须用一条底线符（"_"）代替。将定界符改为底线符后得到的字符串仍然代表同一个名字；所以，举例来说，BUSY SUB 和 BUSY_SUB 一样指的是同一个名字。在图 D-3.13.3中给出一些用名准确的例子。

有一点很重要，即要注意底线符也能在名字中用作连接符，使名字可以分写成几行。在这种情况下，含有一个底线符并在后面跟有一个或多个定界符的名字，也可用略去底线符和定界符两者的办法来表示。

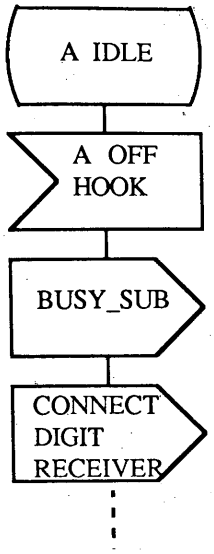
对同一名字的不同表示法的例子在图 D-3.13.4中给出。

```
STATE A IDLE;

   INPUT A OFF HOOK;

    OUTPUT BUSY_SUB;

   OUTPUT CONNECT
            DIGIT
             RECEIVER;
```



SDL/PR　　SDL/GR

图 D-3. 13. 1

由几个字组成的名字举例

a)　A := BLOCK B1 B / PROCESS P1 P C;

b)　DCL A B C D;

c)　NEWTYPE X Y Z (PAR) ENDNEWTYPE;

图 D-3. 13. 2

名字使用不准确的例子

a)　A := BLOCK B1_B / PROCESS P1_P C;

b)　A := (BLOCK B1 B) / (PROCESS P1_P C);

c)　DCL A B_C_D;

d)　DCL A_B C_D;

e)　DCL A_B_C D;

f)　NEWTYPE X Y_Z(PAR) ENDNEWTYPE;

g)　NEWTYPE X_Y Z(PAR) ENDNEWTYPE;

图 D-3. 13. 3

名字使用准确的例子

a) CONNECT DIGIT RECEIVER

b) CONNECT_DIGIT_RECEIVER

c) CONNECT DIGIT_RECEIVER

d)
```
CONNECT
DIGIT
RECEIVER
```

e)
```
CONNECT DI_
GIT RECEIVER
```

f) CONNECT DI_ GIT RECEIVER

g)
```
CONNECT_
_DIGIT_
_RECEIVER
```

图 D-3.13.4
对同一名字的不同表示法举例

D.4 SDL 系统的构成和具体化

D.4.1 概述

这一章将讨论某些技术和 SDL 构件，它们可自顶向下地规定大的系统的规格。这些技术通常使用“划分”和“具体化”等术语，其含义如下：

— 划分：将系统的一个部件再分成几个更小的部件，这些部件的总体行为相当于未被划分过的该部件。划分能用于功能块（由新的子功能块、信道和子信道构成、信道（由功能块、新的信道和子信道构成），以及进程（由服务构成）。

— 具体化：为系统的功能增添新的细节。从环境角度看来，系统的具体化可以丰富它的行为，比如能够处理更多种类的信号和信息。

请注意：一个系统部件的内部结构提供了关于该结构的较多细节，却不一定提供该系统的行为的较多细节。从概念上讲，可以区别一较详细的行为表示（例如对一个新信号的处理）和一较详细的结构的面貌，但实际上这两个方面互相交织在一起，所以在提供有关系统结构的新的细节时，也同时提供系统行为的新细节。

在 SDL 中，一个系统的最小结构如本建议的第二章所描述的那样，即：一个系统由一组

以信道连接的功能块组成—而功能块含有进程。

关于在详细程度不同的几个层次上的划分以及信号具体化的概念在建议的第三章中介绍。对于不需要进一步划分的系统，这种概念就不必要了。

D.4.2 划分准则

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

有一些对系统表示进行划分所遵循的准则，它们是：

a) 定义一定大小的智能化的且易管理的功能块或进程；

b) 做到与实际的软件和/或硬 件划分相协调一致；

c) 遵循固有的功能划分；

d) 使功能块间的交互作用最小；

e) 重复使用已有的规格（例如信令系统）。

实际采用的准则取决于许多因素，包括所要求的详细程度。

由于层次之间的关系取决于所选用的划分准则，清楚地说明已选定哪些准则是重要的，以便读者容易理解该表示法。划分的准则取决于用户，但存在某些限制，以保证正确的 SDL 表示法。这些将在后面几节中讨论。

D.4.3 功能块划分

可以把一个功能块划分成为一组功能块和信道，其方法几乎与将系统划分为功能块和信道的方法一样。在 SDL/GR 中，这是用功能块子结构图来表示的。一个功能块子结构图的例子示于图 D-4.3.1。该图的第一个图是功能块 B1的功能块图，该功能块 B1含有一个引用其子结构的引用标记，第二个图是功能块 B1的子结构图。在第一个图中，用一条虚线连向信道 C2的功能块符号表示引用信道 C2的子结构的一个引用标记（见§D.4.5）。

在 SDL/PR 中，功能块的划分由功能块定义内部、关键字 SUBSTRUCTURE 和 ENDSUBSTRUCTURE 之间的一组定义来表示。功能块子结构中的诸定义与系统定义中所定义的一样，除此以外，还必须提供信道与子信道之间连接的规格，如图 D-4.3.2中所示。

在一个功能块定义内，诸进程的规格和一个功能块子结构的规格可以并存，在这种情况下，功能块诸进程在一定的详细度层次上表示该功能块的行为，而在子功能块定义里面的其它进程将在更具体的程度上表示同一行为。同时用进程和子结构两者来描述功能块的例子可在§D.4.7找到（图 D-4.7.1）。

如果在功能块内没有进程，只有功能块子结构，且若该功能块子结构未加引用标记，则在 SDL/GR 中提供了一种简化表示法来简化图形。这种简化表示法可以使功能块嵌套起来，把外面的功能块框看成是隐含的功能块子结构框。使用这种简化表示法，可以把图 D-4.3.1的例子画成图 D-4.3.3。

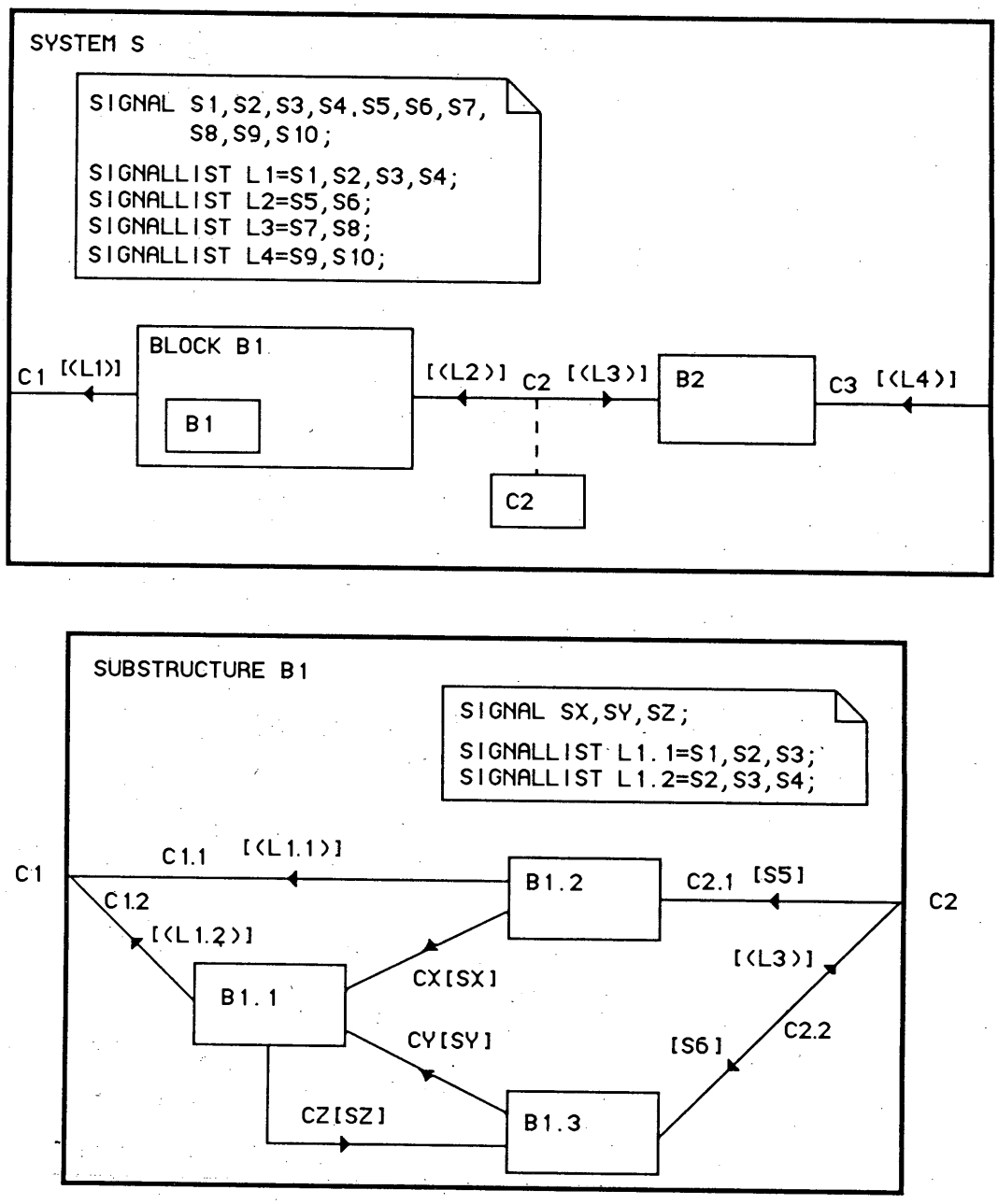

SYSTEM S
SIGNAL S1,S2,S3,S4,S5,S6,S7,
S8,S9,S10;
SIGNALLIST L1=S1,S2,S3,S4;
SIGNALLIST L2=S5,S6;
SIGNALLIST L3=S7,S8;
SIGNALLIST L4=S9,S10;
BLOCK B1
B1
C1
[(L1)]
[(L2)]
C2
[(L3)]
B2
C3
[(L4)]
C2
SUBSTRUCTURE B1
SIGNAL SX,SY,SZ;
SIGNALLIST L1.1=S1,S2,S3;
SIGNALLIST L1.2=S2,S3,S4;
C1
C1.1
[(L1.1)]
B1.2
C2.1
[S5]
C2
C1.2
[(L1.2)]
B1.1
CX[SX]
[(L3)]
CY[SY]
[S6]
C2.2
CZ[SZ]
B1.3

图 D-4.3.1

功能块划分

```
SUBSTRUCTURE B1;

    SIGNALLIST  L1.1 =S1,S2,S3;
    SIGNALLIST  L1.2 =S2,S3,S4;

    SIGNAL  SX,SY,SZ;

    CHANNEL  C1.1
        FROM B1.2 TO ENV WITH (L1.1);
    ENDCHANNEL  C1.1;
    CHANNEL  C1.2
        FROM B1.1 TO ENV WITH (L1.2);
    ENDCHANNEL  C1.2;
    CHANNEL  C2.1
        FROM ENV TO B1.2 WITH S5;
    ENDCHANNEL  C2.1;
    CHANNEL  C2.2
        FROM ENV TO B1.3 WITH S6;
        FROM B1.3 TO ENV WITH (L3);
    ENDCHANNEL  C2.2;
    CHANNEL  CX
        FROM B1.2 TO B1.1 WITH SX;
    ENDCHANNEL  CX;
    CHANNEL  CY
        FROM B1.3 TO B1.1 WITH SY;
    ENDCHANNEL  CY;
    CHANNEL  CZ
        FROM B1.1 TO B1.3 WITH SZ;
    ENDCHANNEL  CZ;

    CONNECT  C1  AND  C1.1,C1.2;
    CONNECT  C2  AND  C2.1,C2.2;

    BLOCK  B1.1  REFERENCED;
    BLOCK  B1.2  REFERENCED;
    BLOCK  B1.3  REFERENCED;

ENDSUBSTRUCTURE  B1;
```

图 D-4.3.2

用 SDL/PR 来表示图 D-4.3.1中的例子

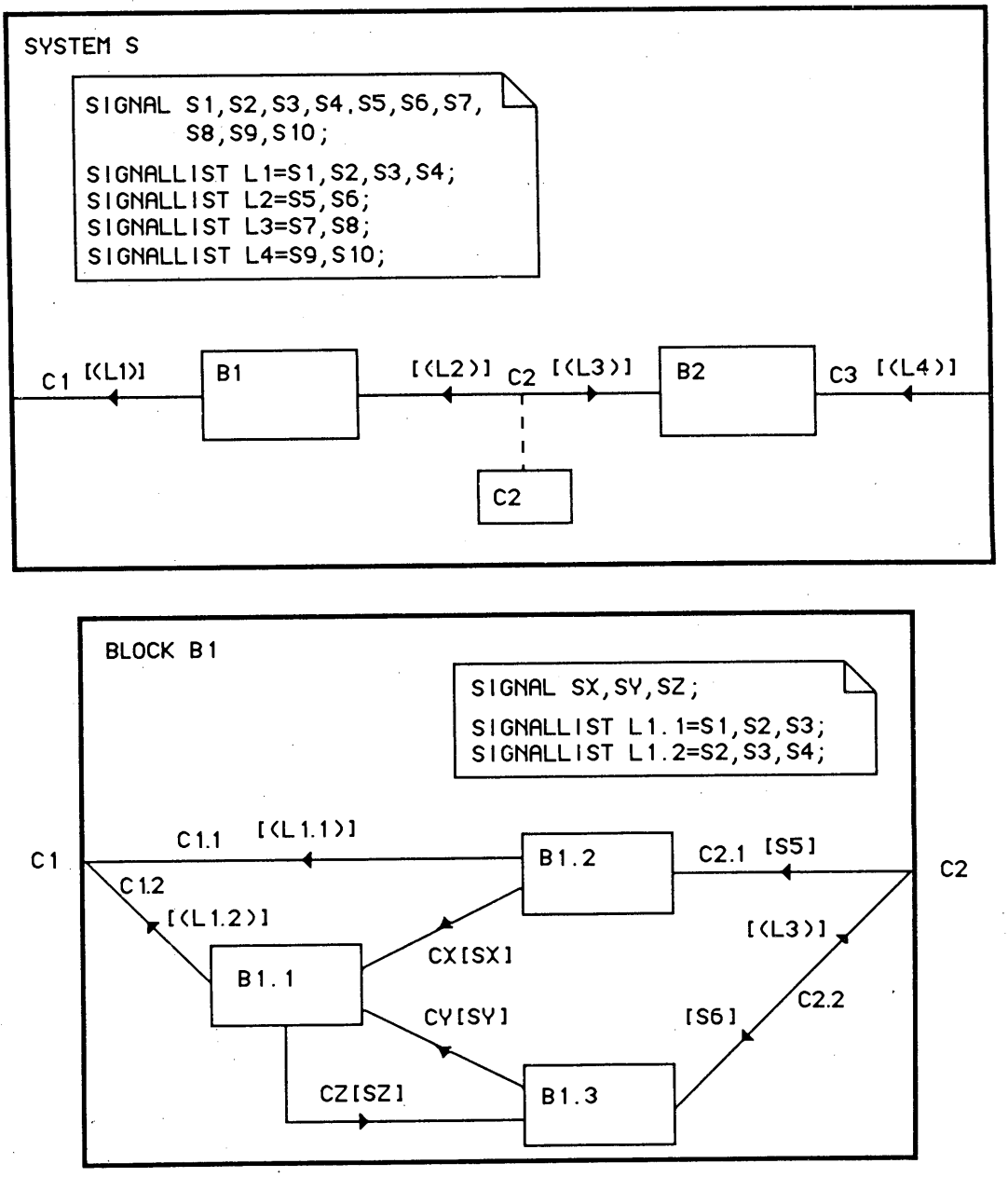


图 D-4.3.3

SDL/GR 的功能块划分简化表示法

从划分一个功能块得出的每一个子功能块本身还可以再进行划分，结果形成一种功能块及其子功能块组成的分层的树状结构。一种称为“功能块树图”的辅助图表示了这种结构的一般情况，见§D.4.4中所述。

除非涉及信号的具体化，进行划分时必须遵守下列规则：

1) 连接到一个外入信道的诸子信道，其信号表中必须不包含新的信号，但必须包含原信

道表中的所有信号(对于双向信道则必须考虑到外入路径表)。因此在图 D-4.3.1所示的例子中，C2.1和 C2.2在外入路径上传送 L2中的全部信号；此外，在 C2.2的外入路径上，不能出现由 C2.1传送的信号。

2) 连接到一个外出信道（例如 C1）的诸子信道（例如 C1.1和 C1.2)，其信号表中必须不包含新的信号名，但必须包含原信道中的所有信号名。因此 L1.1和 L1.2包含了 L1中的所有信号名，信号表 L1.1和 L1.2中可以含有相同的信号标识符。

3) 若原功能块含有进程，则可有两种选择。第一，每个进程的副本可在某个新的子功能块中重新定义；第二，可在子功能块内定义新的进程，但必须做到接口不变。

4) 父本功能块中的数据定义可用于其诸子功能块，因此每个子功能块都能使用已在父本功能块中定义了的数据类型，不需对它重新定义。

5) 若在父本功能块中已经定义的某一数据类型在一子功能块中用同一名字被重新定义，则该新定义只能用于作出定义的该子功能块，而其它子功能块仍沿用旧定义。不应提倡只是为了刻划某一子功能块的特征而进行一次重新定义，因为它会被读者所忽视，读者会认为旧定义是有效的。在某些情况下，特别当涉及到行为的进一步具体化时，进行重新定义是应该的。这时应注意用适当的注解来强调这种重新定义。

D.4.4 功能块树图

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

功能块树图是由一些功能块符号和一些“划分”线所组成的分层树，如图 D-4.4.1所示。

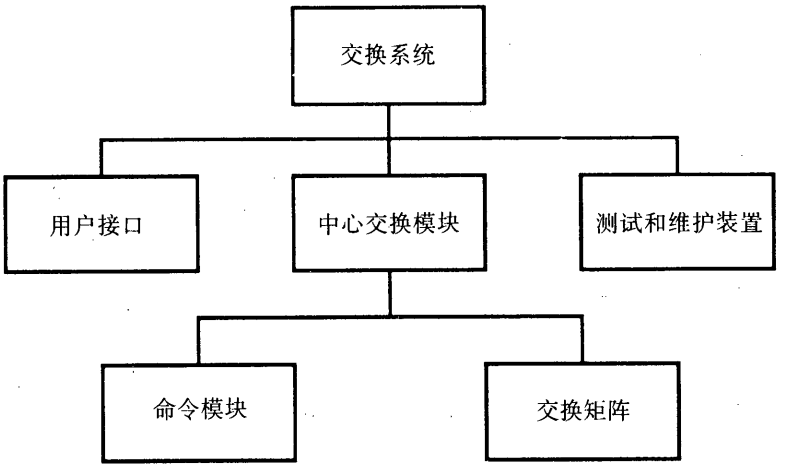


图 D-4.4.1

功能块树图举例

该图中所有功能块符号最好画得一样大，这就使图中同一划分层次内的各功能块并排出

现，在图中呈现相同的高度。如果图大，一页画不下，那就应将它划分成“部分”功能块树图，如图 D-4.4.2所示。

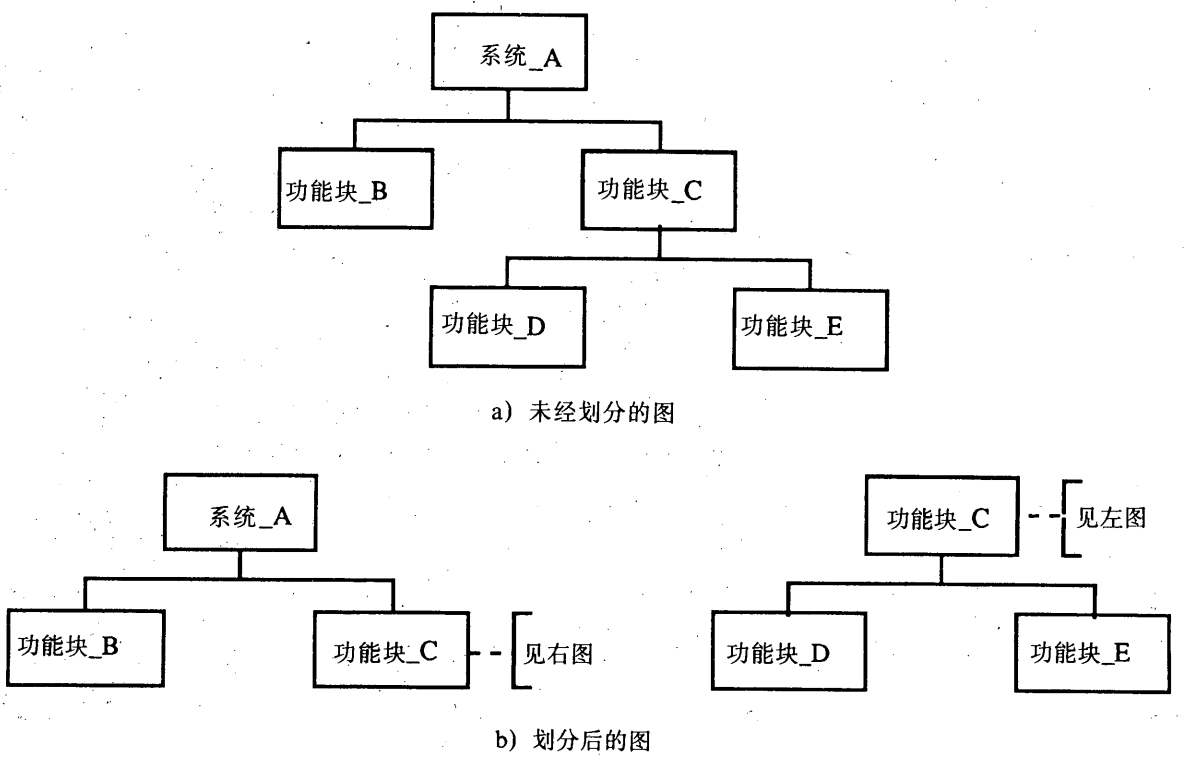


图 D-4.4.2

部分功能块树图举例

把一个功能块树图划分成几个部分图常常是有用的。

划分成若干部分图的目的是使以系统作为根的第一个图被截断，且使那些将被进一步划分的功能块看起来象不再被划分。原图被截断处的各功能块则在表示进一步划分的图中作为根出现。

如果用了部分图，而进一步划分一个功能块和/或在什么地方找到接续的图又不很明显，则应使用注释符号说明相互的联系。

D.4.5 信道划分

信道可以划分，与它所连接的诸功能块无关。这样就可表示信道的行为。为了准确地表示信号被传送的情况，在某些情况下会有必要表示信道的行为。做到这一点的办法是把信道看成是一个独立的项目，就如把它所连接的两个功能块看作是信道的环境一样。用这种方法考察信道，就可以用功能块、信道和进程来表示它的结构。

在 SDL/GR 中，信道划分由信道子结构图来表示，如图 D-4.5.1所示（该例表示图 D-4.3.1中信道 C2的子结构）。

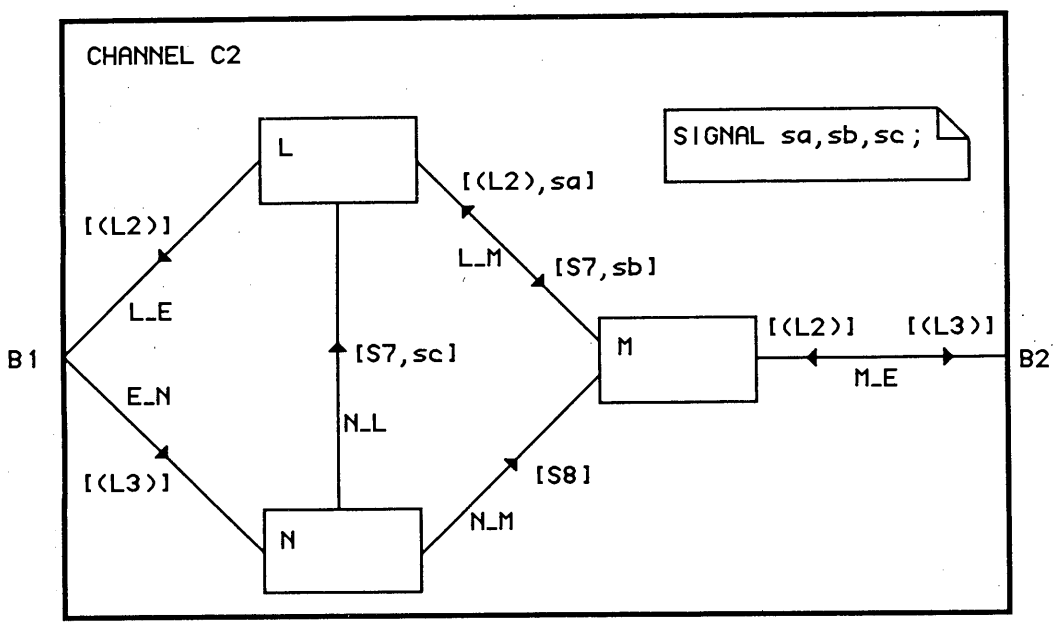


图 D-4.5.1

信道子结构图的例子

信道子结构图描述一个信道如何被划分成子成分。该图类似系统图（只有连接到功能块这一点不同），所有在§D.4.3中给出的准则对信道子结构图也是有效的。

在功能块交互作用图中，在出现被划分的信道的地方应该加一注释，指出有关的信道子结构图，那里可以找到信道划分的描述。

在 SDL/PR 中，信道子结构的形式类似于功能块子结构定义的形式，其唯一差别是在 CONNECT 语句中，对应信道端点的子信道是连接到外部的功能块（图 D-4.5.2中的 B1和 B2），而不是连接到外部的信道。

允许在一个功能块和一个信道子结构之间进行值的输入/输出，这样就可以直接地表示出 OSI 模型，在该模型中，对等层的通信由信号交换来模拟，相邻层的通信则通过共享值来模拟。

D.4.6 划分情况下的系统表示

在如下的情况下，即把系统用一组由信道连接起来的功能块来表示，以及每个功能块的行

为由一个或多个进程来表达的场合，可用单一层次来表示。这意味着所看到的所有用于表示的元素是处在同一个层次上。当进行划分时，就在各种文件之间引入了一种层次关系。在系统由“n”个功能块组成的场合，将存在一个包含了系统结构表示的文件。

```
SUBSTRUCTURE C2;

    SIGNAL  sa,sb,sc;

    CHANNEL L_E
        FROM L TO ENV WITH (L2);
    ENDCHANNEL L_E;

    CHANNEL E_N
        FROM ENV TO N WITH (L3);
    ENDCHANNEL E_N;

    CHANNEL M_E
        FROM M TO ENV WITH (L3);
        FROM ENV TO M WITH (L2);
    ENDCHANNEL M_E;

    CHANNEL L_M
        FROM L TO M WITH S7,sb;
        FROM M TO L WITH (L2),sa;
    ENDCHANNEL L_M;

    CHANNEL N_M
        FROM N TO M WITH S8;
    ENDCHANNEL N_M;

    CHANNEL N_L
        FROM N TO L WITH S7,sc;
    ENDCHANNEL N_L;

    CONNECT B1 AND L_E,E_N;
    CONNECT B2 AND M_E;

    BLOCK  L  REFERENCED;
    BLOCK  M  REFERENCED;
    BLOCK  N  REFERENCED;

ENDSUBSTRUCTURE C2;
```

图 D-4. 5. 2

用 SDL/PR 表示图 D-4. 5. 1中的例子

另一个文件可能把这个系统表示成为是由不同的一组功能块所组成，其中有些功能块是从包含在原先文件中的功能块所导出的（原先文件中的有些功能块已被因功能块划分而得到

的子功能块所代替)。后一个文件必须和原先的文件相联系。

获得一个系统的完整表示不仅仅是为了联系文件彼此，而且还应按如下方法组织它们：即有可能在各个不同层次上来获得系统表示，从总的概览开始，一个层次比一个层次表示得详细，这意味着把各种文件分组，使它们形成不同层次的系统表示。

并非所有的层次都包含相同的元素。在第一层，系统表示可由一些功能块和信道表示所组成，不包括描述每个功能块行为的诸进程。在一较低的层次上，我们可能希望包括某些功能块的行为表示，但却不包括另外一些功能块的行为。表示的最低层次（表示最为具体的层）应包含所有功能块的行为的完整表示，亦即表达这种行为的进程的全集。

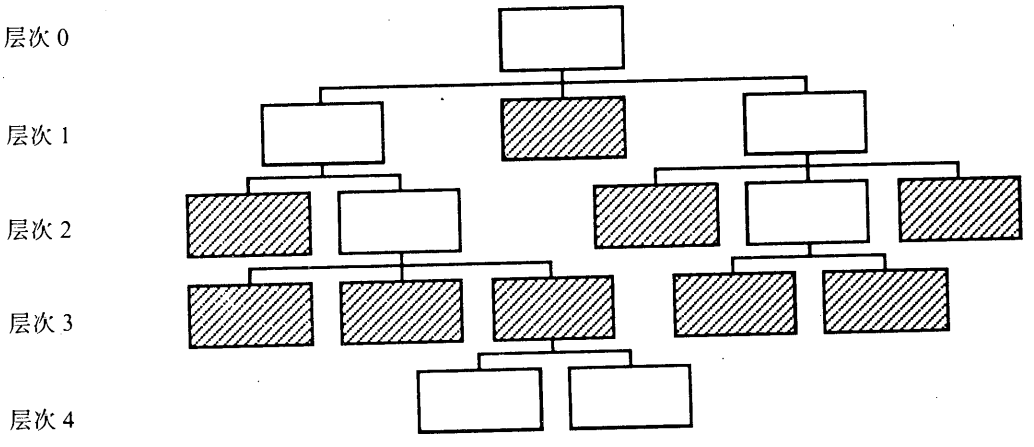
考察功能块树，可作出以下几点分析：

首先，每棵树总有一个且只有一个根，即系统。即使在一个系统从开头就被表示为由几个功能块所组成的场合，也是如此。在功能块树中，这些功能块将表示在比如层次1上。根可以只含有系统名字。对层次1上的诸功能块也可以给出信道定义，虽然这并不是必要的，除非诸功能块含有进程。

第二点考虑来自对树的叶子的考察：它们并非全须在同一个层次上。这可以是树上的功能块划分次数彼此不同的结果。划分的次数取决于几个方面，大多根据制定规格的人（或设计师）的主观判断。

SDL 要求仅当一个叶子功能块的行为已完全规定（即所关联的诸进程定义是以表示其行为时），它才不需要进一步进行划分，因此一个叶子功能块必须至少含有一个进程。

当一个系统表示是在几个抽象层次上给出时，可选择这些层次中的任何一个来表示该系统。选择某一个层次意味着要考虑该层次上的诸功能块，考虑与它们有关的进程，以及在较高层次上的所有叶功能块和它们所联系的进程（见图 D-4. 6. 1）。



注–表示层次 3 由打斜线的框标明。

图 D-4. 6. 1

用层次3来表示系统

为了不同的目的往往需要选择不同的层次才合适。例如，为了表示系统而选择一个概览层次，为了实现而选择一个较详细的层次。

在某一个层次上表示一个系统可能不完全，这是由于在该层次上有的功能块没有所关联的进程。

讨论表示的层次、以及选择某一个表示层次的理由是：

— 在设计中可能会达到某种详细“程度”，它由某个层次来表示（在这种情况下，该层次上的诸功能块都是叶功能块，且它们是不完全的，因为还须做进一步的工作）。

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

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

D.4.6.1 一致划分子集

除了总的系统规格和用不同层次给出的系统规格以外，SDL 还有“一致的系统子集”概念。它可被看成是一个单一层次的表示，这里的所有功能块可以取自系统结构的任何层次，条件是：

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

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

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

在某些信道已被划分的场合，由于把这些信道看作是系统，必须提供这些“系统”中的每个系统的规格。和通常的系统表示对比，这些规格应包含相同种类的文件。应该加上注解，说明这些系统与其所从属的主系统的规格的关系。这样，在某种意义上可把这些已划分的信道看成是内部系统，每个这样的系统可有几个表示层次，而且如果它们所含的某些信道被进一步划分，则还可再有内部系统。

D.4.7 具体化

在SDL中引入了具体化机制，以便为较高的抽象层次“隐蔽”低层次信号，并且可以自顶向下地规定系统的行为。

具体化可使用户将信号划分成子信号，形成一种层次结构，就像功能块和子功能块的情况一样。这就是说，能够在一个信号定义的内部定义一组新的信号，这些新信号称为所定义的信号的具体化信号或子信号。就象一个功能块定义可画出功能块树一样，出于说明性目的，信号定义也能用一棵信号树图来表示。

具体化与功能块划分有紧密的联系，因为只有与已划分的功能块相连接的信道来传送的那些信号才能予以具体化。也就是说，当信道所连接的功能块进行划分时，该信道信号表中所含的某个信号可由其诸子信号来取代。在功能块划分中产生的相应的诸子信道将在其信号表中规定这些子信号。

当定义一个信号为一信道所传送时，该信道即自动地成为该信号全部子信号的传送者，即使有些子信号的流动方向是相反的（在这种情况下，该信道被认为是隐式的双向信道），也是如此。

图D-4.7.1给出一个具体化的例子。这个例子表示了一个系统，系统中一个功能块内的一进程向另一功能块内的一个进程发送正文文件。在最高抽象层具体化是通过发送代表一个正文文件的信号（信号sf）来实现的。在下一具体化层，要规定这个正文文件是若干逐个发送的记录所组成（信号sr），并且接收器在消耗每个记录后必须作出应答（信号nr）。发送器在发送结束时要发送一个文件结束信号。该例中，在最高抽象层B1和B2内的进程使用信号sf进行通信；而在下一较低层B11和B21内的进程使用信号sr、nr和eof进行通信。

下面是几个能适用具体化概念的实际情况：

— 名字变换：把一个信号加工成另一个具有不同名字的信号。这是一个一对一的变换，仅仅改变信号的名字。当要使每层本身的不稳定性完全取决于本层时通常采用这种变换（根据上下文可方便地修改信号名）。

— 分裂变换：这是一对多的变换，这里一个信号通过某一表示特性被分裂成为若干个信号。例如一个普通的“Alarm”信号被分裂成“Register_Alarm”，“Central_Processor_Alarm”和“Subcriber_Alarm”。

— 算法变换：把原来信号变换为一组信号。该组信号可通过驱动一算法，以提供原来信号的信息。

D.4.7.1 一致的具体化子集

当把具体化应用于一系统规格说明时，一致地划分子集的概念只限于避免不同具体化层次之间的不同信号进行通信。在这种情况下，则认为该系统定义包含有若干一致的具体化子集。如果一个一致的具体化子集包含一个与子信号通信的进程，那么该进程就不应该与父本信

号通信，通信链路的另一端也应与上述子信号通信。

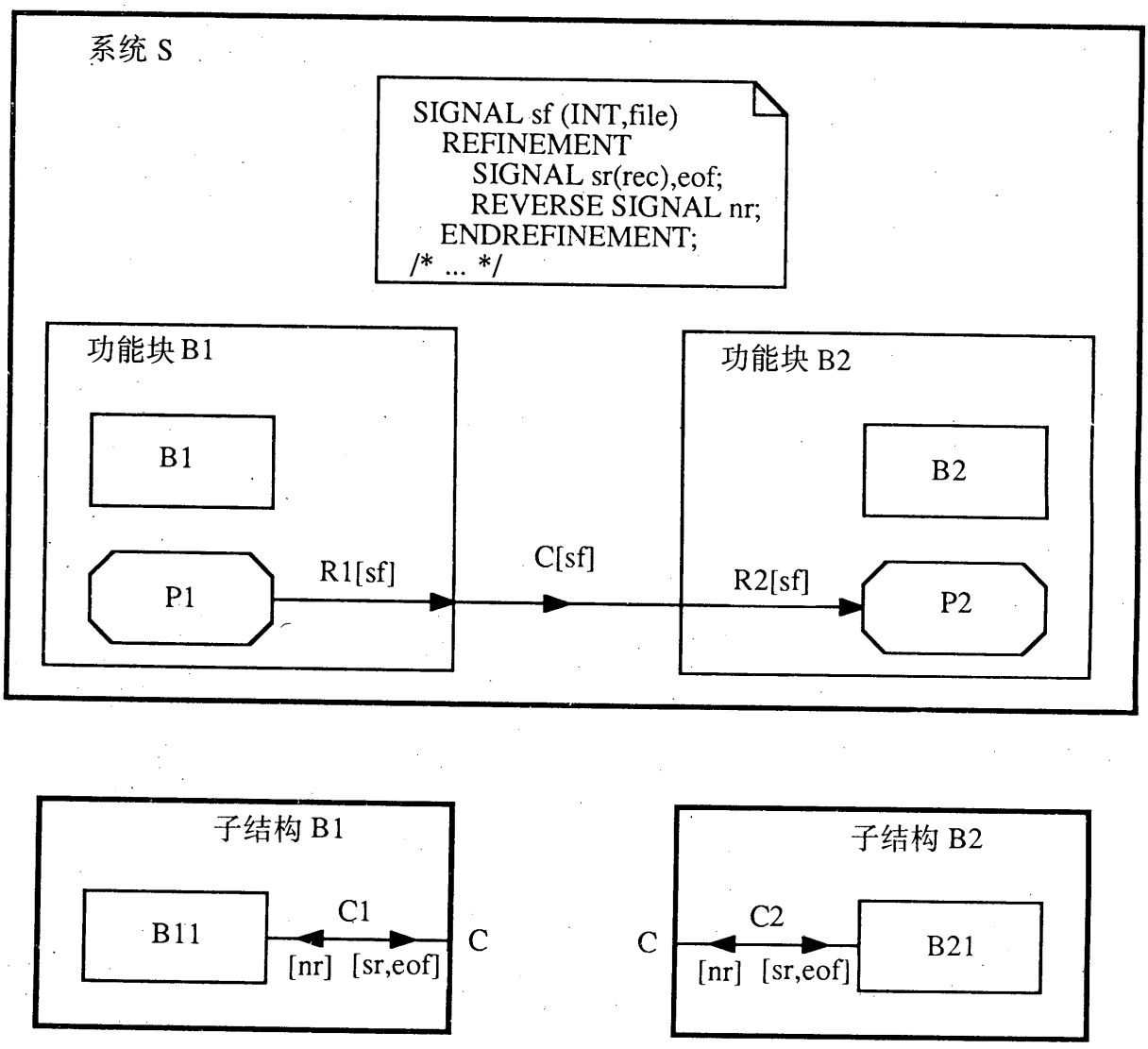


图 D-4.7.1
具体化举例

D.4.7.2 信号与子信号之间的变换

用户为了模拟的目的可能，需要描述不同具体化层次之间的变换，或者要在不同的具体化层次上检验系统的行为。这可以借助一些附加的 SDL 进程以非形式方式，或以形式方式实现，其中附加的 SDL 进程是用来描述一信号及其子信号的动态变换的。

图 D-4.7.2引入了两个进程，用来描述图 D-4.7.1中所用的变换。其中具体化进程规定了一个高层信号如何具体化成为一组在下面一层的信号，恢复进程则描述了相反的变换。

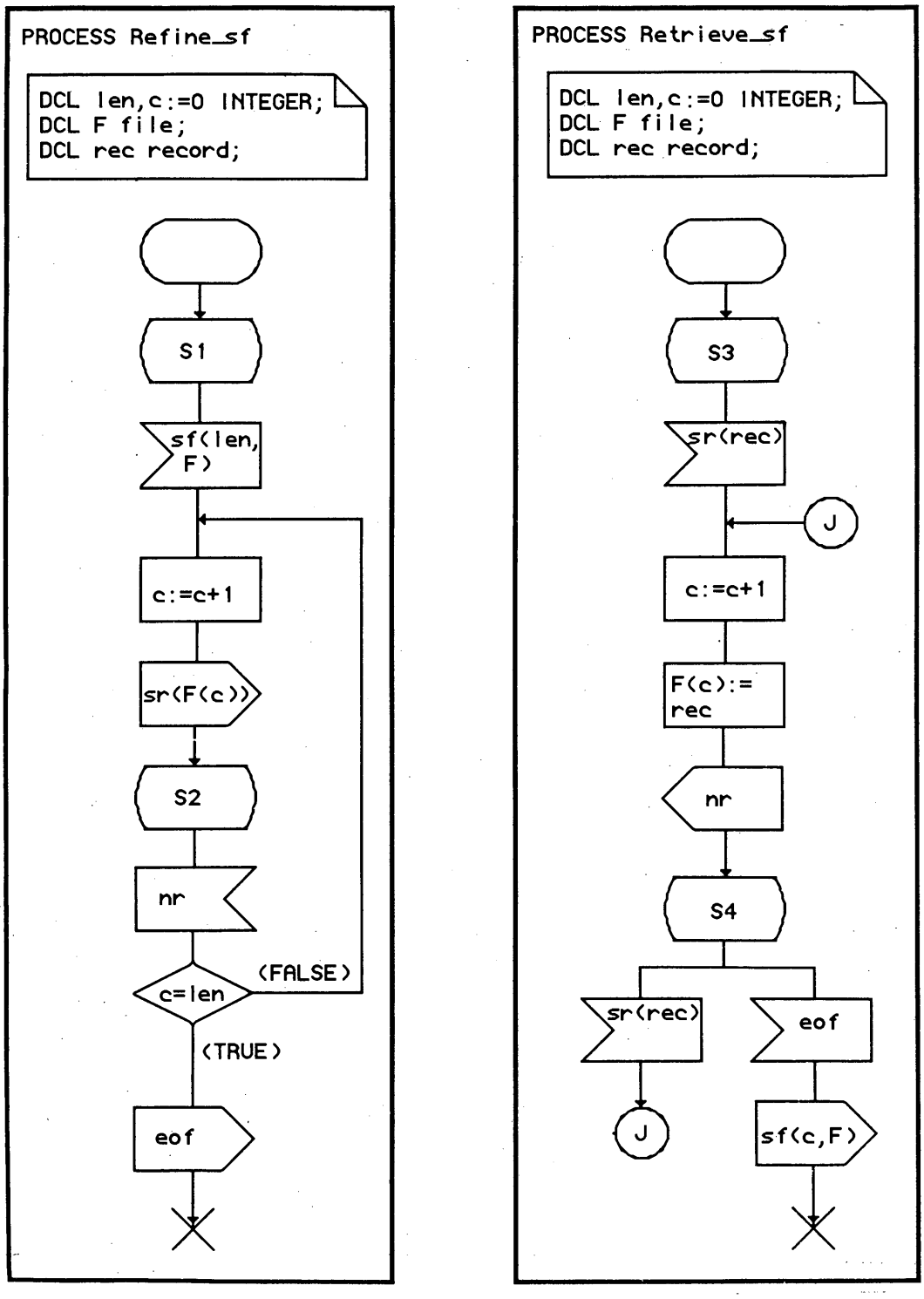


图 D-4.7.2

信号变换的规格举例

D.5 附加概念

D.5.1 宏

宏构件是一种用来处理重复和/或构造一个描述的手段。它由一个宏定义组成，该定义含有一SDL规格的某一部分，能在系统规格中的其它地方被引用（宏调用）。

凡是在允许定义数据的地方都能给出宏定义。然而宏名的作用范围不受限制，这样在一功能块内部定义的宏可以在该功能块外部被引用。

在SDL/PR中，有可能用一宏定义替代任意词法单位序列；这与只可替代语法单位集合的SDL/GR宏定义不同。

为了在SDL/GR和SDL/PR之间映射含有宏的SDL文件，必须对SDL/PR宏的使用进行如下的限制：

1. 一个宏只能替换下列的一个或多个语法构件（这些构件与SDL/GR符号相对应）：

   启动
   状态
   输入
   允许条件
   连续信号
   保存
   动作语句
   终止语句

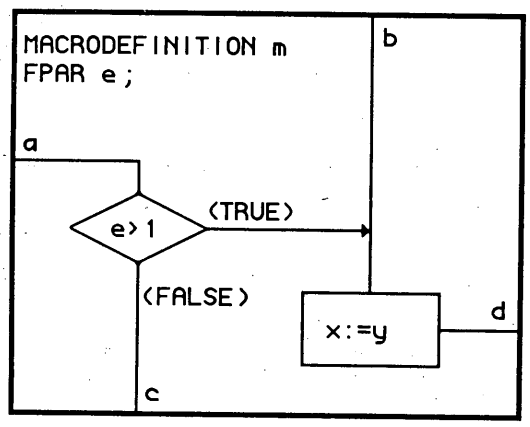
2. 在要确定SDL/PR构件的类型处不能使用宏形式参数，即在要使用SDL/PR关键字的地方不得使用宏形式参数。同样实参不应包含下列对应于SDL/GR符号的关键字：START，STATE，PROCEDURE，INPUT，TASK，OUTPUT，DECISION，CREATE，STOP，PROVIDED，CALL，COMMENT，JOIN，RETURE，SAVE或OPTION。

3. 宏定义中的每个语句必须至少从一宏入口处是可达的。

在SDL/PR中，宏至多有一个入口处和一个出口处，因此在SDL/RP中必须使用标号和汇接来分别表示具有多个入口处或多个出口处的SDL/GR宏。如果宏在多处被调用，则标号必须作巍—病//数来传递。

在SDL/GR中，有两种不同的方法表示宏定义的入口处和出口处。用户可以把宏画成一个框，并把入口处/出口处与框相连，也可以显式地使用入口处和出口处符号（宏的框是任选的）。

图D-5.1.1举例表示了具有两个入口处和出口处的SDL/GR和SDL/PR宏（该例中入口处和出口处与宏框相连）。



SDL/GR

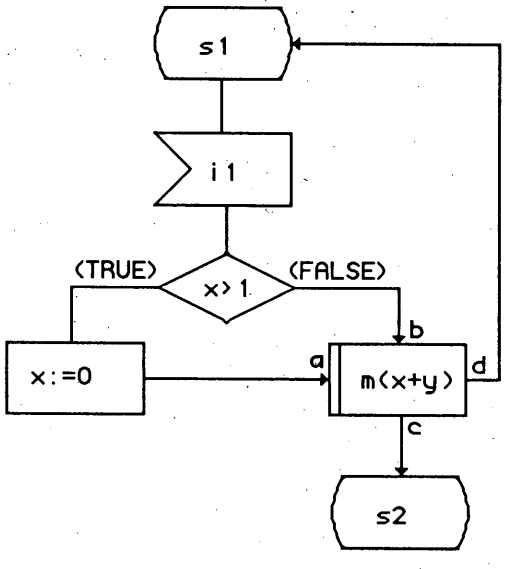
```
MACRODEFINITION m
   FPAR e,a,b,c,d;
   a: DECISION  e>1 ;
      (TRUE):  JOIN b;
      (FALSE): JOIN c;
      ENDDECISION;
   b: TASK x:=y;
   JOIN d;
ENDMACRO m;
```

SDL/PR

图 D-5.1.1

宏定义举例

这意味着在主要的 SDL 规格中（图 D-5.1.2）有相应的汇接和标号。注意对每个不同的宏调用标号通常是不同的。



SDL/GR

```
...
STATE s1;
   INPUT i1;
      DECISION  x>1 ;
      (FALSE): JOIN BB;
      (TRUE):  TASK x:=0;
               JOIN AA;
      ENDDECISION;
      MACRO m (x+y,AA,BB,CC,DD);
      CC: NEXTSTATE s2;
      DD: NEXTSTATE -;
...
```

SDL/PR

图 D-5.1.2

宏调用举例

图 D-5.1.3展示了两个定义同步机制的宏定义，注意这里使用了伪形式参数 MACROID 来产生不会重复的状态名。图 D-5.1.4例中也使用了入口处和出口处符号。

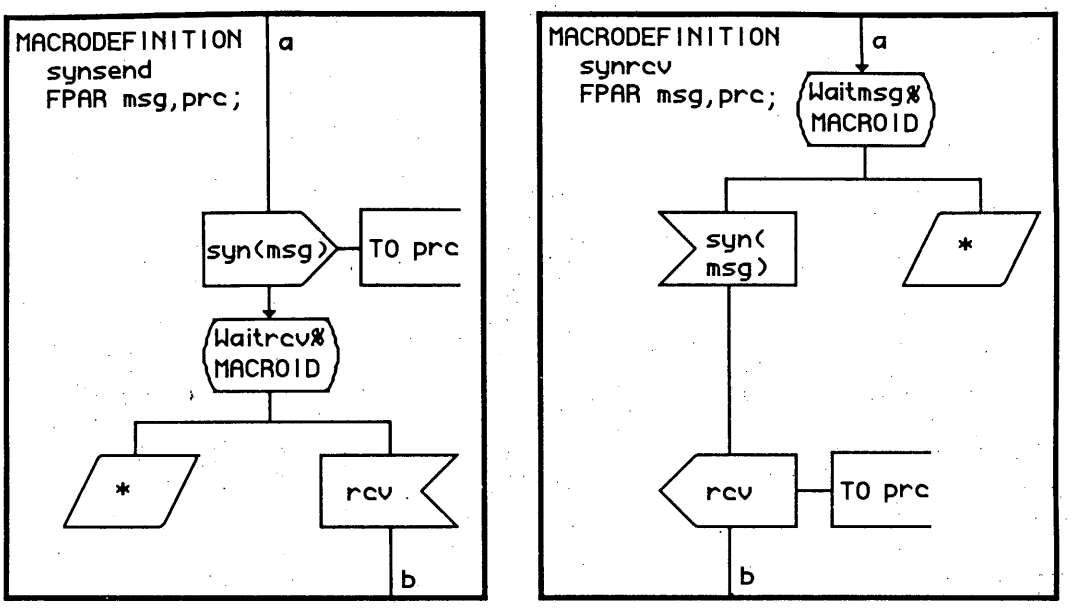


图 D-5.1.3

两个使用 MACROID 的宏

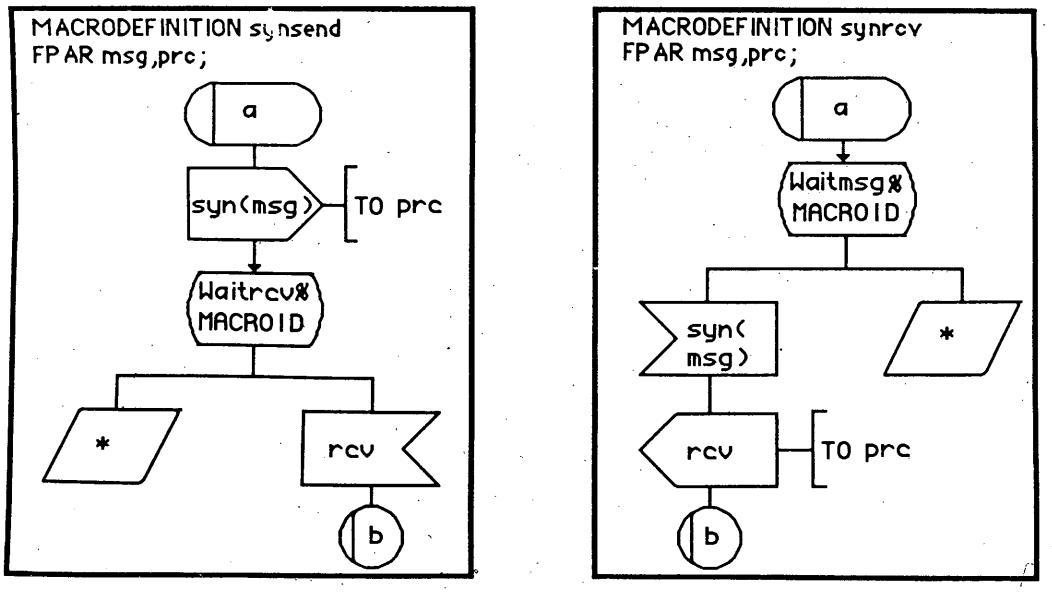


图 D-5.1.4

使用入口处和出口处符号举例

在嵌套宏的情况下，也就是当在一个宏定义的内部有一个或多个宏调用时，应该注意最外层宏的展俊一病"影响最内层宏的展开。更准确地说对于嵌套宏的展开应该先完成外层宏的展开，然后才开始展开此宏内部的宏调用。

D.5.2 类属系统

在 SDL 中，利用系统参数，可按照一个系统规格来定义不同的多个系统，该系统参数的值事先未作规定，它能由外部提供，以便按照用户需要得到一特定的系统定义。

系统参数实际上是外部同义词，能够用在可以使用同义词的任何地方。当然在解释一系统之前所有外部同义词都必须赋值。图 D-5.2.1表示了使用外部同义词的若干有效的例子。

```
SYSTEM s;
    . . .
    SYN inst.numb INTEGER = EXTERNAL;
    SYN rate.incr  REAL = EXTERNAL;
    . . .
    BLOCK b;
        PROCESS p (inst.numb);   /*参数的实例号*/
            . . .
            val := val + rate.incr ;  /*参数的增量*/
            . . .
        ENDPROCESS p;
    ENDBLOCK b;
ENDSYSTEM s;
```

图 D-5.2.1

系统参数使用举例

SDL 还提供了两个辅助构件，以便以外部同义词为条件提供更强的选择：

— SELECT 构件：有条件地选择一段规格。条件为一布尔表达式所规定，该表达式必须在系统解释之前被静态求值，当表达式的值为 TRUE 时就选择这段规格。

— ALTERNATIVE 构造：在两段或更多可供选择的规格中，有条件地选择一段规格。它只能在进程、过程或服务体中用来选择不同的跃迁。

D.5.3 服务

D.5.3.1 概述

设置服务概念的目的在于把进程定义进行划分但又不引入并行的可能性。每个服务可被看作是由进程提供的一种"功能"，它是一个表示该进程"子行为"的进程定义的一部分。这样

的一个“子行为”是进程总行为的一部分。因此使用服务概念是构造进程的一种手段。

有种种理由需要构造一进程，例如控制复杂性和提高可读性。而划分也是一种用来孤立进程的某些部分并分别加以描述的手段，这样的一些部分可以是由系统提供的功能的“子部分”。因此运用服务概念，我们可以通过描述一个或多个进程中的一组从属的服务把一系统的功能分离出来。

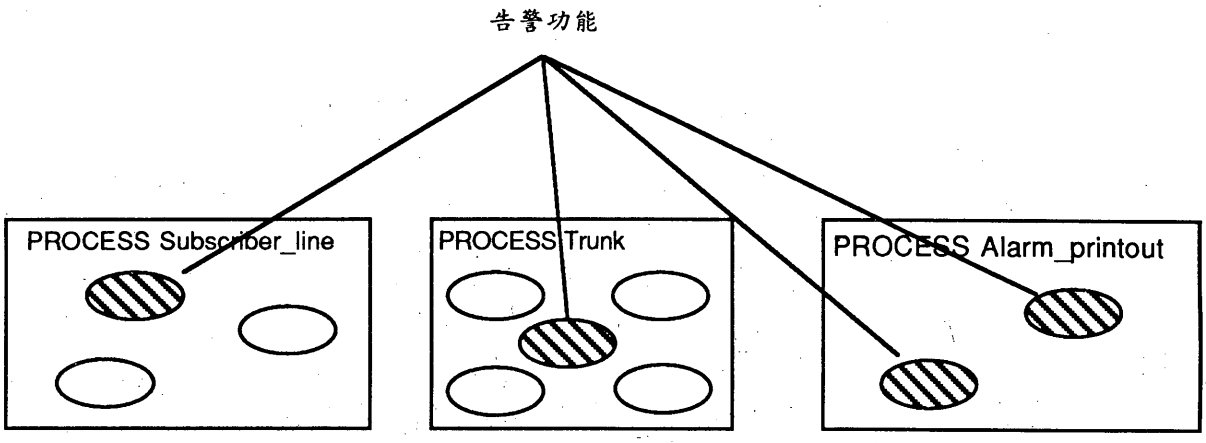


图 D-5. 3. 1

非形式举例说明不同进程中的服务如何能组成系统中的一高级功能

注意 SDL 语言没有任何从若干进程中选择一些服务以构成一系统功能的设施。这样的构成由用户负责，完全在本语言范围之外。

因为某一服务独立于其它服务，且被描述成拥有自己的状态空间的有限状态自动机，所以对它进行开发或更改不影响其它的服务。然而应当注意一进程中的各个服务经常有公共的数据，这意味着它们可以通过数据操作相互影响。

因为将进程行为划分成服务时并没有改变该进程行为，故这些服务只是同一行为的另一种描述。当然在设计时可以决定用几个而不是一个进程来模拟一个行为，然而由于若干进程的并行运行，可能会产生不同的行为，而且进程间如果不采用复杂的信号交换，就不能共享（读和写）数据。

注意用服务的方式来构造一个进程并不意味着进程将完全消失，它仅仅表示描述该进程行为的进程体完全为若干服务体所代替，而在进程级的各种形式参数、声明和定义将保留。除此之外在服务定义中可以包含若干局部的定义和声明。

一服务定义由下列的项（其中有些是任选项）组成：

— 服务名字

— 有效输入信号集：一张信号标识表。这些信号标识符是用来定义那些能为服务所接收

的信号的。

— 过程定义：局限于服务的各种过程的规格。过程之中还可以引用过程。
— 数据定义：局限于服务的用户定义的各种新类型、同义类型及生成程序的规格。
— 变量定义：局限于服务的各种变量声明。这些变量不能被透露或出口到其它进程（不允许使用关键字 REVEALED 和 EXPORTED）。对每个被声明的变量，必须指明它的类别标识符。可以任选地指定一初始值。
— 视见定义：声明一些变量标识符。使用这些变量标识符可以获得为其它进程所拥有的变量的值。对每个变量标识符必须指明变量类别。
— 进口定义：规定一些变量标识符，它们为其它进程所拥有，但又是本服务要进口的。对每个标识符，必须指明变量类别。
— 计时器定义：在§D.3.11中说明。
— 宏定义：在§D.5.1中说明。
— 服务体：用状态、输入、输出、任务等来表示服务的实在行为的规格。与进程体类似，服务体也能包含优先信号（§D.5.3.2）的发送与接收。

在 SDL/GR 中服务定义是用服务图来表示的。服务图由下列诸项组成：

— 任选框符号：一矩形符号，它包围所有其它符号。
— 服务标题：关键字 SERVICE 后面跟着服务标识符。服务标题安排在框的左上角。
— 任选页编号（放在右上角）。
— 正文符号：正文符号用来包围局限于服务的数据、变量、视见、进口和计时器定义。
— 过程引用标记：包含一过程名的过程符号，该过程名表示一个局限于服务并被单独定义的过程。
— 宏图：见指南§D.5.1。
— 服务图形：用状态、输入、输出、任务等表示的服务的实在行为的规格。与进程图形类似，服务图形也能包含优先信号（§D.5.3.2）的发送与接收。

图 D-5.3.2给出了一个简单的进程计时器作为服务概念的一个例子。该进程引用了两个服务，也可以说它是用两个服务构成的，图 D-5.3.3的两个服务图中引用并规定了这两个服务。

两服务之间有一内部接口，借助信号路由“IR6”传送一“优先”信号（§D.5.3.2）。

应该注意到，因为在第一个服务的启动跃迁中包含了动作（输出），所以在第二个服务的启动跃迁中不允许有动作。

服务概念的另一个例子在§D.10中给出。

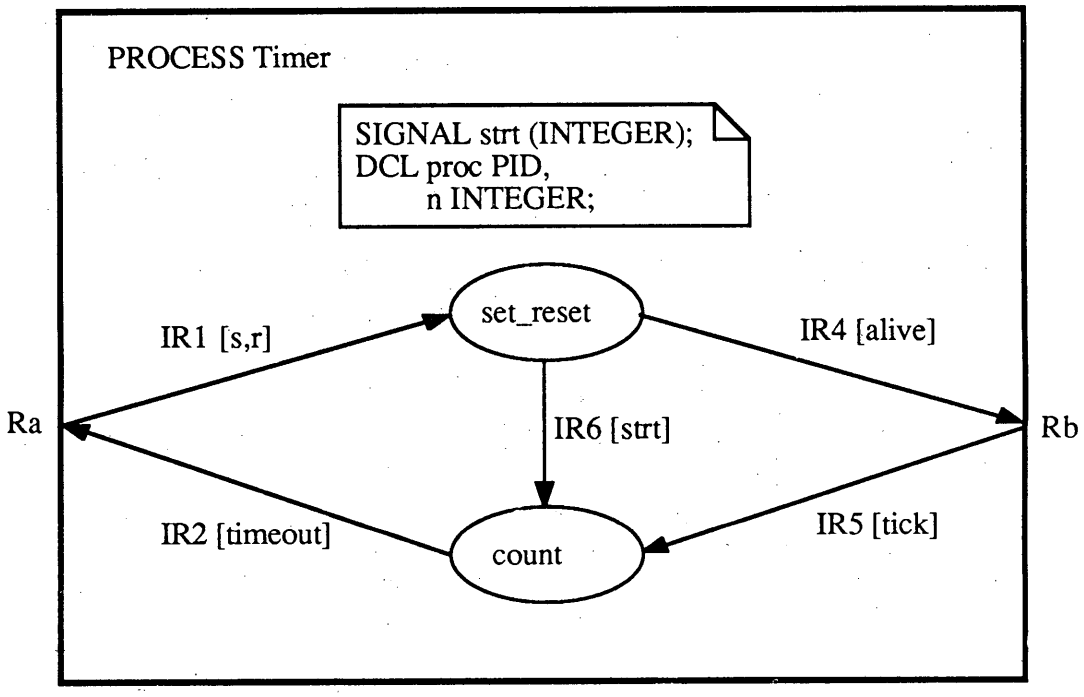


图 D-5. 3. 2

引用服务的进程图

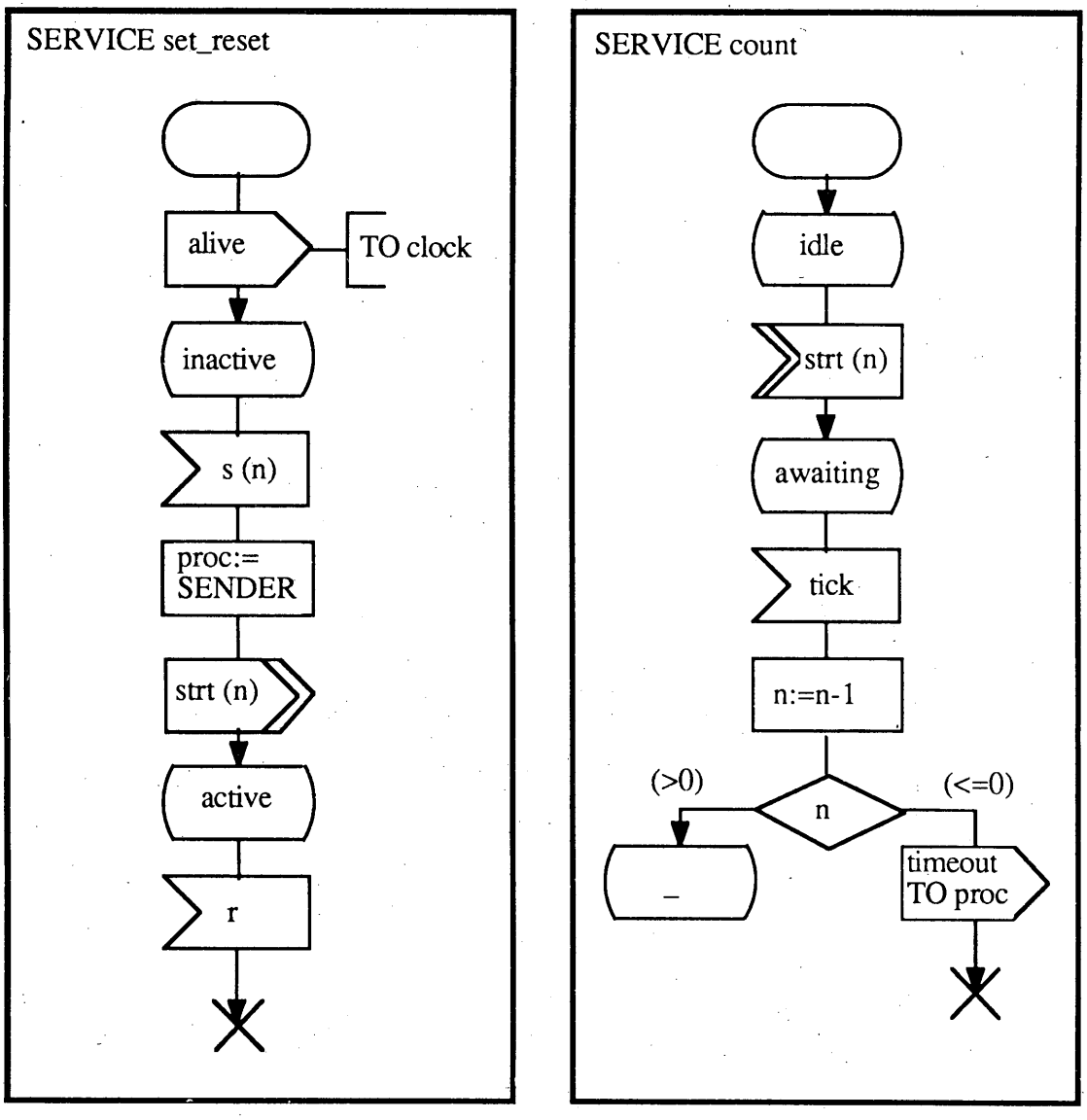


图 D-5. 3. 3

服务图

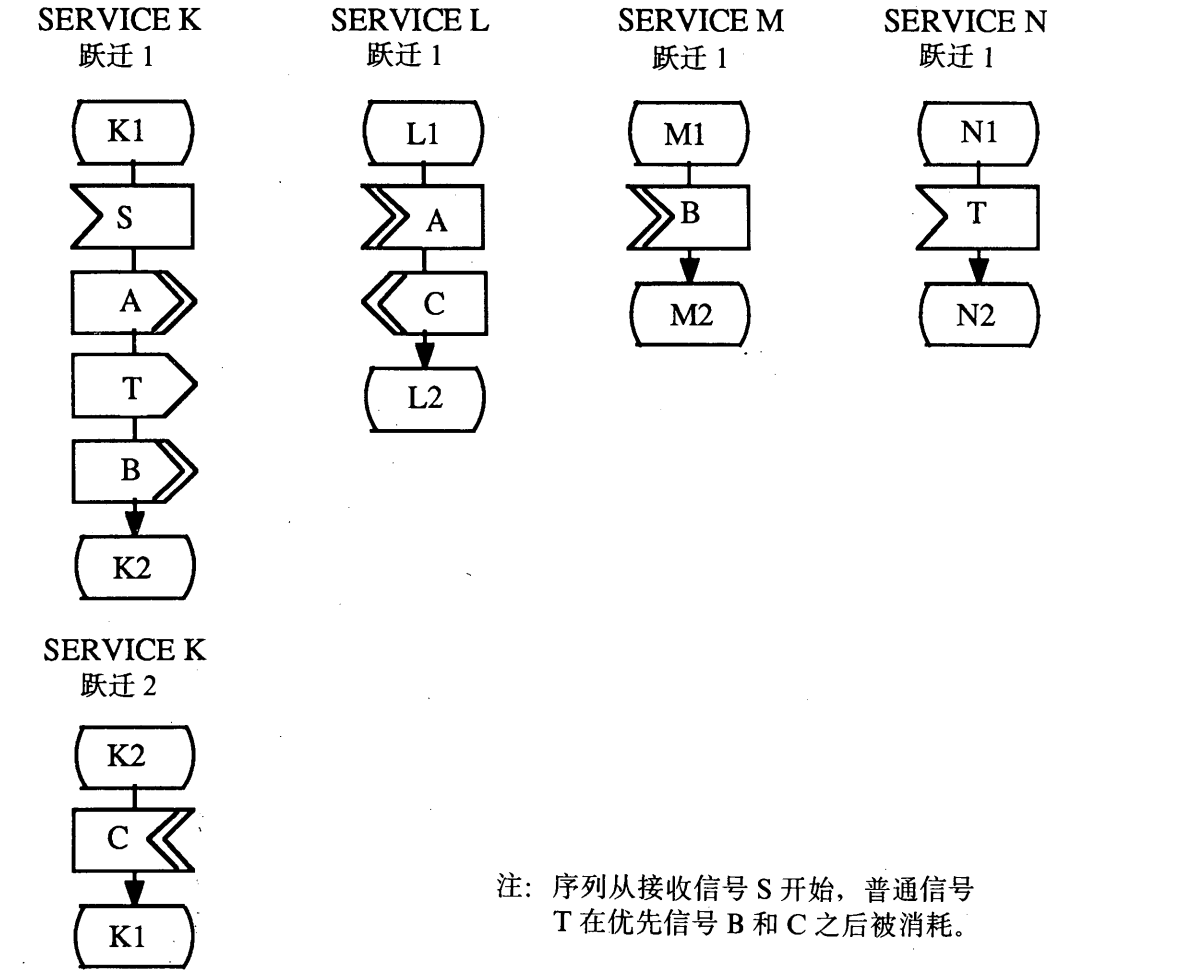
D. 5. 3. 2 优先信号

利用优先信号可在进程中属于不同服务的两个跃迁之间进行通信。在这两个跃迁之间的时间间隔内，不允许吸收（来自其它进程的）外部信号，这样跃迁就被“链接”起来。

下图（D-5.3.4）是在使用优先信号时，跃迁链接的图示。

建议中阐述了使用通常的输入队列获得优先信号结构。通过把优先信号发送到初始状态这一操作，就可将它们作为普通信号来处理，从而引起到主状态的各种跃迁（见 D-5.3.3.1）。

下面的例子用图表示了使用优先信号的结果。在阐述该例时没有使用“初始/主状态”模型。



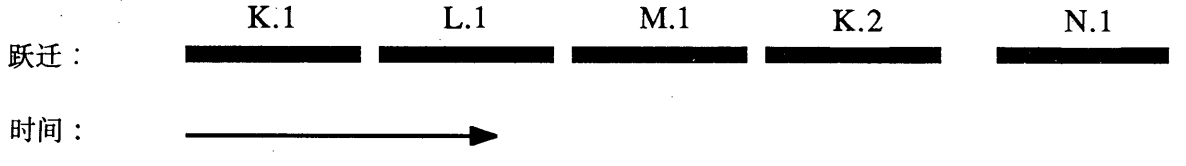


图 D-5.3.4

在一进程中属于不同服务的跃迁的链接（非形式 SDL）

顺序图（图 D-5. 3. 5）是从进程“SUBSCRIBER_LINE”（在§D. 10. 2中说明）中的各服务之间的交互作用中抽取出来的。

该图既包含发送至（或来自）进程环境的信号，又包含服务之间的优先信号。优先信号箭头是较粗的，信号箭头的顺序（自顶向下）指明信号在队列中是如何排列的，竖粗线则指明跃迁的执行在时间上是如何安排的。

序列从来自寄存器的信号“CONGESTION”开始启动，表明应拒绝呼叫。这也意味着下次呼叫也应立即拒绝。

该图指出，由一优先信号例如“RESERVE_FOR_MEASUREMENT”激活的跃迁要比由一通常信号，例如“A_ON_HOOK”激活的跃迁启动得更早，即使它们的队列顺序相反。由信号“RESERVE_FOR_MEASUREMENT”激活的跃迁切断用户线，这表示下一次呼叫（信号“A_OFF_HOOK”）将被立即拒绝。

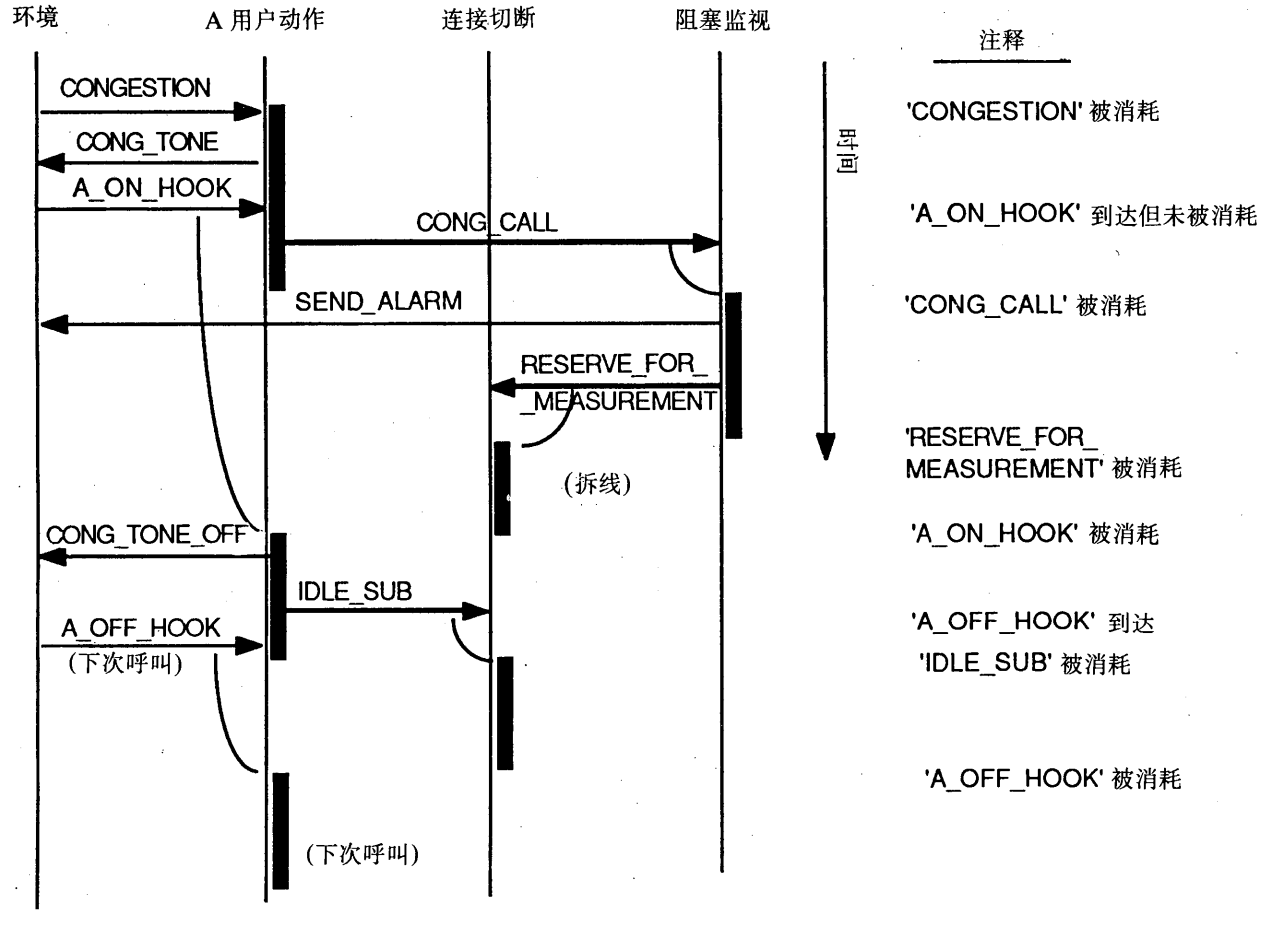


图 D-5. 3. 5

表示优先信号如何改变跃迁顺序的顺序图

（每根粗竖线表示执行一次跃迁）

在下一顺序图中情况相同，但图中没有使用优先信号，各个服务用普通信号互相配合工作。

与图 D-5.3.5比较，可以看出跃迁的顺序已经改变，下次呼叫在用户线被切断之前到达，这意味着该呼叫将不被立即拒绝。

D.5.3.3 变换

服务和优先信号是辅助的概念，因为它们是由更基本的 SDL 概念（如进程和信号）构成的。为了能够从语义上对服务和优先信号进行解释，必须把它们变换回为这些基本概念。通常这样的变换不必由用户来完成，至少不必用人工来完成，当然用户必须了解这种变换。在服务的语义检验工具中能自动完成这种变换。

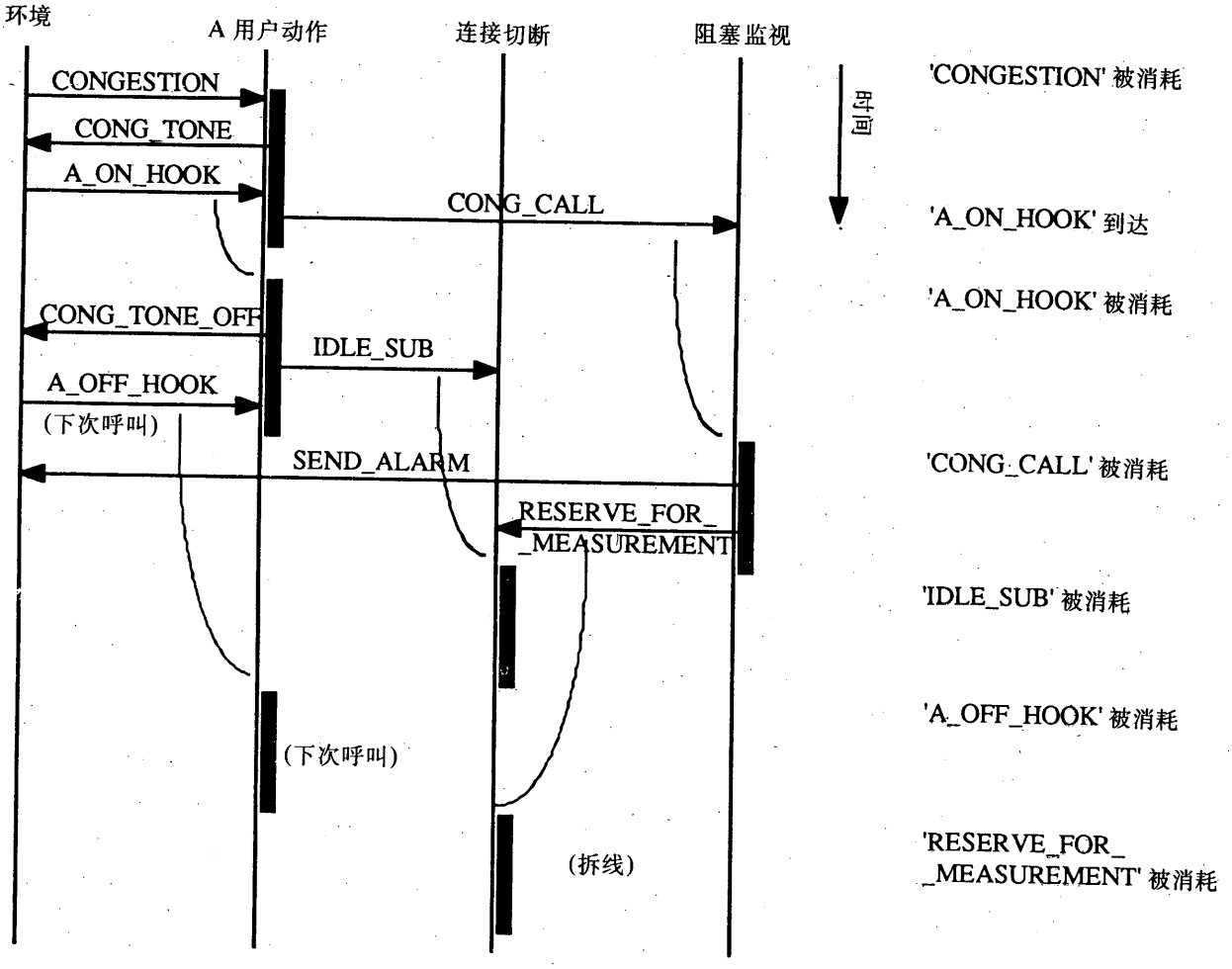


图 D-5.3.6

具有普通信号和正常跃迁顺序的顺序图

在变换之后，各个服务消失并为一从语义上解释的扩展进程定义所替代。该变换已对行为

作出了规定。

D.5.3.3.1 状态变换

在建议中给出了状态变换的专门规则。

这些规则的结果是进程中状态的数目是各服务中状态数的乘积。

除此之外，使用优先信号将使该乘积加倍，因为每个被变换的状态分裂成一初始状态和一主状态。在下面状态变换的例子中不涉及这种由于优先信号所造成的状态数加倍的情况。

在许多情况下，“被变换”进程中的许多状态是不相关的，同时可以减少状态空间。在§D.10.2的例中讨论了这种情况。

在进程“SUBSCRIBER_LINE”中，有四个服务，即“A_subscriber_actions”、“B_subscriber_actions”、“Connection.Disconnection”和“Congestion_Supervision”，状态数为10，5，3和2，即总数为20。

把状态变换的规则应用到这些服务上，在进程中将有300个状态（10 * 5 * 3 * 2），这意味着有300个名字元组，元组的维数为4（4个服务），元组中的位置按下图（D-5.3.7）安排。

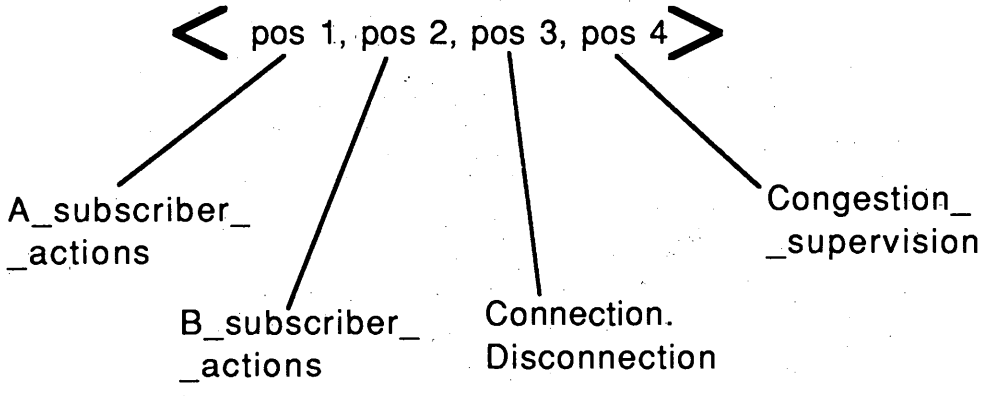


图 D-5.3.7

名字元组中位置的安排

如果在不同位置上的所有可能的状态名字全部出现的话，就可以得到300个组合。

例：

Ex. <A_IDLE, B_IDLE, CONNECTED, NO_ALARM>
<AWAIT_CONN, B_IDLE, CONNECTED, NO_ALARM>
<AWAIT_FIRST_DIGIT, B_IDLE, CONNECTED, NO_ALARM>
.....
<AWAIT_A_ON_HOOK_2, B_IDLE, CONNECTED, NO_ALARM>

<A_IDLE, B_RINGING, CONNECTED, NO_ALARM>
<A_IDLE, B_CONVERSATION, CONNECTED, NO_ALARM>
.....
<A_IDLE, AWAIT_B_ON_HOOK, CONNECTED, NO_ALARM>
.....
.....
<AWAIT_A_ON_HOOK_2, AWAIT_B_ON_HOOK, SEIZED, ALARM>

这些组合中有许多是不合适的，因为在同一呼叫中A用户和B用户二者永远不会使用一条用户线。这意味着‘A_subscriber_actions’中的10个状态只能与‘B_subscriber_actions’的一个状态（’B_IDLE’）组合，同样‘B_subscriber_actions’中的5个状态只能与‘A_subscriber_actions’中的一个状态（A_IDLE）组合，于是合适的状态数为（10 * 1 * 3 * 2）+（1 * 5 * 3 * 2）=90。

如上例所述，状态空间的减少取决于具体的例子，而不能作为通用规则应用于自动变换。然而，如果变换由人工来完成，则状态空间很可能会减少。为了比较两种设计方案（即：不采用服务设计进程，或把进程划分为服务来设计）的复杂性，应当采用状态空间已减少了的进程来比较，才能得到合理的比较结果。在大多数情况下，使用服务可大大减少状态的数目。

D.5.3.3.2 跃迁的变换

在建议中给出了跃迁变换的专门规则。下面是变换的一个例子，例中的进程是一个简单协议的规格，该进程被划分成两个服务：发送（Send）和接收（Receive）。

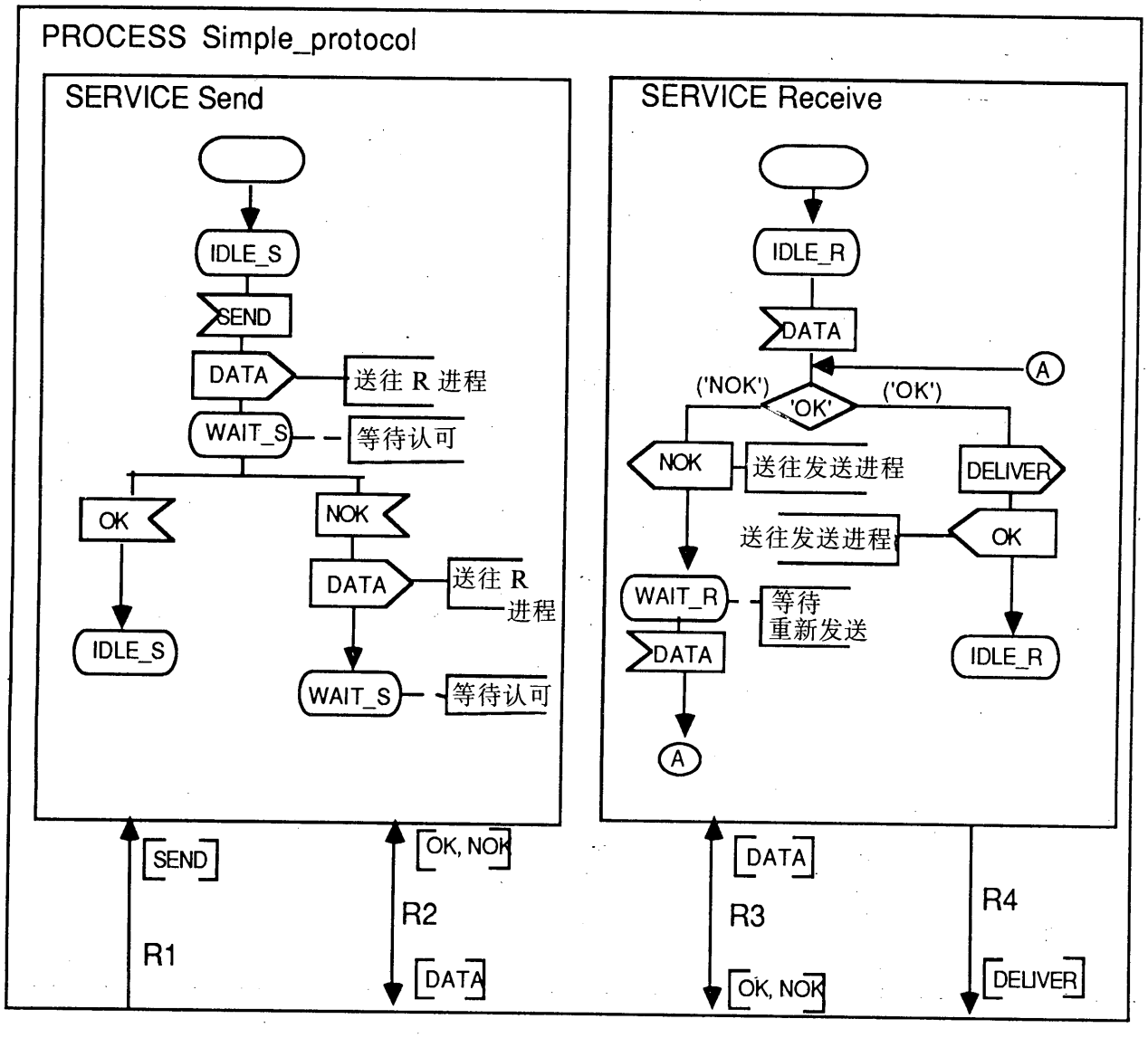


图 D-5. 3. 8

包含两个服务图的进程图

下图（D-5. 3. 9）是这两个服务的状态概览图，跃迁按1-7编号。

把形式化规则应用于状态和跃迁的变换，其结果对上述的进程形成下面的状态概览图。由于没有使用优先信号，因此就不需要额外地把状态分离成初始状态和主状态。

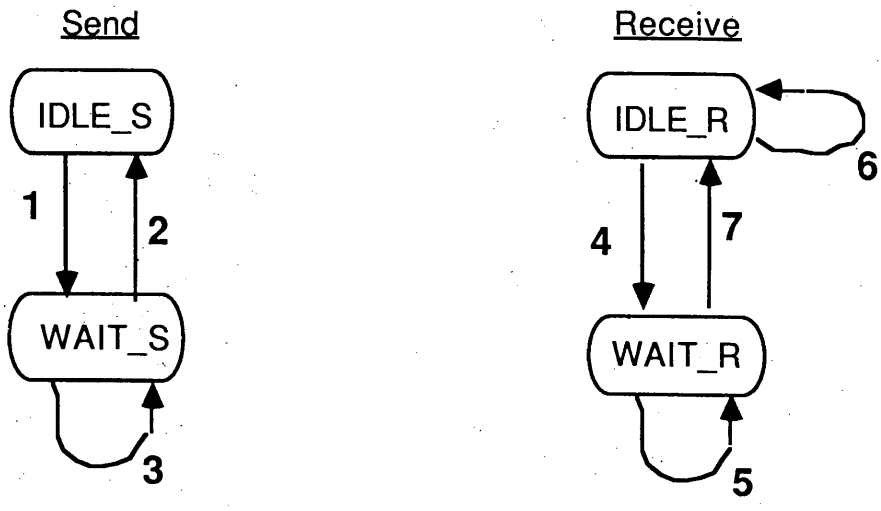


图 D-5. 3. 9

具有编号的跃迁的状态概览图

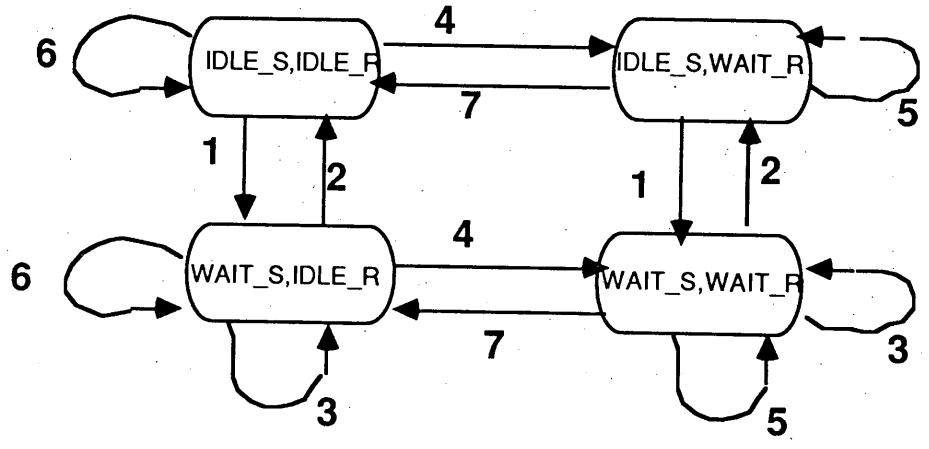


图 D-5. 3. 10

经变换的服务状态概览图

图 D-5. 3. 11表示了进程图形，不能减少图中的状态数目。进程图形中的状态名与图 D-5. 3. 10中的名字元组对应。例如：IS. IR 对应于 IDLE_S，IDLE_R。

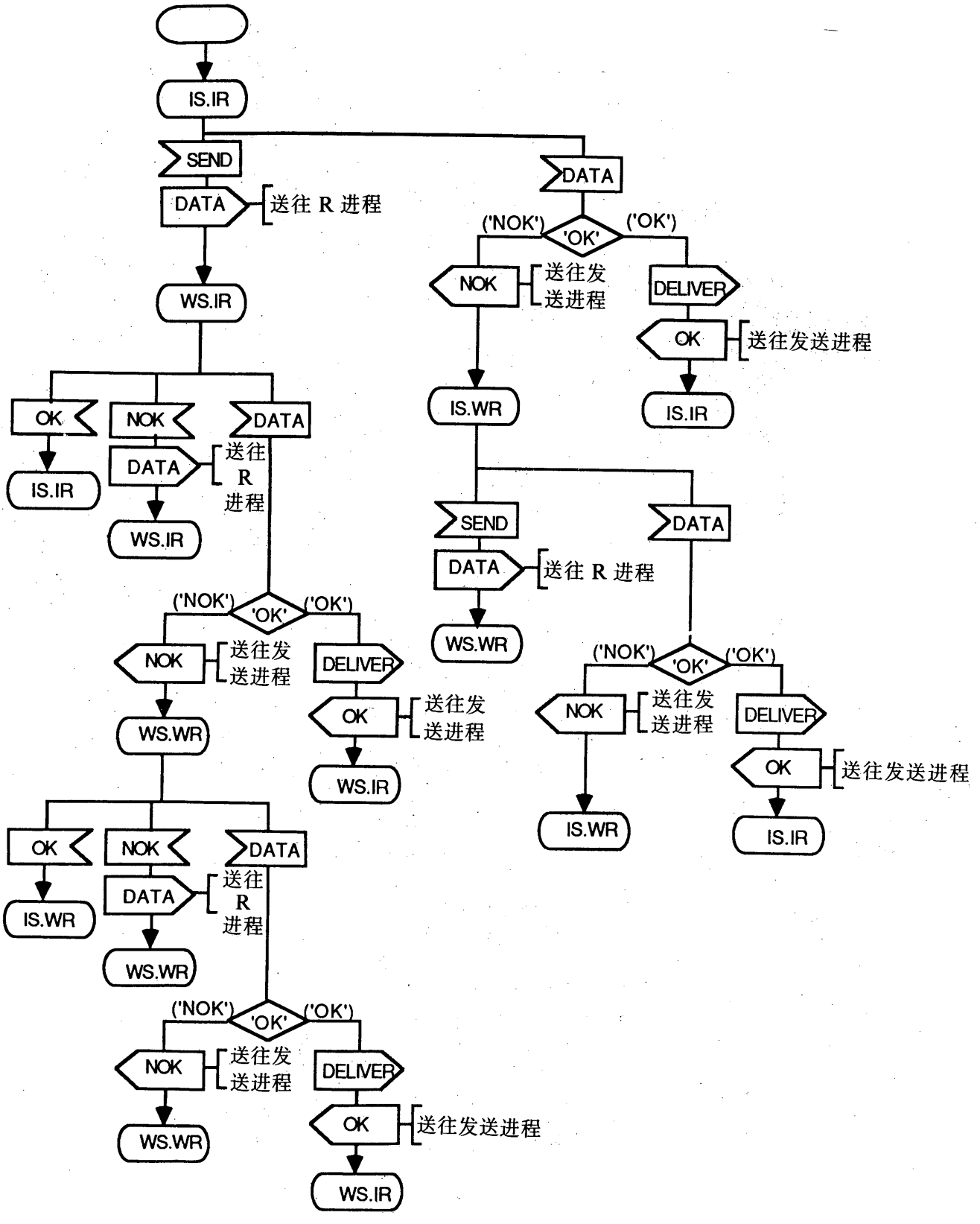


图 D-5. 3. 11

由两个服务图经变换得到的进程图

SDL 建议 - 附件D：用户指南 - § D. 5

D.5.4 面向状态表示和图形元素规则

本章阐述 SDL/GR 的面向状态表示法和图形元素。

第一节（D.5.4.1）是关于面向状态表示法及其特性、适用场合和变化的总说明。

第二节（D.5.4.2）说明了如何用图形元素来描述状态。

D.5.4.1 关于面向状态表示法的总的说明

SDL/GR 给出了描述进程图的三种不同形式，可选其中之一。

第一种叫做面向跃迁的 SDL/GR 形式，在这种形式中，跃迁完全用显式的动作符号来描述。

第二种叫做面向状态的 SDL/GR 形式，或叫做面向状态的 SDL 图形扩展形式。在这种形式中用状态图形来描述一个进程的多个状态，且仅靠始发状态和终止状态之间的差别就可隐式地表示跃迁。

最后一种称为组合形式，它是上述两种形式的组合。

这三种形式的举例在本卷的附件 E 中给出。

当动作的序列重要，而详细的状态说明不那么重要时，面向跃迁的形式是合适的。

面向状态的形式明确而详细地描述状态，因此它适用于下列情况：当一进程状态比每个跃迁内的动作序列更重要时，当人们希望用直观的图形来阐述时，以及当人们关心的是各种资源和它们与状态的关系时。

状态图形通常用各种图形元素来构成。这些图形元素指明了与进程当前状态有关的资源。在某些应用中，如果已经定义了合适的图形元素，那就可以采用状态图形。因此如果需要的话，用户可以定义合适的图形元素，从而把这种表示形式应用到各种场合中。

当既需要考虑在每个跃迁内的动作序列又需要考虑详细的状态描述时，就适合采用组合形式。

D.5.4.2 状态图形和图形元素

D.5.4.2.1 图形元素和限定正文

如果选用状态图形，那么一个状态图形就由图 D-5.4.1 a）至 d）所示的图形元素和限定正文组成。

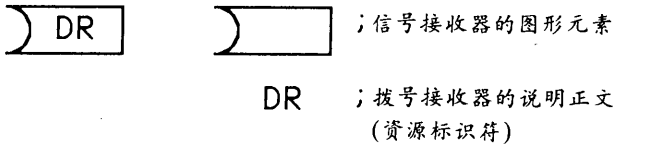
这种组合使得状态图形易于理解。例如，图 D-5.4.1a）给出了一个受进程控制的拨号接收器的含义，例 b）给出了一个正在向环境发送一固定信号的拨号音发送器的含义。

注意输出信号（非固定信号）和有关资源不在状态图形中描述，输出信号可以在跃迁图中说明。

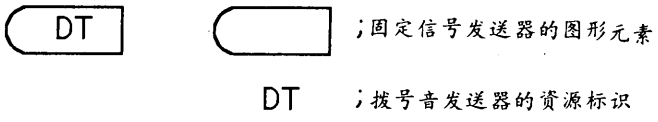
例 c）表示了一个计时器，用一输入表示其计时终了。应注意，对计时器所建议采用的图形元素包括了有关的输入信号 t1。

最后一例 d）表示一声音和信息记录机正在记录。

资源标识应尽量简略，并且应尽可能置入相应的图形元素中，这样就很明显地表达了所限定的是哪一个图形元素。



a) 信号接收器



b) 固定信号发送器

c) 计时器

d) 控制元素

图 D-5.4.1

带有说明正文的图形元素举例

D.5.4.2.2 状态图形的完整性

在每个状态图形中，应当安排足够的图形元素和限定正文，以便指明：

a） 处于当前状态，该进程与哪些客体或资源有关。资源诸如交换通路，信号接收器，固定信号发送器以及交换模块等等；

b） 当前是一个还是多个计时器在监视该进程；

c) 在该进程涉及呼叫处理时，当前用户呼叫计费是否在进行，以及在呼叫的这一状态期间，哪一个用户负担费用；
d) 在当前状态期间，为另一进程（环境）所拥有的哪些客体与该进程的资源有关；
e) 在该状态期间被输出的输出固定信号；
f) 在该状态中信号与资源的关系；
g) 与进程当前状态有关的各种资源的状态。

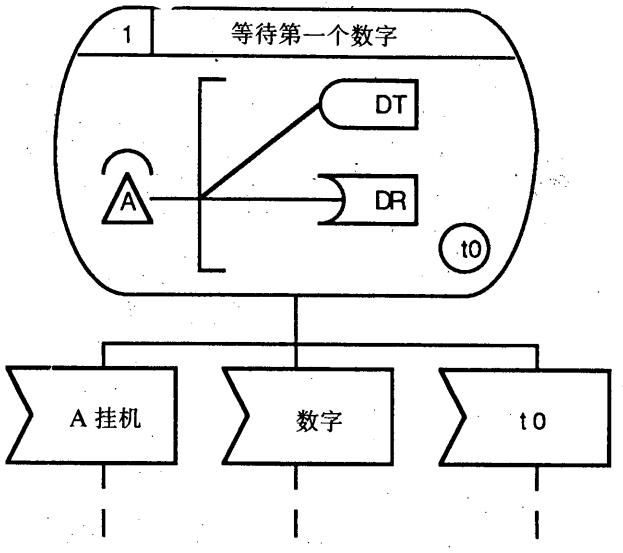


图 D-5. 4. 2
呼叫处理进程中的一个状态举例

D. 5. 4. 2. 3 举例

图 D-5. 4. 2中用状态图形来举例说明如何应用上述原则。可以看出在该状态期间：
a) 进程所涉及的资源由拨号数字接收器、拨号音发送器、属于环境的用户话机和连接这些设备的交换通路组成；
b) 计时器 t0正在监视该进程；
c) 没有进行对此呼叫的计费；
d) 标识用户为 A 方，但不考虑其它方面的信息；
e) 等待下面的输入信号；A_on_hook（即送受话器联机信号），digit（即拨号数字）和 t0（即监视计时器 t0在运行）；

f） 输出固定信号 DT（即拨号音），在进入此状态之前已开始输出 DT。

D. 5. 4. 2. 4 具有图形元素的 SDL 图的一致性检验特性

很明显，状态图形使读者感到更加简洁明了，且在一定意义上使读者能看到更多的信息；但同时人们又必须仔细地查看资源，才能准确地知道在跃迁中所执行的那些动作。

此外，单凭查看状态图形不可能知道跃迁中各种动作的顺序。

在功能块边界符号外部所示的图形元素意指那些不受给定进程直接控制的元素，而在功能块边界符号内部所示的图形元素意指那些受给定进程直接控制的元素。例如，图 D-5. 4. 3中所说明的部分呼叫进程，能够分配或重新分配回铃音发送器、振铃发送器和话音通路，以及启动或停止计时器 T4，但是它不能直接控制用户的送受话器状态。

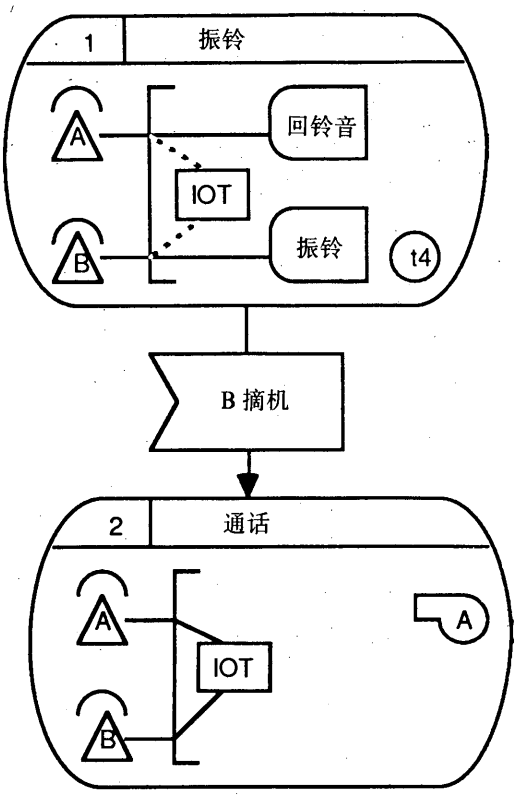


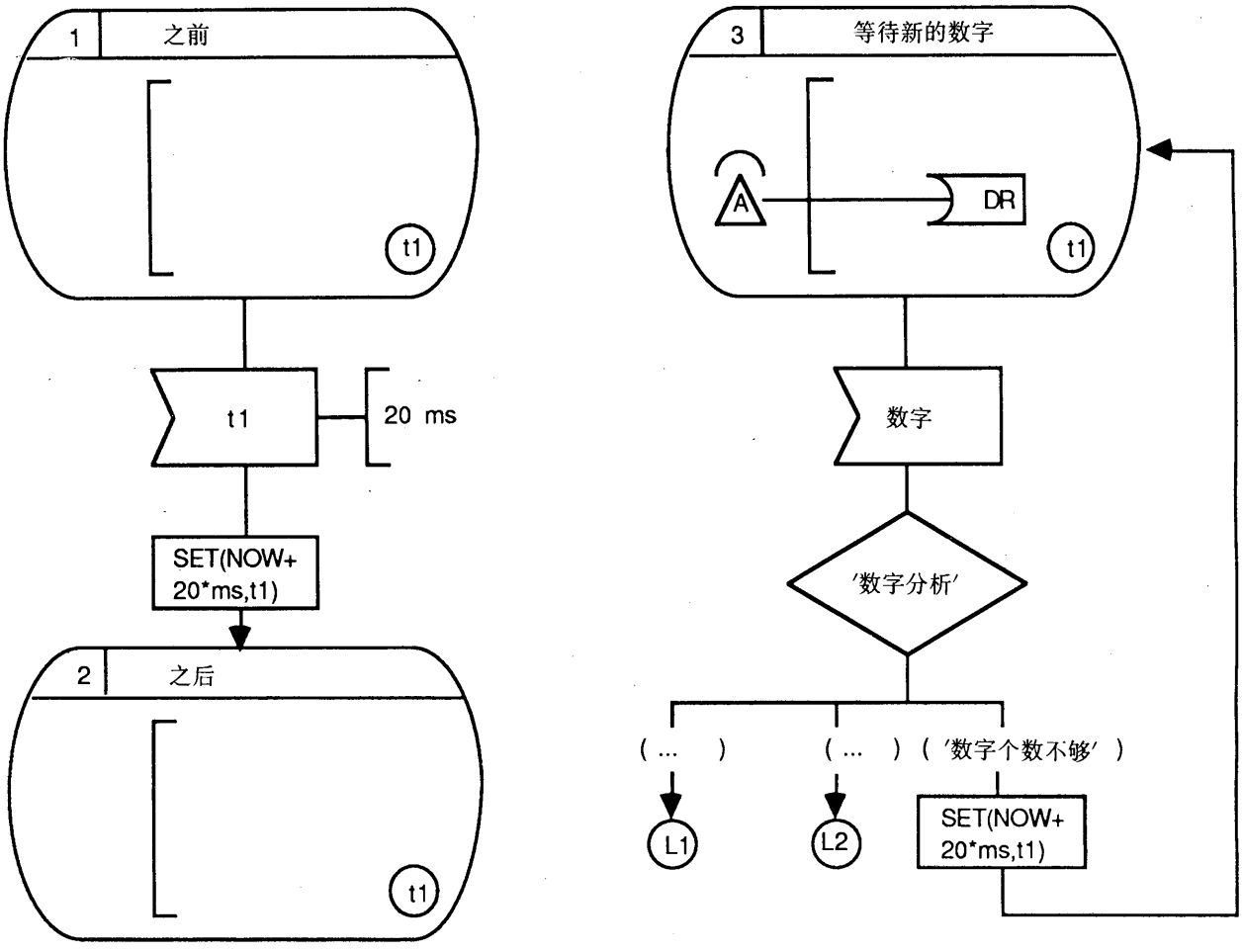
图 D-5. 4. 3

两个状态间的跃迁举例（状态图形的差别隐含了跃迁发生时的所有处理动作）

根据具有图形元素的 SDL 规格进行逻辑设计时，只有画在功能块边界内部的那些图形元素才会影响跃迁序列中执行的处理动作。但功能块边界外部所示的图形元素，通常也被包含在状态图形中：

a） 因为它们指明了在给定的状态期间与进程的输入信号有关的资源和环境的状态；

b） 改善了图的可懂度。



a) 在计时器计时终了后，
计时器的重新启动

b) 计时器尚未计时终了时，
计时器的重新启动

注–每个计时器 t1 有两个互斥的状态：活跃的和静止的。

图 D-5. 4. 4

计时器的重新启动

D. 5. 4. 2. 5 计时器符号的使用

不管是否使用图形元素，在计时器计时终了时总是要用一输入来表示。

在一状态图形中，计时器符号的出现意味着在该状态期间内有一计时器正在运行。

按照在建议中叙述的一般原则，启动、停止、重新启动和计时器计时终了是按下列方法用

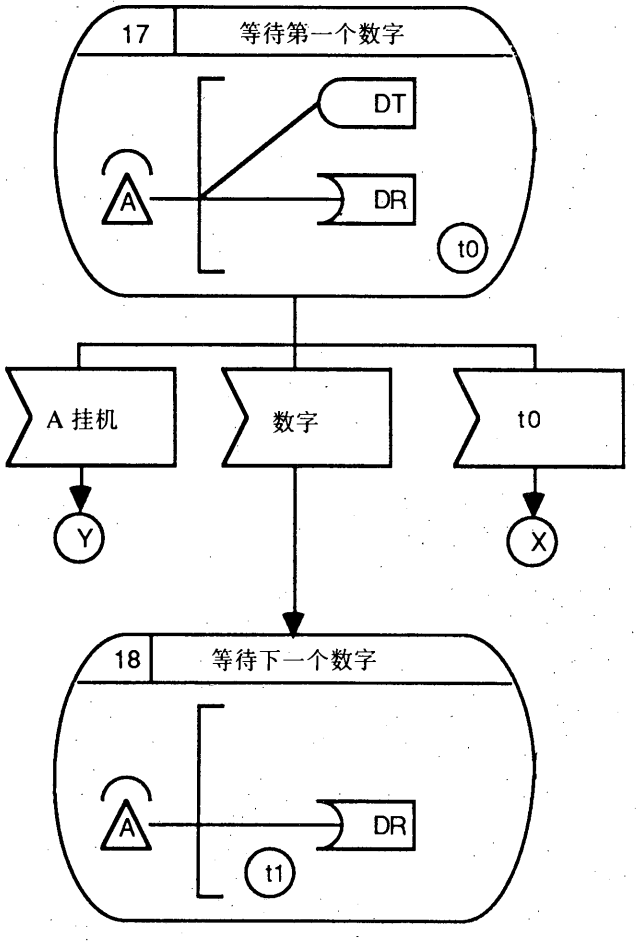
图形元素表示的：

a）如要表示出在一给定跃迁期间计时器的启动，则该计时器符号应出现在该跃迁结束的状态图形中，而不是出现在该跃迁启动的状态图形中。

b）如要表示出在一跃迁期间计时器的停止，则该计时器符号就应出现在该跃迁启动的状态图形中，而不是出现在该跃迁结束的状态图形中。

c）如要表示出在一跃迁期间计时器重新启动，则应在该跃迁中安排一个显式的任务符号（图 D-5. 4. 4举了两个例子）。

d）一特定计时器的计时终了可通过与一状态（在该状态的状态图形中包含相应的计时器符号）相关联的一个输入符号来表示。当然如果需要的话，多个计时器可以同时监视同一进程，如图 D-5. 4. 5所示。



注：计时器 t0 监视第一个数字的到达，随后从呼叫中除去拨号音，且计时器 t0 停止。计时器 t1 继续监视足够的数字个数的到达，以便为该呼叫选择路由。

图 D-5. 4. 5

在同一状态中使用两个监视计时器举例

D.5.5 辅助图

为了帮助阅读和理解庞大的和（或）复杂的进程图，作者可以提供一些非形式的辅助图，这些文件的目的在于对进程的行为（或它的一部分）给出一个概括的或简单的说明。辅助文件并不能代替 SDL 文件，而只能用来介绍 SDL 文件。

在本节中给出一些通常使用的辅助图的例子，也就是状态概览图、状态/信号矩阵、顺序图（§D.4.4中描述的功能块树图也是一个辅助图）。

D.5.5.1 状态概览图

状态概览图的目的在于给出一个进程的各个状态的概括描述以及它们之间可能有的一些跃迁。因为仅仅是一概括性的描述，故可以把那些“不重要的”的状态或跃迁省略掉。

该图由状态符号、表示跃迁的有向弧以及可以任选的启动和停止符号组成。

状态符号内应写明所涉及到的状态的名字。状态符号也可以包含多个状态名，并可使用一星号（*）作为代表全部状态的符号。

对每条有向弧除了可以标注引起跃迁的那个信号（或一组信号）的名字外，也可以注明在跃迁期间可能发出的各种输出。

图 D-5.5.1给出了一个状态概览图的例子。

一个进程的状态概览图可分解成若干个图，每个图涉及不同的方面，例如“正常情况”，“故障处理”等。

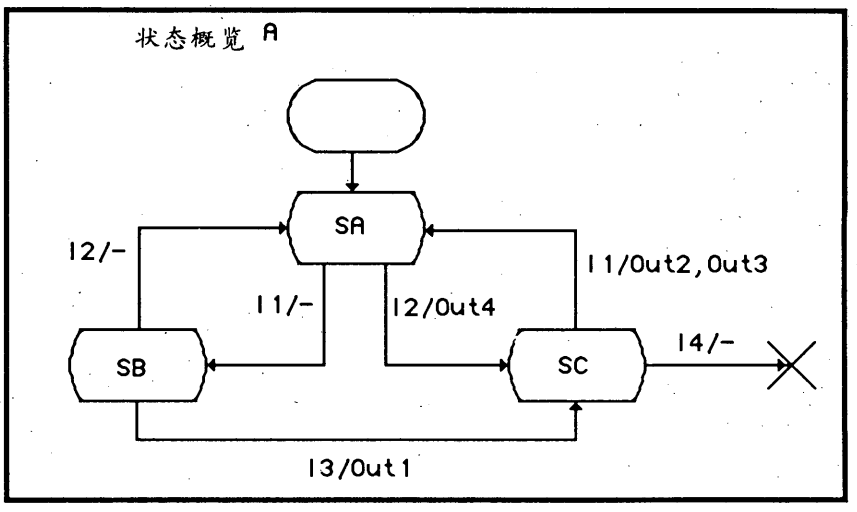


图 D-5.5.1
状态概览图举例

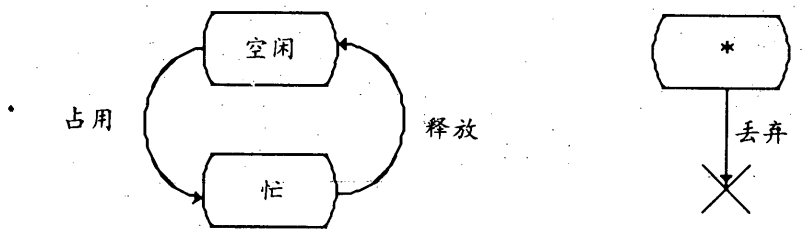


图 D-5.5.2

分开的状态概览图举例

D.5.5.2 状态/信号矩阵

状态/信号矩阵可以替代含有完全相同信息的状态概览图。

该图由一个二维矩阵组成，其中一维由这个进程的所有状态来索引，而另一维由这个进程的所有有效输入信号来索引。在每个矩阵元素中，给出下一状态和跃迁期间可能的输出，如果存在元素，则利用二维索引可以引用相应的矩阵元素。

对应假状态“START”和假信号“CREATE”的矩阵元素是用来指明进程的初始状态的。

这种矩阵也可分解为子矩阵，以便放到其它的页上，对这些页的参照办法和在文件编制中用户所通常使用的参照办法相同。

最好各信号和各状态应分群地组合在一起，以便矩阵的每个子矩阵包括进程行为的一个方面。

状态／信号 A

状态 / 信号	"START"	SA	SB	SC
"CREATE"	SA/-	-	-	-
I1	-	SB/-	-	SA/Out2,Out3
I2	-	SC/Out4	SA/-	-
I3	-	-	SC/Out1	-
I4	-	-	-	"STOP"/-

图 D-5.5.3

状态/信号矩阵举例

D. 5. 5. 3　顺序图

顺序图可以用来表示所容许出现的信号顺序，这些信号在服务、进程、功能块和它们的环境之间互换。

采用顺序图是为了对系统各部分之间的信号交换有个概括的了解。可以用它表示信号交换的全过程或部分过程，这取决于要突出的是哪些方面。

图中的竖线表示各种通信实体（服务、进程、功能块或环境）。

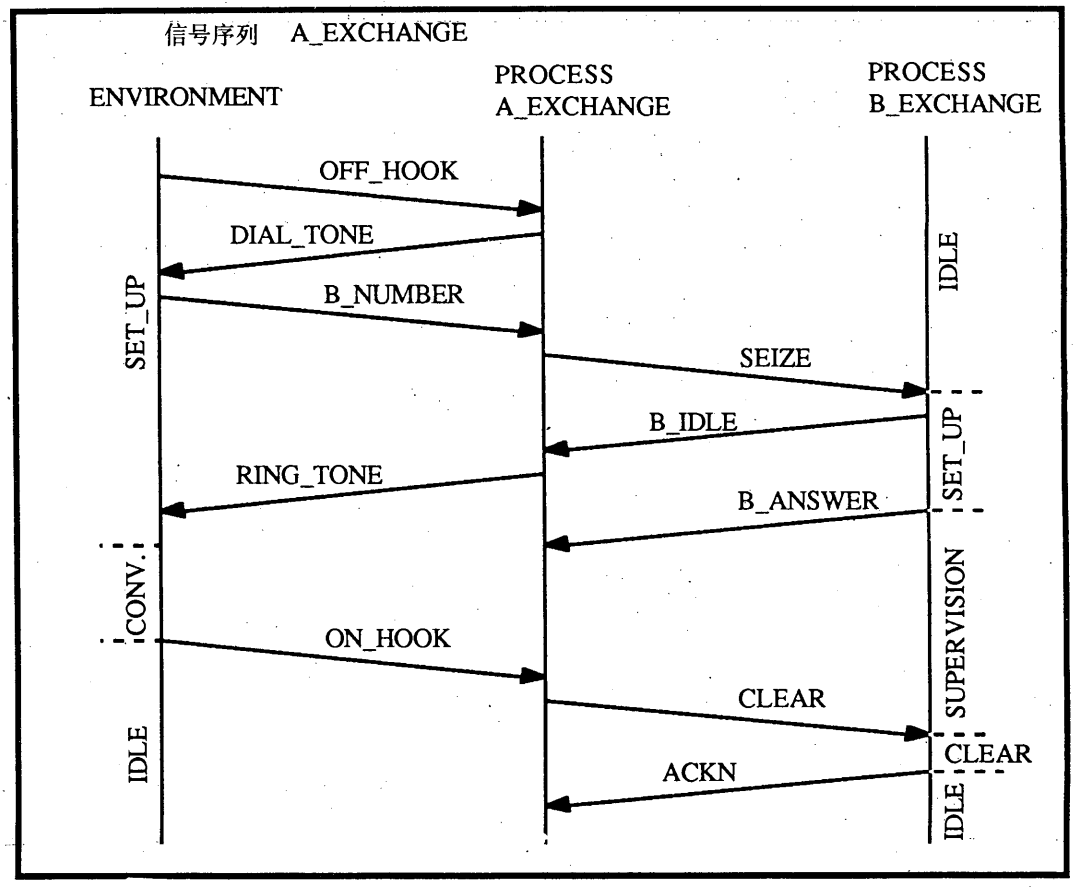


图 D-5. 5. 4

顺序图举例

它们的相互作用是用一组有向线来表示的，每根有向线代表一个信号或一组信号。

为了使人们能看清所交换的是一组什么信息，可以在一序列信号旁边加注。并在每根信号线旁注明所包含的信息（信号名或过程）。

在各竖线上可以安放一个判定信号，用以指出如果指定的条件为真，则随后的序列有效。通常在这种情况下，该判定信号出现若干次，以指出由不同的条件值引起的不同序列。

这个图能够完整地表示交换信号的所有序列，或者只是表示其中的一个有意义的子集。

这种图可有效地表示由划分一进程所引起的各服务间的相互作用。

顺序图通常不涉及所有可能的序列；相对于完整的定义而言，它们往往是初步的。

D.6 SDL 数据定义

D.6.1 SDL 中关于数据的准则

本节对在 SDL 建议的 §5中定义的概念给出了更多的信息。该节与早先的 Z.104之间的主摇—病☆别是在阐述方面及与 ISO 的协调方面更具体。对有些关键字做了修改并在某些方面作了扩充，但在语义上与红皮书原定的意义是一致的，在新建议 §2、§3、§4中关于数据的使用（先前的 Z.101-Z.103）没有改变。

D.6.1.1 一般介绍

处理 SDL 的数据类型所依据的方法是“抽象数据类型”，这种方法不去描述一个类型必须如何（HOW）被实现，而只是当用它来操作各种数值时，可被告之运算的结果是什么（WHAT）。

在定义抽象数据时，称定义中的每个段为“部分类型定义”，它是由关键字 NEWTYPE 引入的。每个部分类型定义影响其它的一些部分类型定义，以使得在系统层的所有部分类型定义构成一个唯一的数据类型定义。如果在一较低层（如在功能块层）再引入部分类型定义，则它们与较高层定义一起构成一数据类型定义，也就是在规格中的任何位置仅有一个数据类型定义。

数据类型定义基本上由三部分构成：

a） 类别定义；

b） 运算符定义；

c） 方程。

下面对每个部分进行阐述。数据类型定义是用部分数据类型定义来构造的，每个部分数据类型定义引入一个类别。各种运算符定义和方程分布在各个部分数据类型定义内。

D.6.1.2 类别

类别是多个值的一个集合，该集合可以有一有限或无限数目的元素，但不能为空。

例如：

a） 布尔类别的值的集合为{True,False}。

b） 自然数类别的值的集合是无限多个自然数的集合{0,1,2,…}。

c） 基色类别的值的集合为{Green,Red,Blue}。

一个类别的每个元素不需要由用户直接提供(在自然数的情况下,这样做需要无限长的时间),但必须给出类别名字。在具体的语法中,类别名字直接跟在关键字 NEWTYPE 的后面(有些另外的写法以后会介绍)。如在§D.6.1.3和变量声明中所述的，该名字主要用于运算符定义。

D.6.1.3 运算符、字面值和项

一个类别值可用三种方式定义：

a） 枚举法；在字面值节内定义各个值。

b） 采用运算符；“应用”运算符产生的结果作为值。

c） 枚举法和采用运算符相结合。

字面值和运算符组合起来产生项。各项间的关系用方程表示。下面各节介绍字面值、运算符和项；§D.6.1.4介绍方程。

D.6.1.3.1 字面值

字面值是一个类别的枚举值。部分类型定义可以不包含字面值;一个类别中的全部元素可以通过运算符来定义。可以把字面值看成是不带变元的运算符,字面值之间的关系可以用方程表示，在具体的语法中，字面值在关键字 LITERALS 之后引入。

例：

a） 布尔类别的定义包含两个字面值，即真（True）和假（False），因此布尔类型的定义为：

```
NEWTYPE Boolean
    LITERALS True, False
    ...
ENDNEWTYPE Boolean;
```

b） 自然数类别可用一字面值0来定义，其它的值可用该字面值和运算符来产生。

c） 基色类别的值可以用类似于布尔字面值的方式来定义：

```
NEWTYPE Primary_colour
    LITERALS Red, Green, Blue
    ...
ENDNEWTYPE Primary_colour;
```

d) §D.6.1.3.2中第三个例子（C）表示一个没有字面值的部分类型定义。

D.6.1.3.2 运算符

运算符是一种把一个或多个值（可能来自不同的类别）映射到一个结果值的数字函数。运算符在关键字 OPERATORS 之后引入，它们的变元的类别和结果的类别也都要表示出来（这称作运算符的特征）。

例：

a) 在布尔类型中，可以定义一个称之为“not”（非）的运算符，它具有一个布尔类别的变元，也有一个布尔类别的结果值。在布尔类型的定义中，它表示如下：

```
NEWTYPE Boolean
    LITERALS True, False
    OPERATORS "Not": Boolean -> Boolean ;
    ...
ENDNEWTYPE Boolean;
```

b) 在上节涉及到的构造所有自然数所需要的运算符是 Next。该运算符用一个自然数类别的变元，并给出（下一个较大的）自然数数值作为结果。

c) 对于各种颜色，可以定义一个新类型。它没有字面值，但使用了基色类别的字面值和一些运算符：

```
NEWTYPE Colour
    OPERATORS
        Take: Primary_colour -> Colour ;
        Mix : Primary_colour, Colour -> Colour ;
    ...
ENDNEWTYPE Colour;
```

其意图是取一基色并把它作为一种颜色，然后开始把另外的基色混合进去，以得到更多的颜色。

D.6.1.3.3 项

对每一类，使用字面值和运算符可以构造项的集合，如下所述：

1. 把该类别中所定义的所有字面值，收集到一个集合中，其中每一字面值是一个项。
2. 对每个运算符，把它应用于原先各项的所有可能的组合上，可以建立一组新项。原先各项即

指属于该类别的原先已建立的那些集合的所有的项。

a) 对布尔类，字面值的集合是{True,False}，因为只有运算符 Not，所以这一步的结果是{Not(True),Not(False)}。

b) 对于自然数类别，该步的结果是{Next(0)}。

c) 对颜色类别，字面值的集合为空，但该步的结果是{Take(Red),Take(Green),Take(Blue)}。

3. 在上一步中建立的集合的各项全部是应用运算符所产生的结果的类别，例如，运算符 Not 的全部结果是布尔类。现在要取同一类的所有集合（既有原来的又有新建立的集合）的并集。

a) 对布尔类别，并集为{True,False,Not(True),Not(False)}。

b) 对自然数类别，该步的结果为集合{0,Next(0)}。

4. 通常如要定义一个由项组成的无限集，以便不断重复上述两步。

a) 由字面值 True、False 和运算符 Not 产生的布尔项的集合是{True,False,Not(True),Not(False),Not(Not(True)),Not(Not(False)),Not(Not(Not(True))),…}

b) 由字面值0和运算符 Next 产生的自然数项的集合是{0,Next(0),Next(Next(0)),Next(Next(Next(0))),…}

c) 由基色类别的字面值 Red，Green，Blue 和运算符 Take，Mix 产生的颜色项的集合是{Take (Red),Take(Green),Take(Blue),Mix(Red,Take(Red)),Mix(Red,Take(Green)),Mix(Red,Take(Blue)),Mix(Green,Take(Red)),Mix(Green,Take(Green)),Mix(Green,Take(Blue)),Mix(Blue,Take(Red)),Mix(Blue,Take(Green)),Mix(Blue,Take(Blue)),…}。

D.6.1.4 方程与公理

如同上节那样产生的项的数目一般来说大于相应类中值的数目，例如，布尔值的数目为2，但其相应布尔项的集合却有无限个元素。虽然如此，仍有可能给出规则来说明哪些项被认为是表示同一值的。这些规则称为方程，将在下一小节中予以阐述。两类特殊的方程（公理和条件方程）要在§D.6.1.4.2和§D.6.1.4.3中介绍。

在具体的语法中，方程、公理及条件方程全都在关键字 AXIOMS 之后给出，该关键字由于历史原因沿用至今。

D.6.1.4.1 方程

一个方程说明哪两个项被认为表示同一值。方程把位于等价符号==两侧的两个项联系起来了。

例如，“Not(True)==False”说明项 Not(True)和 False 是等价的，凡是 Not(True)出现的地方，都能用 False 代替而不改变原来的意义，反之亦然。

通过给出的一些方程，项的集合被分成一些不相交的子集，每个子集中的项表示同一个值，这些子集称为等价类。通常用等价类的一个值来标识该等价类。

例：

a) 布尔类别的项的集合借助下列两条公理可被划分成两个等价类：

Not(True)==False;
Not(False)==True;

形成的等价类为：

{True,Not(False),Not(Not(True)),Not(Not(Not(False))),Not(Not(Not(Not(True)))),Not(Not(Not(Not(Not(False))))),…}

和

{False,Not(True),Not(Not(False)),Not(Not(Not(True))),Not(Not(Not(Not(False)))),Not(Not(Not(Not((Not(True))))),…}

这两个等价类用值 True 和 False 来标识。

b) 在颜色的情况中，人们想要指明把一基色与一含有该基色的颜色相混合不产生新的颜色，而且与多个基色混合起来的次序是没有关系的，这可用方程说明如下：

```
Mix(Red, Take(Red))                 == Take(Red);
Mix(Red, Mix(Red, Take(Green)))     == Mix(Red, Take(Green));
Mix(Red, Mix(Red, Take(Blue)))      == Mix(Red, Take(Blue));
Mix(Red, Mix(Green, Take(Red)))     == Mix(Green, Take(Red));
Mix(Red, Mix(Blue, Take(Red)))      == Mix(Blue, Take(Red));      etcetera.
```

这是一件工作量很大的工作，因为对 Red、Green 和 Blue 的所有排列都有相似的方程出现，因此 SDL 有 FOR-ALL 构件，它引入了代表一任意等价类（或与此等价类相关的值）的值名字。在上述情况中，这点很有益；以上表示的所有方程及其他种种所表示的那些方程，可简略描述如下：

```
FOR ALL p1, p2 IN Primary_colour
/* 1 */    (    Mix( p1, Take( p1 ) ) == Take( p1 );
/* 2 */         Mix( p1, Mix( p1, Take( p2 ) ) ) == Mix( p1, Take( p2 ) );
/* 3 */         Mix( p1, Mix( p2, Take( p1 ) ) ) == Mix( p2, Take( p1 ) );
/* 4 */         Mix( p1, Take( p2 ) ) == Mix( p2, Take( p1 ) );
               FOR ALL c IN Colour
/* 5 */         (    Mix( p1, Mix( p2, c ) ) == Mix( p2, Mix( p1, c ) );
/* 6 */              Mix( p1, Mix( p2, c ) ) == Mix( Mix ( p1, Take( p2 ) ), c ))
          )
```

以上方程虽有重复，但只要方程彼此不矛盾就不会出现问题。

在颜色类别的项的集合中，上述方程建立起了7个等价类，因此对这些方程来说也有7个颜色值。下列各项是在不同的等价类中的项：

```
Take( Red ), Take( Green ), Take( Blue ),
Mix( Red, Take(Green) ),
Mix( Green, Take(Blue) ),
Mix( Blue, Take(Red) ),
Mix( Blue, Mix( Green, Take(Red) ) ).
```

颜色类别的所有其它项与上面的其中一项等价。

在采用 FOR-ALL 构件的方程例中即通常所说的显式指定类别方程，P1和 P2是基色类别的值标识符的信息是多余的；运算符 Take 的变元和运算符 Mix 的第一个变元只能是基色类别。一般说来当它们被显式指定类别时，可使方程更易于理解，但如果值标识符的类别能由上下文推出则允许省略指定类别的式子，在这种情况下就说方程是隐式指定类别的。

例：

上面的方程4和方程5与下面的两个方程是相同的。

```
Mix( p1, Take( p2 ) )     == Mix( p2, Take( p1 ) );
Mix( p1, Mix( p2, c ) )   == Mix( p2, Mix( p1, c ) );
```

D.6.1.4.2 公理

公理只是一类特殊的方程，它是由于在一些实际情况中许多方程都与布尔量有关而引入的。在这种情况下，一些方程往往以形式…==True 出现，也就是说这些方程表明某个项与 True 是等价的。

例：

假定对颜色类型定义了一个运算符：Contains：Colour，Primary_Colour→Boolean；它的意图是如果在颜色中包含基色，则给出 True，否则为 False。所涉及到的一些方程是：

```
FOR ALL p IN Primary_colour
(     Contains( Take( p ), p )                    == True ;
      FOR ALL c IN Colour
      (     Contains( Mix( p, c ), p )            == True )
)
```

这些方程的“==True”部分可以省去，其结果称为公理。可通过等价符号==有无来识别公理，若无，则认为方程是公理，并且一个公理表示一个项，它与布尔类别的值 True 等价。

第二个方程的结构看来似乎有点勉强，我在介绍一些有用的构件后，还将指明写这些方程的较好办法。

D.6.1.4.3 条件方程

为了书写仅在某些条件满足的情况下才成立的方程，就要用到条件方程。用与普通方程（非条件方程）相同的语法来表示条件，并用==>符号来把方程和条件隔开，如果条件满足，则方程成立。

例：

一个条件方程的标准例子是实数类型中的除法定义，式子

```
FOR ALL x, z IN Real
(     z /= 0 == True        ==>          ( x / z ) * z == x )
```

表明如果条件“Z 不等于0”成立，则任一值 X 在被 Z 除之后接着乘以 Z 就得到原来的值 X。如果实数类别的一个值被0除，将会怎样呢?这个条件方程并未对此作出规定。如果在被0除的情况下，人们想对此做出规定，则须给出如下形式的一个条件方程：

```
FOR ALL x, z IN Real
(     z = 0 == True         ==>          ( x / z ) * z == ... )
```

然而，在这种情况下，为了提高可读性，建议在==符号右端采用一个所谓的‘条件项’。对于上述情况方程可写为：

```
FOR ALL x, z IN Real
(     ( x / z ) * z      ==   IF z /= 0
                              THEN x
                              ELSE ...
                              FI
)
```

D.6.1.5 有关方程与公理的补充内容

下面两节阐述当运算符有一已经定义的类别的结果时，人们可能遇到的某些困难。§D.6.1.5.3介绍了关于差错的概念，差错可看成是方程的一个项。

D.6.1.5.1 层次一致性

在一SDL规格的任何位置，有一个且仅有一个数据类型定义，该数据类型定义包含预定义类别，运算符和方程，以及由用户在部分类型定义中所定义的在该位置可见的所有类别、运算符和方程(这就是为什么要把一正文 NEWTYPE…ENDNEWTYPE 称为部分类型定义的原因)。

这里对较低层的类型定义有一些影响，它们可能会以人们不希望的方式影响该类型。例如，人们可能错误地规定两项是等价的，从而在一周围作用域中使它们等价，即使它们实际上并不等价。

以下述方式给出方程是不能容许的：

a) 规定一类别中的一些值等价，而这些值在高层作用域中并不相等；

b) 把一些新的值加入在一较高层作用域内已定义的类别中。

这意味着，例如，在系统层的一个功能块中，由用户所规定的部分类型定义，如果含有一具有预定义结果类别的运算符，则必须把由该运算符产生的所有的项与该结果类别的值联系起来。

例：

a) 由于某种原因，如果给出公理：

```
FOR ALL n,m IN Integer
((Fact(n)=Fact(m))=>(n=m))
```

其目的是想指明如果运算符 Fact 的运算结果是相同的，则其变元相同（注意=>是布尔蕴含，这与条件方程符号==>基本上没有联系），于是无意中使一些值一致起来了。由上面例子中的方程可以推出 Fact(0)=Fact(1)，这个方程说明0和1是同一值的不同表示，由此方程可以证明整数类别中元素的数目可简化到1个。

利用条件方程，可以说明，只要 n 和 m 不等于0，运算符 Fact 作用在 n 及 m 上，如果得出相同的结果，就可得到 n=m，用 SDL 可表示成：

```
FOR ALL n,m IN Integer
(n/=0,m/=0==>(Fact(n)=Fact(m))=>(n=m))
```

注意这一最后方程没有把任何东西加到整数的语义上，它是一条能由其它方程推导出的定理。另一方面，增加一个可证明的方程不会有什么害处。

b) 假定人们认为在规定某一类型时需要一个阶乘运算符。在该类型的部分类型定义中，

可引入一运算符 Fact：

Fact：Integer→Integer；

并且给出下列方程用来定义该运算符：

```
Fact( 0 ) == 1 ;
FOR ALL n IN Integer
(     n > 0 ==> Fact( n ) == n * Fact( n-1 ) )
```

这些方程没有对 Fact(－1)作出定义，因而 Fact(－1)是整数类别的一项，但它与整数类别的其它项没有关系。所以，Fact(－1)成为整数类别的一个新值（同样适用于 Fact(－2)、Fact(－3)等），这是不能容许的。§ D. 6. 1. 5. 3的例 b）表示了 Fact 的一个正确定义。

D. 6. 1. 5. 2　相等与不相等

在每个类型中，都含有相等和不相等运算符。因此，如果一个部分类型定义引入了 S 类，则有隐式的运算符定义：

```
"=" : S, S -> Boolean ;
"/=" : S, S -> Boolean ;
```

（注：引号表示＝和/＝用作中缀运算符）。

相等运算符具有人们所期望的性质：

```
a = a ,
a = b => b = a ,
a = b AND b = c => a = c ,
a = b => a == b,
a = b => op( a ) = op( b ) for all operators op.
```

在 SDL 语法中没有给出上述性质，且不可以用公理或方程的方式来表示，因为它们是隐含的。当应用相等运算符时，如果左边的项与右边的项是在同一等价类中，则得到的布尔值为 True，否则为 False。如果不显式地指明，则该值既可能是 True 也可能是 False，那么规格就是

不完全的。

对不等运算符，其语义最好用一个 SDL 方程来说明：

```
FOR ALL a, b IN S
(     a /= b == Not( a = b ) )
```

相等与等价之间没有什么差别，等价的两项表示同一值而它们之间的相等运算符给出结果 True。

D. 6. 1. 5. 3　差错

先前的例子说明有必要对某些值的运算符的错误的应用予以说明。SDL 有一种从形式上指出这一点的方法：这就是 Error。Error 应被用来表示。

“把该运算符作用于这个值是不允许的，且当遇到这种情况时，所产生的行为是未定义的”。

在具体的语法中，这种情况用项 Error! 来表示，但这个项不能被用作为一个运算符的变元。

当一运算符运算的结果是 Error，且该运算是另一个运算符的一个变元时，外层运算符运算的结果也就是 Error（差错传播）。在一条件项中要对 THEN 部分或者对 ELSE 部分进行处理，因此其中之一可以直接是 Error，而不需要对 Error 进行运算（因为另一种选择可进行处理）。

例：

a)　在实数类的值的除法中，能用 Error 插入点处：

```
FOR ALL x, z IN Real
(     ( x / z ) * z ==  IF z /= 0
                        THEN    x
                        ELSE    Error!
                        FI
)
```

为清楚起见，可以加上：

```
FOR ALL x IN Real
(     x / 0 ==   Error! )
```

b) 在运算符 Fact 的例子中，应说明这一情况，即该运算符作用在负整数上是一个差错。但这样做却避免了 Fact(－1),Fact(－2)，…成为整数类的新的值。运算符 Fact 的一个好的定义将是：

```
n < 0     ==> Fact( n ) == Error! ;
              Fact( 0 ) == 1 ;
n > 0     ==> Fact( n ) == Fact( n-1 ) * n ;
```

上面形式的三行表示比下面的程序形式的方程表示清晰多了。一般如果有两个互补的情况，就应使用条件项。条件项的嵌套损害了可读性，这从下面的方程可以看出：

```
Fact( n )  ==  IF n > 0
               THEN Fact( n-1 ) * n
               ELSE    IF n = 0
                       THEN 1
                       ELSE Error!
                       FI
               FI
```

D.6.2 生成程序和继承

这一节涉及两个构件，它们能用来规定有公共部分的类型。生成程序规定的不是一个类型而是一个模式，当形式的类别、运算符、字面值及常量用实在的参数代替后，该模式就成为一个类型。

继承提供了从一已存在的类型出发产生一种新类型的可能性。可以对字面值和运算符的名字重新命名，还能规定增加的字面值、运算符和方程。

D.6.2.1 生成程序

生成程序的定义规定了一个以类别、字面值、常量及运算符的形式名字巍—病//数的模式。生成程序用来表示“在一个主题上变化”的类型，如项的集合，项组成的串，记录组成的文件，查询表、数组等。

下面用一个可表示这种变化的例子来说明这一点。假定需要一个类似一个整数集合的数学概念的类型，下面的正文是该整数集的部分类型定义。

```
NEWTYPE Int_set
    LITERALS empty_int_set
    OPERATORS
        Add : Int_set, Integer -> Int_set ;
        Delete : Int_set, Integer -> Int_set ;
        Is_in : Int_set, Integer -> Boolean
    AXIOMS
    /* 1 */    Delete( empty_int_set, int )
                        ==   empty_int_set ;
    /* 2 */    Is_in(set,int1) = false ==>
               Delete( Add( set, int1 ), int2 )
                        ==   IF int1 = int2
                             THEN  set
                             ELSE  Add( Delete( set, int2 ), int1 )
                             FI ;
    /* 3 */    Is_in( empty_int_set, int )
                        ==    False ;
    /* 4 */    Is_in( Add( set, int1 ), int2 )
                        ==   int1 = int2 OR Is_in( set, int2 ) ;
    /* 5 */    Add( Add( set, int1 ), int2 )
                        ==   IF int1 = int2
                             THEN  Add( set, int1 )
                             ELSE  Add( Add( set, int2 ), int1 )
                             FI
ENDNEWTYPE Int_set;
```

图 D-6.2.1

新类型 Int_set

所有方程已隐含了类别。第一个方程说明从空集中删除一个元素，其结果为空集；第二个方程说明如果在插入一元素之后删除同一元素，其结果巍—病"入前的集合（只要该集合原先不含有该元素），如果插入和删除的不是同一元素，则插入和删除的顺序可以交换；第三个方程说明空集不包含任何元素；第四个方程说明一个元素是在一个集合之中，如果它是最新加入的或在最新元素加入之前它就在集合中；最后一个方程说明元素加入的顺序对集合没有任何影响。

在图 D-6.2.1的例子中，Int_set 只是一个集合的例子，如果在同一规格中人们又需要一个 PId_set、一个 Subscriber_set 和一个 Exchange_name_set 等集合，那么这些集合都会包含运算符 Add、Delete、Is_in 及一个作为空集的字面值，这是很自然的。在这里所给出的这些运算符的方程很容易推广到其它集合。

公共的正文只要给出一次，就能被使用若干次，这正是生成程序概念其有效性之所在。

图 D-6.2.2表示了上述集合的生成程序（注意，由于历史的原因），形式类别名由关键字 TYPE 引入。

```
GENERATOR Set (TYPE Item, LITERAL empty_set)
    LITERALS empty_set
    OPERATORS
        Add : Set, Item -> Set ;
        Delete : Set, Item -> Set ;
        Is_in : Set, Item -> Boolean
    AXIOMS
    /* 1 */    Delete( empty_set, itm )
                    ==  empty_set ;
    /* 2 */    Is_in(st,itm1) = false ==>
               Delete( Add( st, itm1 ), itm2 )
                    ==  IF itm1 = itm2
                        THEN  st
                        ELSE  Add( Delete( st, itm2 ), itm1 )
                        FI ;
    /* 3 */    Is_in( empty_set, itm )
                    ==  False ;
    /* 4 */    Is_in( Add( st, itm1 ), itm2 )
                    ==  itm1 = itm2 OR Is_in( st, itm2 ) ;
    /* 5 */    Add( Add( st, itm1 ), itm2 )
                    ==  IF itm1 = itm2
                        THEN  Add( st, itm1 )
                        ELSE  Add( Add( st, itm2 ), itm1 )
                        FI
ENDGENERATOR Set;
```

图 D-6. 2. 2

生成程序集合

图中用形式类型 Item 代替使用 Integer。为了能够对空整数集和其它类型中的空集给出不同的名字，其字面值也被规定为形式参数。

利用该生成程序，类型 Int_set 可构成如下：

```
NEWTYPE Int_set Set(Integer, empty_int_set)
ENDNEWTYPE Int_set ;
```

比较图 D-6. 2. 1和图 D-6. 2. 2可以看出：

a) GENERATOR 和 ENDGENERATOR 分别为 NEWTYPE 和 ENDNEWTYPE 所代替；

b) 删去生成程序形式参数（即生成程序名之后的括号之间的正文），

c) 在整个生成程序中，Set，Item 及 empty_set 分别为 Int_Set、Integer 及 empty_int_set 所代替。

这样该 Int_set 和图 D-6.2.1中的 Int_Set 之间就完全没有差别了。

但是，如果人们需要 PId 值的一个集合，则可据下面的生成程序建立此类型：

```
NEWTYPE PId_set Set(PId, empty_pid_set)
ENDNEWTYPE PId_set ;
```

如果人们需要由用户组成的一个集合（这里通过引入 Subscr 类别来表示用户），则可据下面的生成程序建立该用户的集合

```
NEWTYPE Subscr_set Set(Subscr, empty_subscr_set)
ENDNEWTYPE Subscr_set ;
```

这不仅节约了纸张，而且使用也变得更容易了，因为人们关于各种集合仅须考虑一次，甚至还可把这项工作委托给有经验的抽象数据类型专家去做。

例：

这个例子表示一个使用形式类别、运算符、字面值和常量的生成程序。它描述了由项组成的一行，行的最大长度为 max_length。此类别有一字面值，用来表示一行中插入或删除一些项的空行和各种运算符，链接的行、所选择的子行及所确定的行的长度。把确定行长的运算符规定为形参，以便能对它重新命名。

```
GENERATOR Row (TYPE Item, OPERATOR Length, LITERAL Empty,
                    CONSTANT max_length)
    LITERALS Empty
    OPERATORS
        Length: Row                  -> Integer ;
        Insert: Row, Item, Integer   -> Row ;
        Delete: Row, Integer, Integer -> Row ;
        "//" : Row, Row              -> Row ;
        Select: Row, Integer, Integer -> Row
        /*和其它的与行有关的运算符*/
    AXIOMS
        /*与上述运算符有关的方程，其中有下面两个方程
          (或与这两个方程等价的方程)*/
        Length( r ) = max_length ==> Insert( r, itm, int ) == Error! ;
        Length( r1 ) + Length( r2 ) > max_length ==> r1 // r2 == Error!
ENDGENERATOR Row ;
```

注意形式运算符 Length 和字面值 Empty 在生成程序体中再次给出，是因为当实例化时它们要被重新命名。在运算符情况下，其变元和结果类别只在体中给出。生成程序 Row 可以用来产生如下的行、页和书：

```
NEWTYPE Line Row( Character, Width, Empty_line, 80 )
ENDNEWTYPE Line ;

NEWTYPE Page Row( Line, Length, Empty_page, 66 )
ENDNEWTYPE Page ;

NEWTYPE Book Row( Page, Nr_of_pages, Empty_book, 10000 )
ENDNEWTYPE Book ;
```

D.6.2.2 继承

继承是一种手段，用它可以得到所谓的父本类别的全部值、父本类型的某些或全部运算符及父本类型的全部方程。对于字面值和运算符两者都可以重新命名。通常，重新命名是个好办法，因为在这种情况下，即使字面值相同，读者也可以根据上下文推断出包含了另一种类型。

如果不继承某个运算符，那么它将被系统地重新命名为用户不能访问的名字。运算符仍然存在这一事实意味着父本类型的所有方程仍然是存在的（具有重新命名的运算符）。这确保了父本值被继承。

避免使用某个运算符（采用不继承的办法）的可能性的同时，也提供了追加新的运算符的可能性。在关键字 ADDING 之后，可以给出各种字面值、运算符和方程，如同在普通类型中那样。然而，人们对新的字面值必须很小心。另外要注意继承的与追加的运算符之间的相互影响。

当追加字面值时，要考虑继承的运算符作用到这些新的字面值时所产生的结果，必须通过方程给出定义。当追加运算符时，人们应当记住那些隐蔽地重新命名的运算符及其有关的方程。对那些定义追加的运算符的方程应当与继承的运算符和未继承的运算符的方程取得一致。

经过上述警告之后，请看一些例子。

a) 假定新类型 Colour 已完全定义并可供利用。该类型是基于取基色和混合基色光束的。而要定义“取”和“混合”颜料之类的类似物，得经过大量的想象、书写和（或）复制。

解决这个问题的一个好办法是只用两个替换把新类型 Colour 做成一个生成程序：

1) 第一行

```
NEWTYPE Colour
```

变为

```
GENERATOR Colour (TYPE Primary_Colour)
```

2) 关键字 ENDNEWTYPE 变为 ENDGENERATOR。

当被实例化时，该生成程序能被重新命名了。假定前者 Primary_Colour 类别称

为 Light_Primary，则 Paint_Primary 类别被定义为：

```
NEWTYPE Paint_Primary
    LITERALS Red, Yellow, Blue
ENDNEWTYPE Paint_Primary;
```

现在很容易即可定义两个类似的类型（如一个对光线而另一个对颜料）：

```
NEWTYPE Light_Colours Colour (Light_Primary) ENDNEWTYPE;
NEWTYPE Paint_Colours Colour (Paint_Primary) ENDNEWTYPE;
```

至此没有什么问题，但人们如何能了解 Light_Colour Take（Red）和具有同样语法的 Paint_Colour 之间的区别呢?如果认为有必要区别二者，继承是一个好办法。通过继承，重新命名运算符 Take，可以定义 Light 类型和 Palette 类型来代替 Light_colours 和 Paint_colours。

```
NEWTYPE Light
    INHERITS Light_colours
    OPERATORS ( Beam = Take, Mix, Contains )
    ADDING
        LITERALS White
        AXIOMS
            White == Mix( Red, Mix( Yellow, Beam( Blue ) ) )
ENDNEWTYPE Light ;
```

现在新类型 Light 具有来自 Light_colours 的字面值及字面值 White。Light_colours 没有它自己的字面值（因为它使用了 Light_Primary 的字面值），因此 White 是 Light 的仅有的字面值。除了运算符名字 Take 被 Beam 所代替外，Light 的运算符和方程与 Light_colours 的相同，另外追加了 White 的方程，追加的公理说明这个追加的字面值变成项的集合的一个元素，这个元素由三个混合基色组成。

新类型 Palette 具有来自 Paint_colours 的字面值，且 Take 运算符被 Paint 代替。

```
NEWTYPE Palette
    INHERITS Paint_colours
    OPERATORS ( Paint = Take, Mix, Contains )
ENDNEWTYPE Palette ;
```

b) 假定人们想要增加一个运算符以扩展前节中引入的整型集合（Int_Set 类别），设该运算符被用来寻找集合中的最小整数。首先应当考虑该运算符是否能在生成程序的定义中引入，以便使其对其它情况的集合同样有效。尽管能这样做，但它将使 Item 局限

于具有大于>和小于<性质的项，这不适合所有的项（例如 PId）。而制作一个提供运算符 Min 的类别为 New_int_Set 的新类型可能更好些。

```
NEWTYPE New_int_set
    INHERITS Int_set
    OPERATORS ALL
    ADDING
        OPERATORS
            Min : New_int_set -> Integer
        AXIOMS
            Min( Empty_int_set ) == Error! ;
            Min( Add( Empty_int_set, x ) ) == x ;
            Min( Add( Add( nis, x ), y ) ) ==
                                IF y < Min( Add( nis, x) )
                                THEN  y
                                ELSE  Min( Add( nis, x ) )
                                FI
ENDNEWTYPE New_int_set ;
```

因为在一生成程序实例之后进行追加是相当常见的，所以从 ADDING 开始到 ENDNEWTYPE 结束之前的正文可以在生成程序实例内部给出。在建议的 § 5. 4. 1. 12中给出了一个例子。

D. 6. 3　对方程的几点考虑

当引入一新数据类型时，必须要引入足够的方程。后面给出了关于方程的三点考虑，这有助于构造各种方程。

D. 6. 3. 1　总的要求

无论怎样去构造方程，必须把握下列事实：

a）　在该组方程中，每个运算符至少出现一次（病态情况除外）。

b）　所有真语句都能由方程推导出来。它们或者被表示成公理，或者能通过在方程中代入等价项推导出来。

c）　不能检测出不一致性，也就是不能从方程推导出 True＝False。

用非形式的 SDL 可表示得出这些方程的步骤。见图 D-6. 3. 1。

D. 6. 3. 2　关于构造元的功能应用

通常，运算符的集合具有一个称之为“构造元”和“函数”的运算符的子集。构造元可用来生成该类别的所有值（等价类）。在这种方法中，字面值被认为是不带变元的运算符。

例：

a）　布尔类型把它的字面值作为构造元。

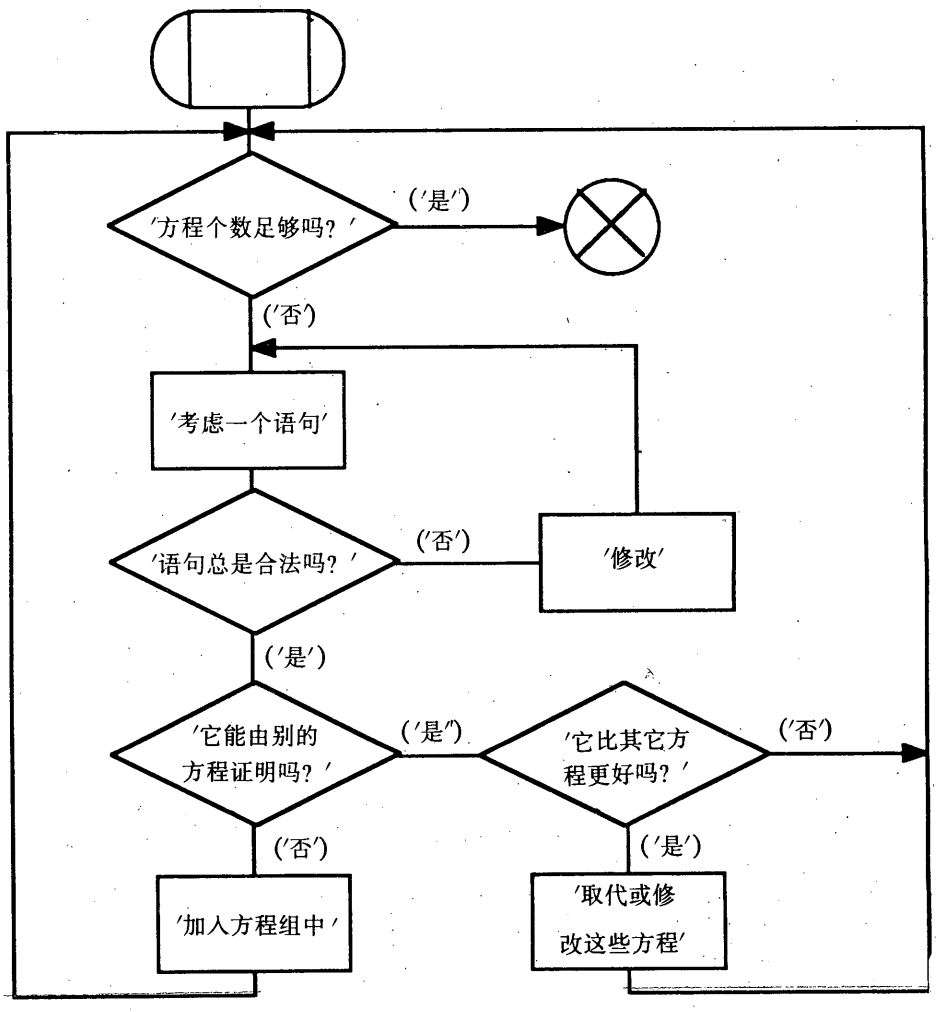


图 D-6.3.1

用非形式 SDL 求得方程的过程

b) 自然数类型把字面值 0 和运算符 Next 作为它的构造元，每个自然数可以只用 0 和 Next 构成。

c) 各种集合的生成程序把字面值 empty_set 和 add 运算符作为它的构造元，任何一个集合都可以只用 empty_set 和 add 构成。

d) 整型可以由字面值 0 和 1、运算符＋和一元减来构造。

应当注意，有时对构造元集合可能有若干种选择，在本节随后的内容中任何选择都可以，但一般最好是选择小的集合。

现在来一一处理全部的函数。对一个函数的每个变元，应该列出仅仅由构造元组成的全部的项。为了避免出现无穷多项的麻烦，应该采用量化的办法。

例：

a) 对于自然数，构造元组成的表可以缩小到只包含下面两项

0

Next(n) 式中 n 是一自然数。

b) 对于集合，可以用下列构造元组成的表

empty_set

Add (s, i) 式中 s 是任意一个集合，i 为任意项。

如果一个方程的右侧项含有 (s, i) 在它的左边，则 s 为空或者不为空二者之间是有差别的，上面的表可改写如下：

empty_set

Add(empty_set,i) 式中 i 是任意项，

Add(Add(s,i),j) 式中 s 是任意集合，i，j 为任意项。

在该表建立以后，方程的左侧可通过把每个函数作用于表中各变元的任意组合来获得。不同变元中的值标识符用不同的名字表示。上面对各种函数所给出的过程也可以为构造元所采用。在这种情况下，它给出了按不同的次序使用构造元的项之间的关系。

例：

a) 对自然数乘法运算符采用符号 *

" * ": Natural, Natural—〉 Natural

该过程给出下列（不完全的）方程的左侧，用户应提供方程的右侧。

```
        0 * 0          == ... ;
        0 * Next ( n )  == ... ;
Next ( n ) * 0          == ... ;
Next ( n ) * Next ( m ) == ... ;
```

b) 对于在 Set 生成程序（§ D. 6. 2. 1）中的运算符 Is_in 和 Delete，已经使用了这个方法。

c) 对颜色类别，其构造元为 Take 和 Mix。

对于变元，必须定义一个类似于 § D. 6. 1. 4. 2中的 Contains 的运算符。

Take(p) 式中 p 是任一基色，

Mix(p,c) 式中 p 是任一基色，c 是任一颜色。

在 § D. 6. 1. 4. 2中曾答应过对此运算符给出简明的方程。它们可详细地表示为：

```
Contains( Take( p ), q ) == p = q ;
Contains( Mix( p, c ), q ) == ( p = q ) OR Contains( c, q ) ;
```

此构造方程的过程可能产生比所需更多的方程，这是很可靠的。例如上述自然数相乘的例子很可能要表示乘法的交换性质，而只有前三个方程的最后一个（或第二）个才是需要的。

本节中描述的过程可以与前节中描述的过程（在任务 "Think_of_a_statement" 中是很有用的）结合使用。

D.6.3.3 测试集规格

从实现的角度出发也可考察研究方程。如果在一程序设计语言中，运算符被作为函数来实现，那么方程须说明这些函数将如何进行测试。

计算对应于一个方程左右两端表达式的值并查看它们是否等效。这一方法对于 FOR ALL 构件可能引起问题。通常用一很实用的方法来解决这个问题：

测试者可以采用　FOR ALL i IN{−10,−1,0,1,10}

代替　　FOR ALL i IN Integer

在大多数情况下，这样做是可行的。

在§D.6.3.1过程中的任务“Think_of_a_statement”中，把方程看做实现的必要条件是会有帮助的。

D.6.4 特性

本节描述 SDL 的若干特性，这些特性较少需要到，或者说即使没有也行，但有时使用它们却显得更为方便。

D.6.4.1 隐蔽运算符

如果引入一额外的运算符，有时可使得方程组简化或者更易理解，但该运算符不应在进程中使用。这意味着该运算符在类型定义内部是可见的，而在外部是隐蔽的。

这种结果可通过定义一‘隐蔽类型’（即用户不应使用的类型）来获得。在该隐蔽类型中，用户可继承允许他访问的所有运算符；应该采用的类型即继承类型。如要检验使用是否正确，可以去检查所有的变量声明（隐蔽类型引入的类别的变量不应出现）。

隐蔽运算符的特性表明，如果把这些运算符的可见性仅局限于有关方程组，也可达到相同的结果。这可以通过在运算符后面加一个惊叹号“!”来实现。

例：

在生成程序 Set 中产生一个只含有一个元素的集合的常规方法是：

```
Add(empty_set,x)
```

这是每个用户应使用的方法。书写规格者可在方程组中采用一特别的运算符，例如：

```
Mk_set!:Item−>Set;
```

其含义通过下面的方程来定义：

```
Mk_set!(itm)==Add(empty_set,itm);
```

前一方程可以在部分类型定义中使用，但不能用在 SDL 进程体中。

D.6.4.2 排序

当必须规定一个类别中各元素的次序时，通常意味着必须定义四个运算符（<，<=，>，

>=）及标准的数学性质（可传递性等等）。如果有许多字面值，那么也得给出许多方程。例如，预定义数据类型 Character 就是用这种方式定义的。

SDL 提供了一个简化的特性 ORDERING，可以克服上述冗长的、不易理解和令人厌烦的类型定义。

ORDERING 在运算符表中给出，最好是安排在该表的开头或结尾。这样就引入了排序运算符和有关的标准方程。当规定 ORDERING 时，所涉及的字面值（如果有的话）必须按递增次序给出。

例：

```
NEWTYPE Even_decimal_digit
    LITERALS 0, 2, 4, 6, 8
    OPERATORS
        ORDERING
ENDNEWTYPE Even_decimal_digit;
```

这样就隐含了次序0<2<4<6<8。

在§D.6.2.2（继承）中曾给出了警告，如要把字面值加到一继承类别中，则须小心，这里恰到好处地解释了给出警告的原因。

假定要扩展类别 Even_decimal_digit 如下：

```
NEWTYPE Decimal_digit
    INHERITS Even_decimal_digit
    OPERATORS ALL
    ADDING
        LITERALS 1, 3, 5, 7, 9
        AXIOMS
            0 < 1;    1 < 2;
            2 < 3;    3 < 4;
            4 < 5;    5 < 6;
            6 < 7;    7 < 8;
            8 < 9
ENDNEWTYPE Decimal_digit;
```

这里给出的公理不可省略。如无这些公理，则只有所谓的部分排序：

0<2<4<6<8

和 1<3<5<7<9。

运用上述的一组公理就有了一个完全的排序：

0<1<2<3<4<5<6<7<8<9

但若用公理“9<0”代替该组公理，所得到的完整的排序将成为：

1<3<5<7<9<0<2<4<6<8。

D.6.4.3 带有字段的类别

如建议的§5.4.1.10所指出，人们可以定义一个没有特殊构件的结构类别，但是结构类别既很普通又很有用，这就值得在此语言中增加某些额外的构件。

当一个客体的值是由若干个类别的值组合构成时，应当采用结构类别。该组合的每个值用一名字来表征，称为字段名，一个字段的类别是固定的。

例：

```
NEWTYPE Subscriber
    STRUCT numbers  Number_key ;
           name     Name_key ;
           admin    Administrative ;
ENDNEWTYPE Subscriber ;

NEWTYPE Name_key
    STRUCT name,
           street   Charstring ;
           number   Integer ;
           city     Charstring;
ENDNEWTYPE Name_key ;
```

对于结构类别，某些运算符被隐式地定义：

a) 构造元运算符，在字段值之前的“（•”和在字段值之后的“•)”；

b) 字段选择运算符，结构类别的变量后面是一惊叹号!以及字段名，或者是括在圆括号中的字段名。注意变量后面不应跟有!以免与隐蔽运算符相混淆（§D.6.4.1）。

图D-6.4.1给出了一个例子。

```
PROCESS Some_process;
    DCL na, st, where   Charstring   ,
        nu              Integer      ,
        nk              Name_key     ;
    /*…这里的正文把值赋给变量 na 和 st*/
    TASK nk := (. na, st, 5, 'London' .) ;
    /*…这里的正文没有出现对 nk 赋值*/
    TASK where := nk!city ;
    /*现在保持 where = 'London'*/
ENDPROCESS Some_process;
```

图 D-6.4.1

隐式结构类别运算符使用举例

D.6.4.4 索引类别

索引类别是这样一个类别，它把 Extract!作为它的一个运算符名字。在预定义数据类型

中，生成程序 Array 就是这样一种类型。数组是加标类型的最常见的例子之一。

对隐蔽运算符 Extract!有一专门的具体语法，该语法应在类型定义外部使用。

预定义生成程序 Array 中的类型 Index 可能会被认为应是一“简单”类型，如 Integer、Natural 或 Character。然而，认为象 Name_key 那样的结构不能作为 Index 使用是没有理由的。

例：

```
NEWTYPE Subsc_data_base
        Array (Name_key, Subscriber)
ENDNEWTYPE Subsc_data_base;
```

类别 Name_key 和 Subscriber 是在先前的节中定义的。假定有一个过程 Bill 具有 Subscriber 类别的参数，且该过程在一进程中定义，而该进程也有 Subsc_data_base 类别的一个变量 Sub_db，那么在该进程中，可以出现下列调用：

```
CALL Bill(Sub_db((.'P. M.',Downingstreet',10,'London'.)));
```

D.6.4.5 变量的缺省值

正如在变量声明一节（§D.3.10.1）中所阐明的，可以在声明之后把值直接赋给一个变量。然而，有些类型的变量却总是（或几乎是）具有一特定的初始值。有一种特性能够避免在每次声明时都必须写出初始值，这就是 DEFAULT 子句。

作为举例，考虑一个集合，很可能几乎所有人们可以想象到的集合变量都用 empty_set（空集）作为其初始值。

在列出一些方程之后，写下：

```
DEFAULT empty_set
```

就可确保生成程序的每个实例的每个变量将被初始化到该实例的空集（empty_set），除非作出了显式的初始化（见§D.3.10.1）。

倘若认为一个类别的所有变量的初始值都是相同的这一点很重要的话，那么就不宜使用 DEFAULT 子句，否则会使人感到意外！

D.6.4.6 主动运算符

熟悉 SDL'84Z.104的用户可能会对删除所谓的主动运算符这一特征感到奇怪，删去的原因如下：

a) 它不是必须的，因为普通的运算符与过程和（或）宏一起提供了同样的表达能力；

b) 它降低了方程的可读性；

c) 许多用户在正确地使用此特征方面遇到困难；

d) 它不完全符合基于初始代数的抽象数据类型模型（SDL 这一部分选择该模型为其数学基础）。

D.7 绘图和书写的附加准则

本建议为每个SDL概念规定了表示概念的主要方法，例如，任务概念在SDL/GR中用一个矩形来表示，而在SDL/PR中用关键字TASK来表示。

本用户指南在绘图和书写方面规定了一些规则，对建议作了补充。这些规则适用于所有的概念，目的在于强调怎样的绘图或书写是合适的、错误的或不好的。

D.7.1 SDL/GR 准则

D.7.1.1 概述

作图的一般准则是：

— 阅读的顺序应当是自顶向下，自左至右。

— 在同一层次上图形不应包含太多的信息，把大的图划分为若干子图通常是合适的（例如采用引用机制），这些子图包含各个不同的部分或方面。

D.7.1.2 入口处和出口处

从入口处连到任一符号或从任一符号连到出口处的线条应画成垂直的（与自顶向下阅读对应），只有在不可行的地方才采用水平连线的入口处和出口处。

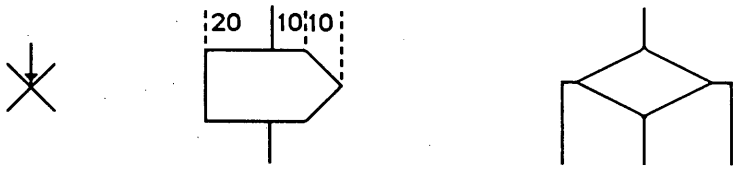


图 D-7.1.1

入口处和出口处的正确画法

这个一般规则的例外情况是：

— 连接到两个或三个出口处的判定框通常画成两个水平出口处（外加一垂直的出口处）。

— 宏调用采用水平的和垂直的连接。

— 在入连接符和出连接符都有的情况下通常采用水平的入口处和出口处。

交叉线只有在特殊情况下才会出现（如对信道和信号路由）。

垂直的入口处（或出口处）的线条应把符号分为具有同样水平长度的两部分。

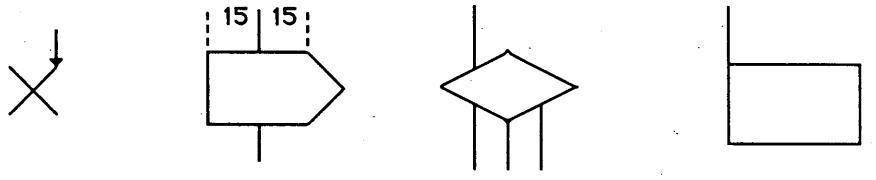


图 D-7.1.2
入口处和出口处的不正确画法

D.7.1.3 符号

a) 符号的垂直线和水平线应与图纸的两个轴相一致。
b) 输入、输出、正文扩展和注释符号可以采用垂直镜面反射的符号式样（见图 D-7.1.3）。
c) 图形中的所有符号其高度与长度之比一般为1:2，对于引用符号也是如此。

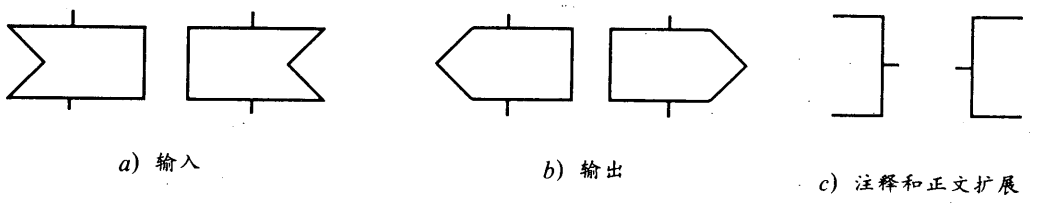


图 D-7.1.3
四种允许的镜面反射符号

D.7.1.4 样板

SDL/GR 符号的样板已在该卷的封三给出，图 D-7.1.4是该样板的示意图。

图中的符号用了三种尺寸，即20×40mm、$20/\sqrt{2}\times40/\sqrt{2}$mm 和10×20mm，直接表示了输入、输出、判定、替换、进程、启动、任务、状态、保存、服务引用标记、连接符及停止

符号。仅对尺寸最大的符号标明了内部的比例。

过程调用、宏调用和创建各符号可由画上附加水平或垂直线的（图中已指明）任务符号来构成。

过程启动符号可由画上附加垂直线的（图中已标明）进程启动符号来构成。

返回符号是连接符号和停止符号的组合。

注释、正文扩展和信号表符号可以用任务符号来画出。

优先输入和优先输出符号可以由画上附加线的（图中已标明）输入和输出符号来构成。

过程引用符号可以由画上两根附加垂直线的（如图所示）进程引用符号来构成。

允许条件和连续信号符号可用停止符号来画出。

信道、信号路由和连接到正文扩展符号的线都是实线。

连接到注释符号的线是1:1的虚线。

正文符号是用任务符号并对折右上角来画出（图中已指明）。

所有建议的符号都在建议中规定。所建议的符号的概要可在附件C-图形语法摘要中找到。

三种指明的尺寸是应优先选用的尺寸。如果采用另外的尺寸，则比例仍应相同(即1:2)。所标明的尺寸，即长度40mm、28mm和20mm允许以兼容的符号尺寸（如$40mm/\sqrt{2}=28mm$和$28mm/\sqrt{2}=20mm$）在从A3到A4型的纸页上进行照相缩版。

D.7.2 SDL/PR准则

编制正文SDL的一般准则是：

— 阅读的顺序应自顶向下，自左至右。

— 正文应划分成若干部分，分别包括各个不同的方面。

— 各语句的注释的开头应对齐，即从同一列开始。

— 各行应采用缩进编排，缩进编排可按照通常的SDL概念的层次，如图D-7.1.5的例子所示。

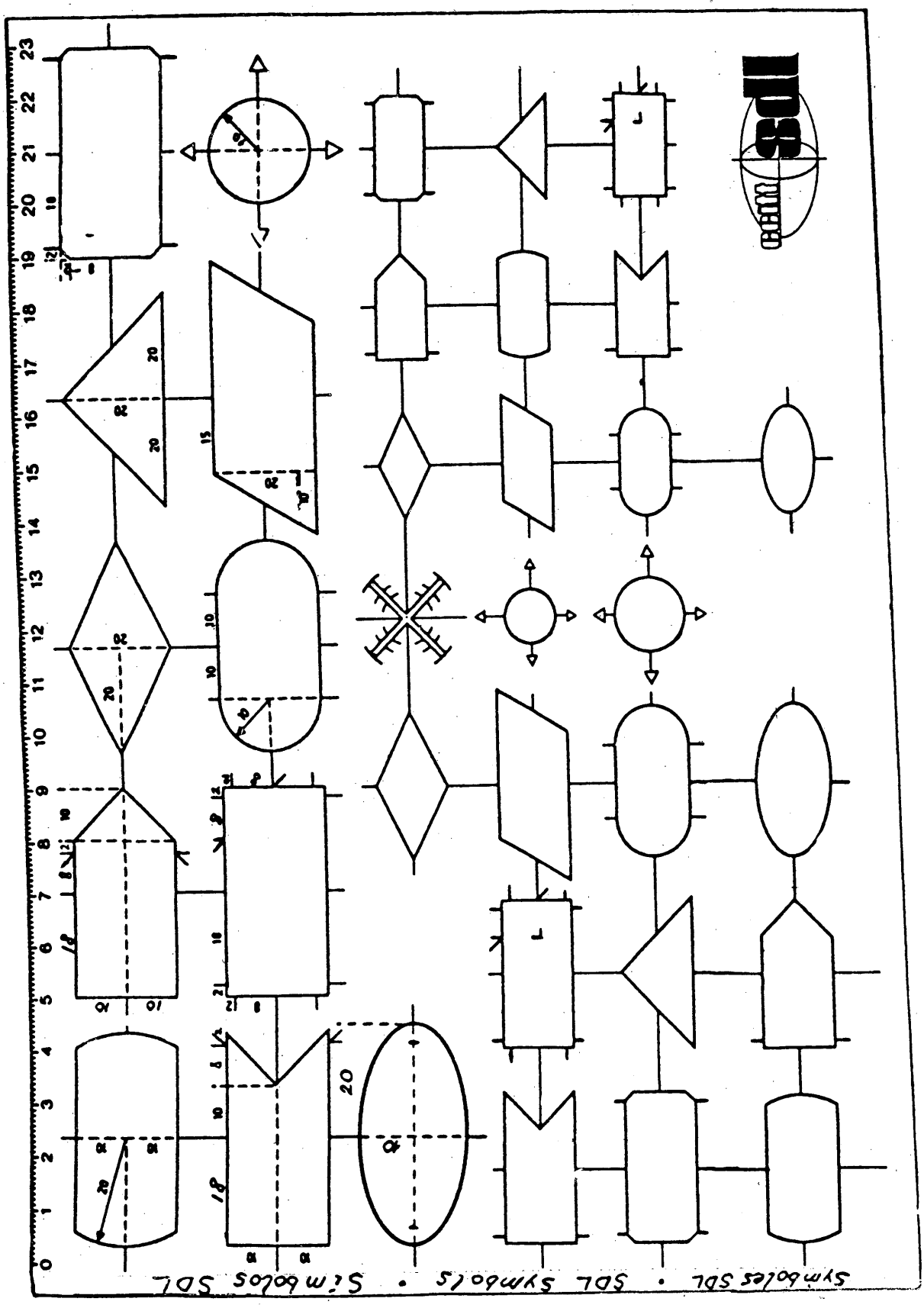


图 D-7.1.4

样板示意图

```
SYSTEM A;
    SIGNAL S1,S2;
    BLOCK B;
        PROCESS P;
            START;
            NEXSTATE ST1;
            STATE F;
                INPUT G: . . .
                INPUT F: . . .
                . . .
            ENDSTATE F;
        ENDPROCESS P;
    ENDBLOCK B;
ENDSYSTEM A;
```

图 D.7.1.5

SDL 中正文缩排举例

D.8 文件编制

D.8.1 引言

ISO 把文件定义为“以可检索的形式贮存在某一媒质上的有限的和相关的信息”，因此应当把它看作是一个严格确定界限的逻辑单位。文件被用来传达与系统（用 SDL 规定的）有关的所有信息。

当用纸作为存贮存文件的物理媒质时，文件这一术语往往用来表示纸张而不是它的逻辑内容，这种表示欠确切。随着磁存贮媒质使用的日益增多，这一述语正恢复到它原来的定义。

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

通过把信息分解成适当数目的文件，能使系统更易于理解和更易于管理。

本语言并不建议某些文件或文件结构。然而，却给出了某些建议，用以帮助用户处理各种文件。

D.8.2 系统表示法的类型

当采用 SDL 来规定一个系统时，其结果是一组用 SDL/GR 和（或）SDL/PR 表示的定义。

这些定义可以是嵌套的或者是顺序的，这取决于系统表示法的类型是分层的还是展开的。图 D-8.2.1和 D-8.2.2中，在 SDL/GR 中，采用两种可供选择的表示法描述了一个系统，这两个图不是完整的系统规格，为了简单起见，仅仅表示了输入和输出，而略去了可能有的信号和数据定义。

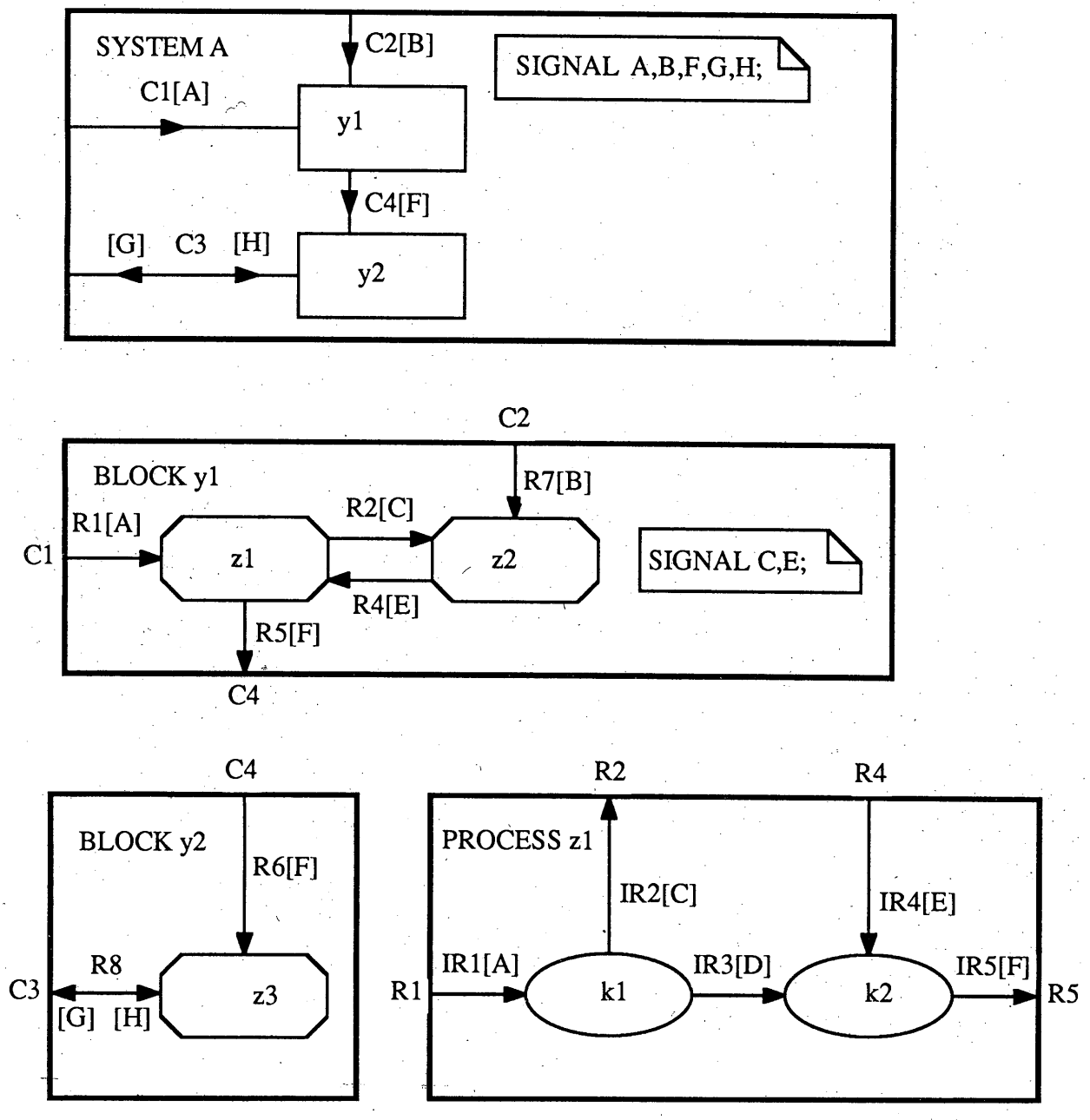


图 D-8. 2. 1a

用于一展开系统表示法的顺序（被引用的）图举例

当然在规定一个系统时允许采用混合表示法。

当定义是顺序表示时（如图 D-8.2.1），它们是‘被引用的’，这是在 SDL/PR 和 SDL/GR 中都可定义的一种机制。

考察用一组定义表示的系统规格，一个文件可以看作是容纳这些定义的容器。

如果系统不大且是分层的（如图 D-8.2.2），用一个单一的文件就够了；如果采用展开表示法（如图 D-8.2.1），可以采用多个文件，例如为每个定义采用一个文件。

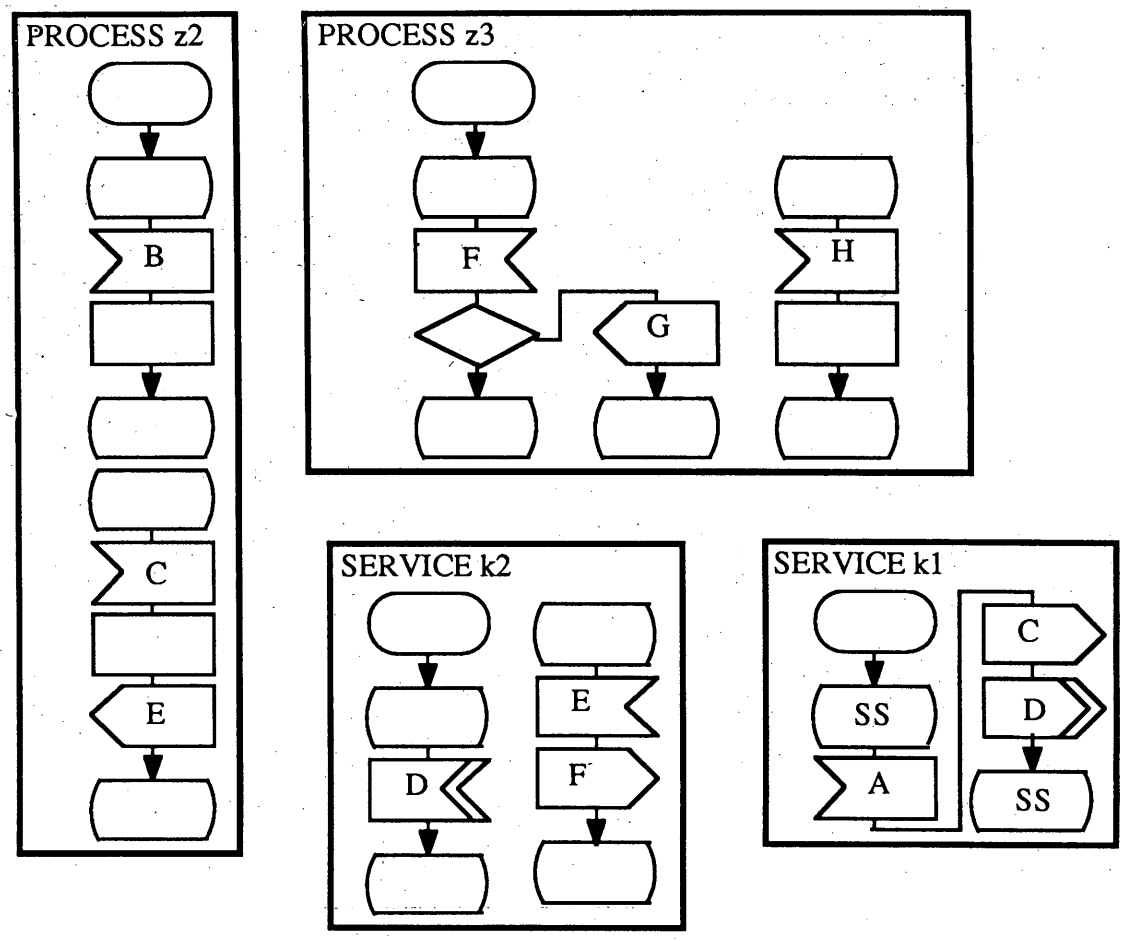


图 D-8.2.1b

用于一展开系统表示法的顺序（被引用的）图举例

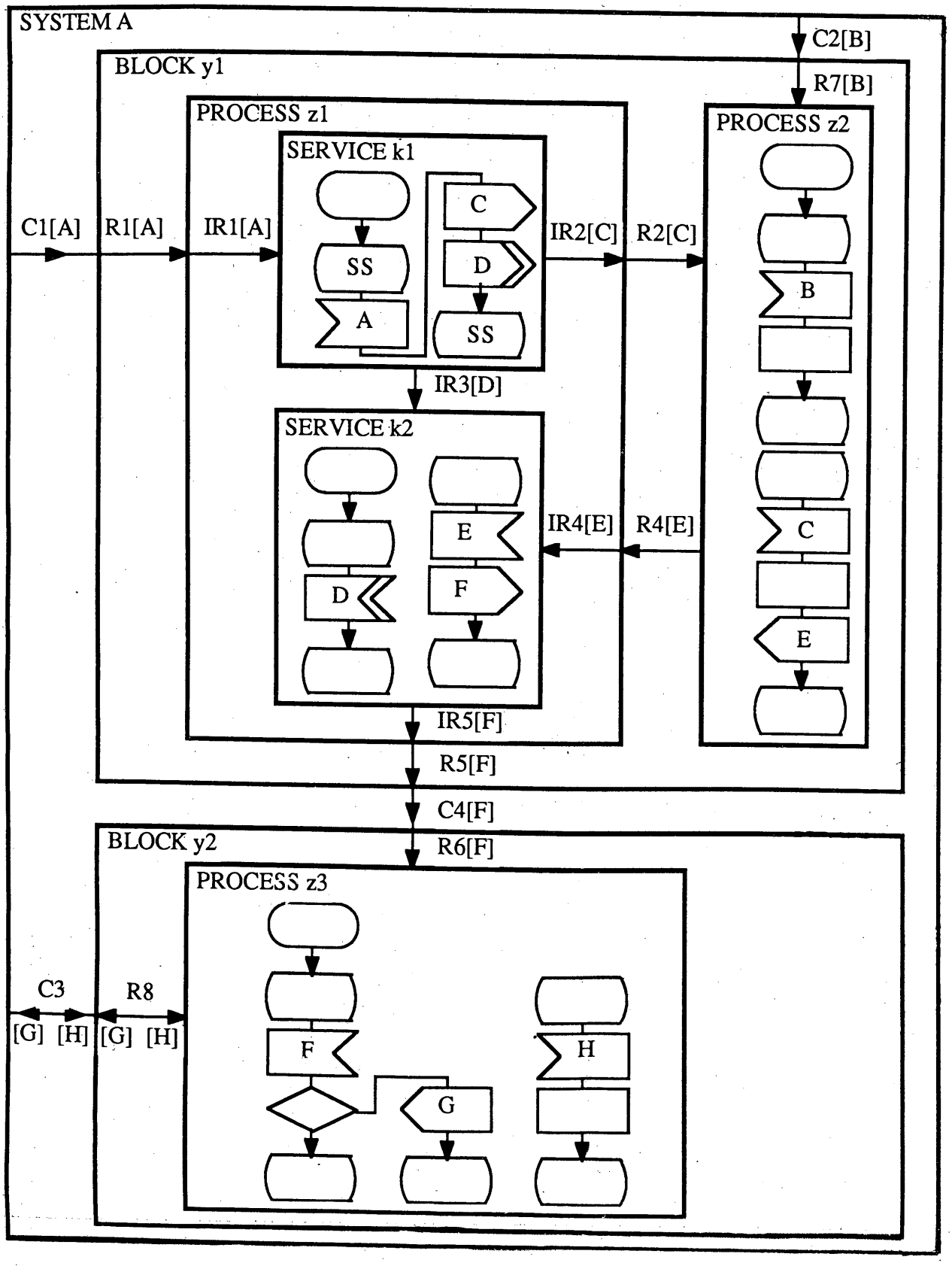


图 D-8. 2. 2

用于一分层系统表示法的嵌套定义举例

在选择系统表示法的类型时，必须考虑所希望要的文件类型。为使每个定义有一个文件，就需要展开的表示法；如果希望整个系统规格只有一个单一的文件，则需要用分层的表示法。

通常的情况多半是上述两种表示法的混合。当选定混合表示法时，下列规则是适用的：

1） 不应把一个定义分配给几个文件；

2） 如果另外一个文件需要用到某个定义，则它必须被引用，而不能被嵌套；

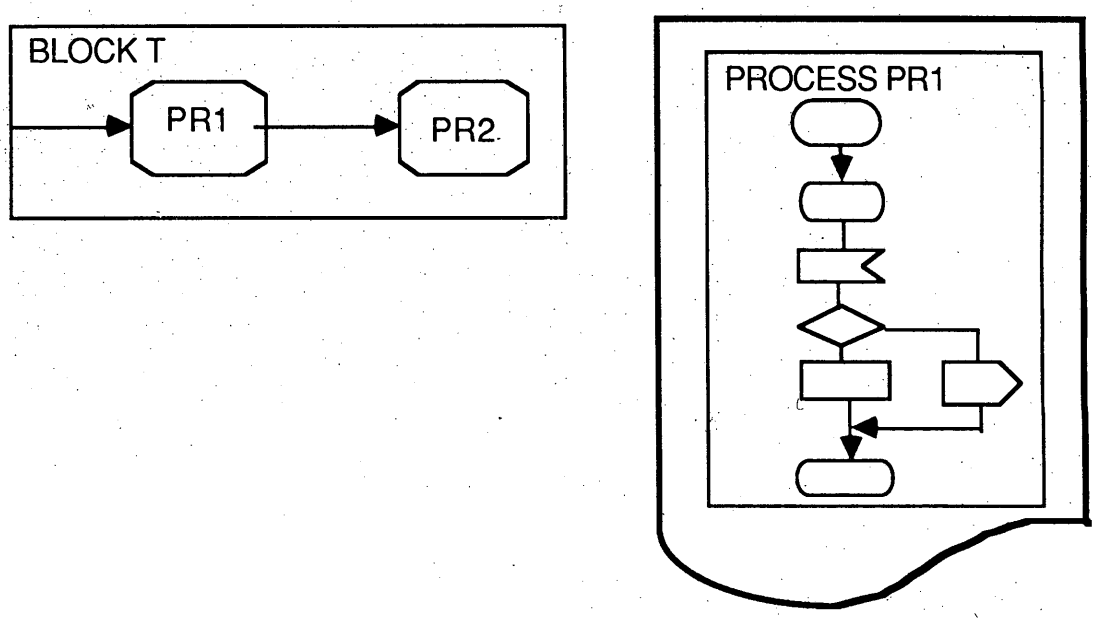


图 D-8.2.3

位于一独立文件中的引用定义

3） 在采用逻辑页概念把一个图分成若干图页时，各图页应当与文件的物理页一致(见图 D-8.2.4)；

4） 如果一个图超过一页时，则它必须被引用，而不能被嵌套。

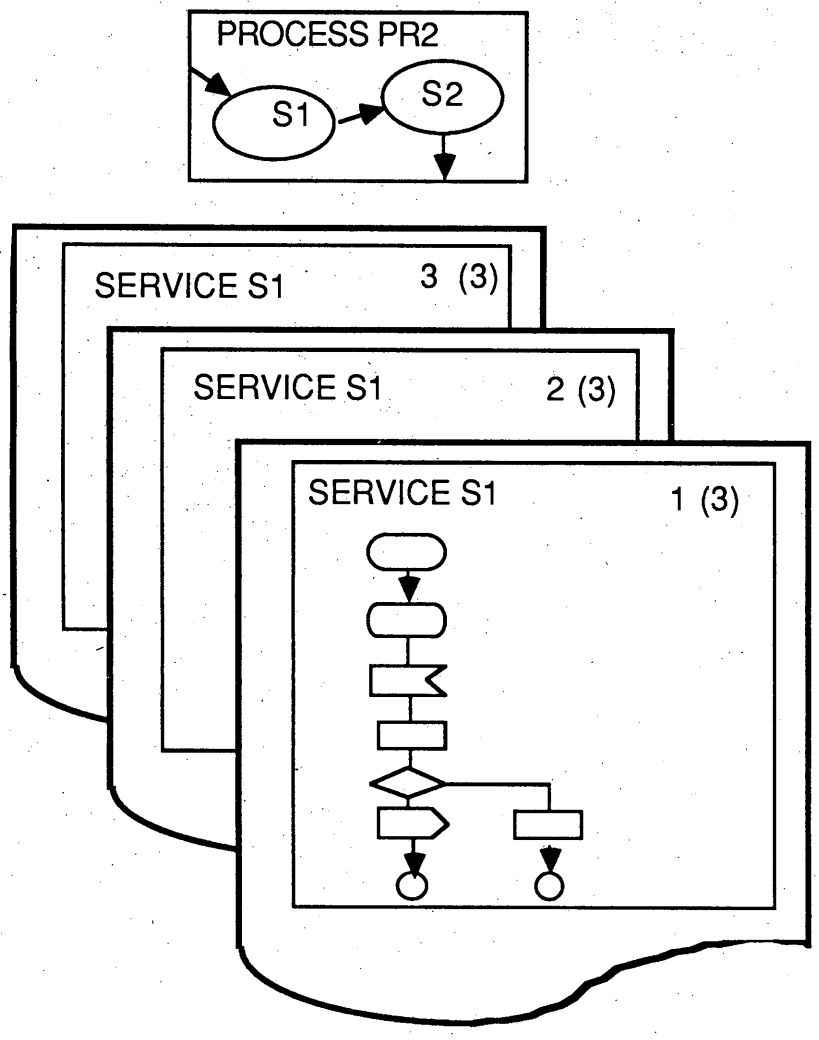


图 D-8.2.4

划分成三页的引用服务图

D.8.3 文件结构

包含整个系统定义的一组文件可以是结构化的。

一个文件结构（此处文件摇—病//考子文件）可以附在系统中的任一实体上（例如系统、功能块和进程），见图 D-8.3.1。

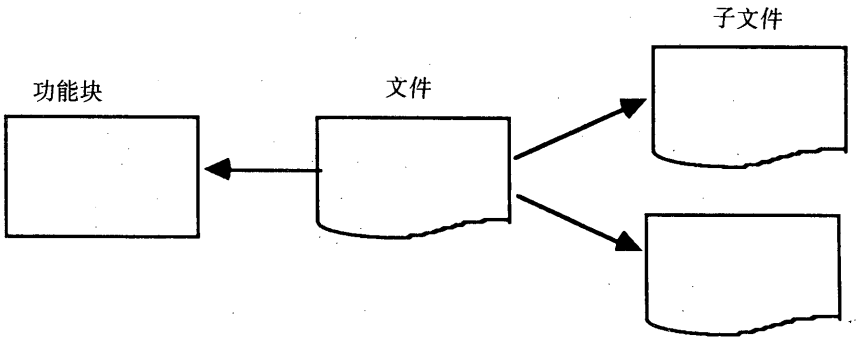


图 D-8.3.1
文件结构

D.8.4 引用机制

语言中的引用机制（这里概念名字用于概念之间的引用）也可用于文件之间的引用。当一个文件对应于一个定义时，很自然会采用这种方法。

D.8.5 文件分类

文件可以按照它们包含的定义的类型来进行分类。

在这样的分类中，用来描述系统动态行为的进程定义或服务定义至少应安排成独立文件。这些文件也可包括变量定义。

图 D-8.5.1给出了一个系统的可能的文件结构。

在这个例子中，信道和信号路由定义被包含在系统、功能块和进程定义的文件中，信号定义、数据定义和信号表定义被安排成独立文件。这里假定所有数据定义都在系统层。

进程定义、宏定义和服务定义组成进程文件的子文件。

如果几个不同的服务组成一系统功能，那么这些服务可以在一公共文件中进行描述。

在一服务文件中多个不同的服务定义可以先后排列，但也可能在同一文件页上并列安排。后一种安排使得服务之间的相互作用更易于理解。图 D-8.5.2是一服务文件中的一个文件页的例子。

对大系统规格应提供多个系统“目录”，用以指明哪里可找到状态、输入、输出等，此外这些目录也应包括所有的概念，即各个定义在哪里?哪些地方用到这些定义?例如系统、功能块、信道、信号、进程、服务、宏、过程等的定义。

这样的系统的多个目录可以分别作为独立的文件来处理。

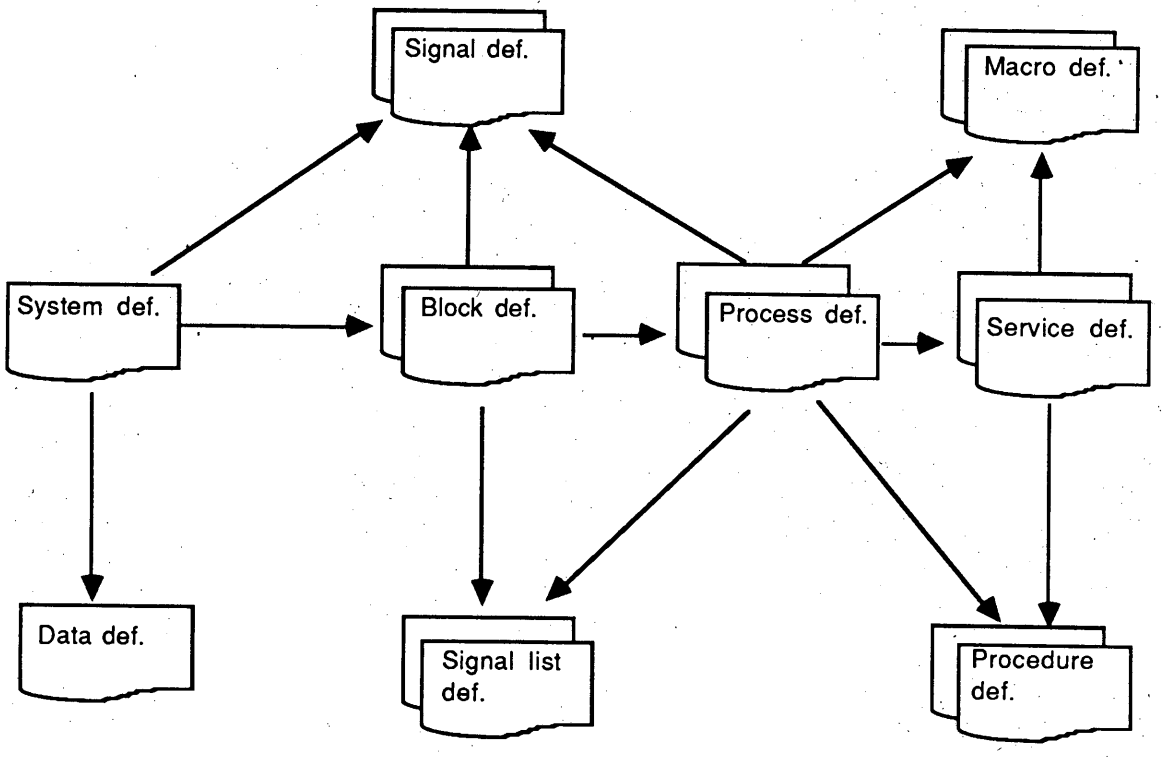


图 D-8.5.1

一个系统的文件结构举例

D.8.6 SDL/GR 和 SDL/PR 的混合运用

在表示一个系统时，SDL/GR 和 SDL/PR 可以同时使用。

系统、功能块、进程、服务、过程、宏、信道等概念可以用 SDL/GR 或 SDL/PR 表示。

从 SDL/PR 转换成 SDL/GR 或从 SDL/GR 转换成 SDL/PR 是采用语言中的引用机制来完成的。在 SDL/PR 中所引用的概念可以用 SDL/GR 来规定，而在 SDL/GR 中所引用的概念又可以用 SDL/PR 来规定。

图 D-8.6.1是一个用 SDL/PR 和 SDL/GR 混合表示的“完整的”系统规格，这是一个与图 D-8.2.1和图 D-8.2.2相同的系统，例中的每个定义可以属于一个独立文件。

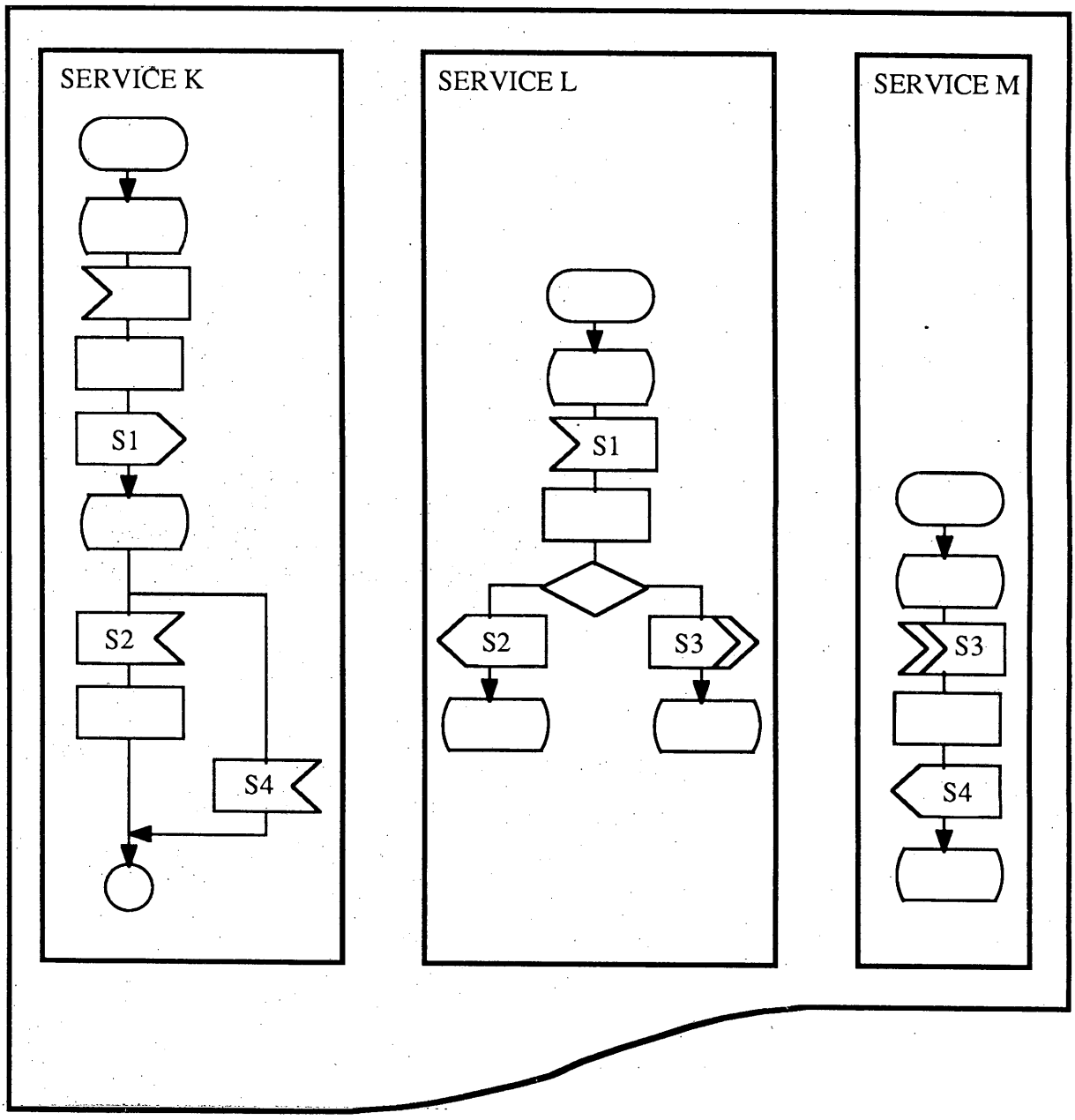
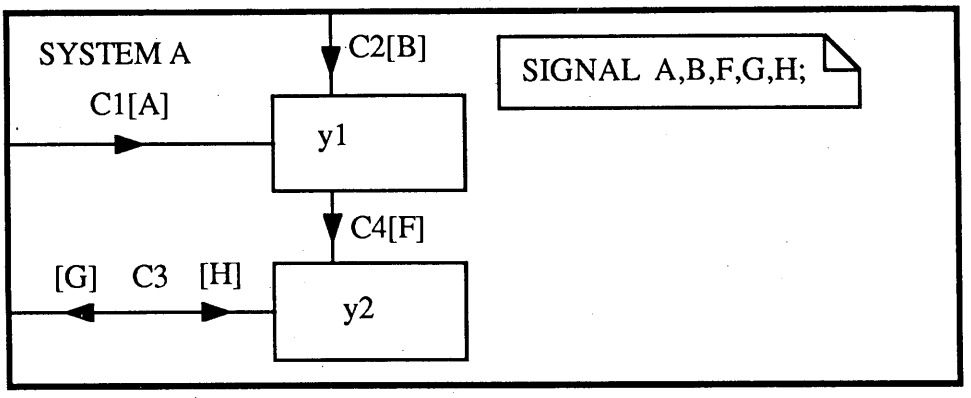
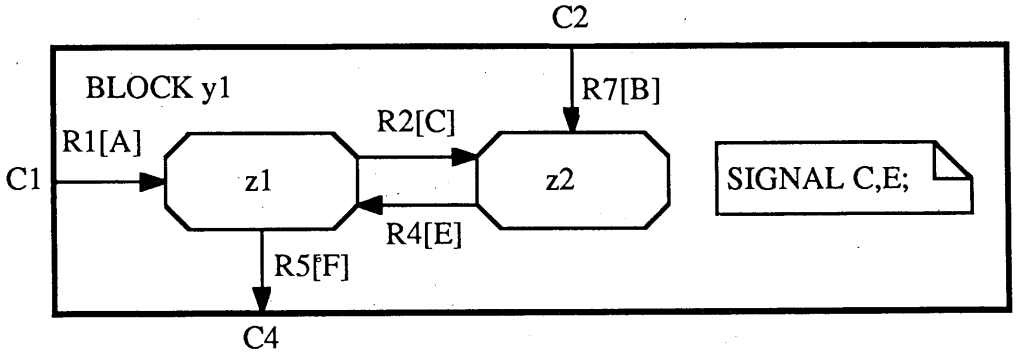


图 D-8. 5. 2

服务文件中的一页






```
BLOCK y2;
   SIGNALROUTE R8FROM ENV TO z3 WITH H;
                  FROM z3 TO ENV WITH G;
   SIGNALROUTE R6FROM ENV TO z3 WITH F;
   CONNECT C3 AND R8;
   CONNECT C4 AND R6;
   PROCESS z3 REFERENCED;
ENDBLOCK y2;
```

```
PROCESS z1;
   SIGNALROUTE IR1 FROM ENV TO k1 WITH A;
   SIGNALROUTE IR2 FROM k1 TO ENV WITH C;
   SIGNALROUTE IR3 FROM k1 TO k2 WITH D;
   SIGNALROUTE IR4 FROM ENV TO k2 WITH E;
   SIGNALROUTE IR5 FROM k2 TO ENV WITH F;
   CONNECT R1 AND IR1;
   CONNECT R2 AND IR2;
   CONNECT R4 AND IR4;
   CONNECT R5 AND IR5;
   SERVICE k1;
      START;
      STATE SS;
         INPUT A;
         OUTPUT C;
         PRIORITY OUTPUT D;
         NEXSTATE SS;
   ENDSERVICE k1;
   SERVICE k2 REFERENCED;
ENDPROCESS z1;
```

图 D-8. 6. 1a

用 SDL/GR 和 SDL/PR 混合表示的系统规格举例

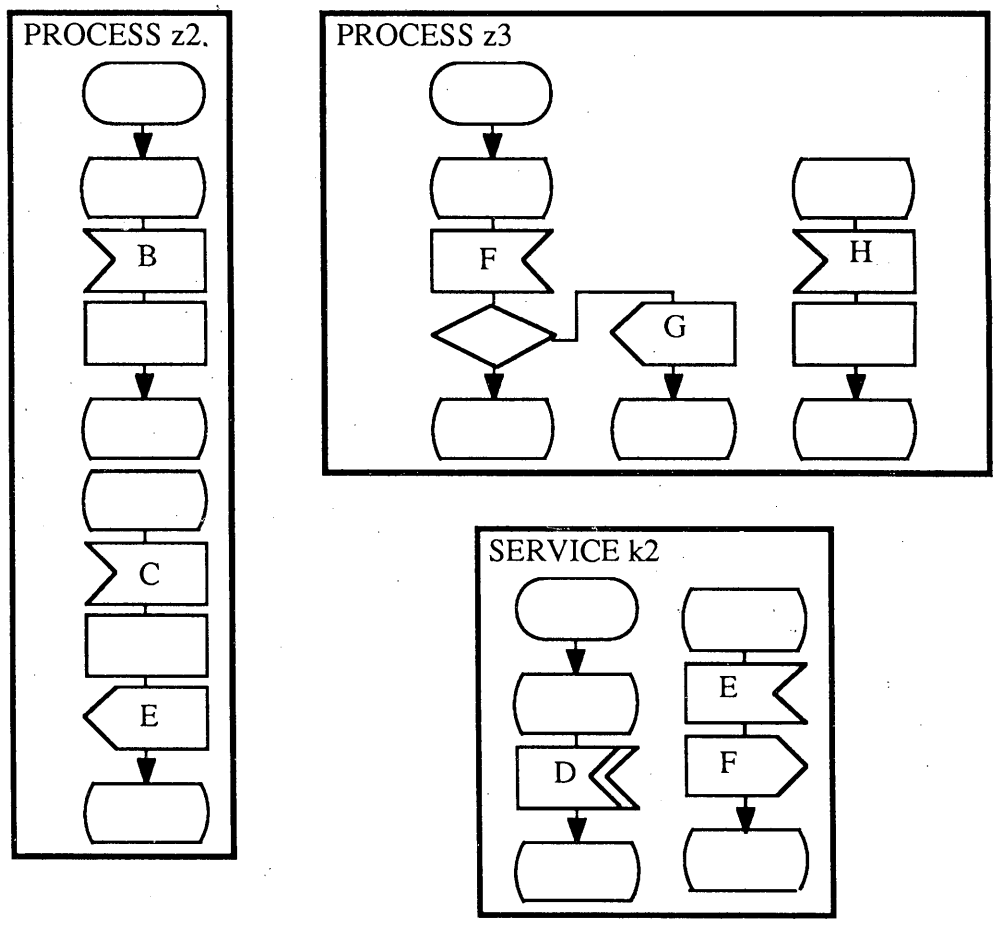


图 D. 8. 6. 1b

用 SDL/GR 和 SDL/PR 混合表示的系统规格举例

D. 9　映射

这一章描述关于 SDL 与 CHILL 之间的映射（§ D. 9. 1）和 SDL/GR 与 SDL/PR 之间的映射（§ D. 9. 2）的一些情况。

D. 9. 1　SDL 和 CHILL 之间的映射

下面说明把 SDL 映射到 CHILL 的某些可能的方法。这里只作简要的而不是详尽的说明，也不打算建议其中的哪一种方法应该在实际中采用。

关于映射的讨论，不应当只考虑已有的 CHILL 编译程序和相应的目标机，而应更广泛一些。映射是一个很复杂的智力活动，只有通过经验，设计者、程序员才能选定一特定的 CHILL 程序结构，用来实现一特定的 SDL 表示。这种情况也适用于用 SDL 表示而由 CHILL 程序实现的功能。在用 SDL 表示各种功能而由 CHILL 来实现时，最好的方式未必是一对一的映射（如果能实现的话）。

运用这种方法，由一个 SDL 图导出的 CHILL 程序的总体结构示于图 D-9. 1. 1。

```
Declaring: MODULE
    /*CHILL 模块，含有在 SDL 图中使用的信号和有关变量*/

    GRANT
    /*信号和变量的移出*/
    SIGNALS
    /*信号定义*/
    SYNMODE (OR NEWMODE)
    /*类型定义*/
END Declaring;

Functional_Block: MODULE
    /*此模块含有 SDL 图的过程部分*/
    SEIZE
    /*移入可以由该功能块接收和发送的所有信号和变量*/

    /*数据定义和声明，这种数据对属于该模块的所有进程是全局性的数据*/

    Process name:PROCESS ( );
        /*局部数据定义和声明*/
        nextstate:=...;
        join:=none;
        DO FOR EVER;
            state_loop: CASE nextstate OF
            /*对于指明该 SDL 状态的变量 nextstate 的循环*/
                (state_label1): RECEIVE CASE
                                    (signal name1):
                                    . . .
                                    (signal namen):
            ESAC state_loop;
            DO WHILE join/=none;
                CASE join OF
                    (join - lab1): join: = none;
                    . . .
                    (join -labm): join: = none;
                ESAC;
            OD;
        OD;
        END process - name;
    END Functional - Block;
```

图 D-9. 1. 1

由一 SDL 图导出的 CHILL 程序的总的结构

在图 D-9.1.2至 D-9.1.5中给出了这两种语言的各种构件的映射的一些例子，它们涉及以下的 SDL 构件：

— 状态和信号的接收与保存，下一状态的选择；

— 输出；

— 汇接；

— 判定。

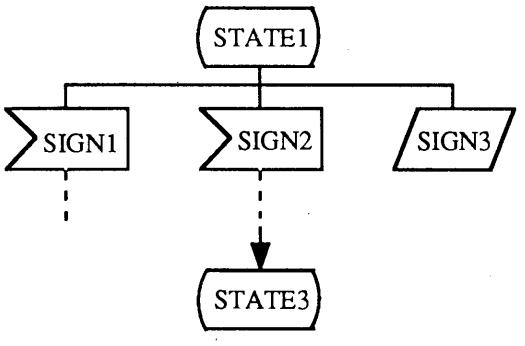
声明模块既包含了被变换的 SDL 图中所用的全部信号的定义和声明，又包含了与这些信号有关的所有变量。所有这些变量都被赋予表示 SDL 图的功能块的模块。

此功能模块用来表达 SDL 各进程的行为（过程部分）。

在这个变换图中，每个 SDL 进程表示为一个无限循环，名为“nextstate”的变量指明待检查的状态，名为“join”的变量指明各可能的汇接点，以确定各组公用语句。

使用 CHILL 的 Case 构件，可以选择 nextstate 的值。case 的每个入口标志着一个 SDL 状态，在每个入口处，要在可能输入的信号之间进行选择，每个输入信号确定一组待完成的动作（“跃迁路径”）。

在每条跃迁路径的终点或者对变量“nextstate”进行赋值，以直接确定待检测的下一状态，或者对变量“join”进行赋值。继变量“join”当前值的一个选择循环后，就 SDL 意义来说，使得每个跃迁结束，并在结束时把一个值赋给变量“nextstate”。



a) SDL

```
STATE1:
  RECEIVE CASE
    (SIGN1): ...
             ...
    (SIGN2): ....
             NEXTSTATE := STATE3;
  ELSE GETOUT (LIST1)
  ESAC STATE1;
```

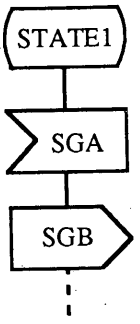
b) CHILL

图 D-9.1.2

映射 STATE/INPUT/SAVE/NEXTSTATE 的例子

在建立 SDL 与 CHILL 的联系中，主要问题之一是信号接收的语义彼此不同，事实上 CHILL 除了所期望的信号（持续信号）外，不消耗（因此不破坏）任何信号；而 SDL 进程却消耗全部接收到的信号，并破坏该状态所不期望的信号。这种语义上的不一致已经通过引入内部例行程序 GETOUT 解决了，它在 CHILL RECEIVE CASE 构件中作为一个选择（ELSE 路径），如图 D-9.1.2所示。CHILL 内部例行程序 GETOUT 能识别（通过参数）那些输入和保存的信号，当调用它时，就破坏进程可得到的其它信号。

在执行 GETOUT 例行程序之后，设置状态选择器，以重复那个状态的循环，直至选择到一个有效的输入信号（或者有一个有效的输入信号到达，如果它原来不存在的话）为止。



a) SDL

```
STATE1:
  RECEIVE CASE
    (SGA): pi := get_instance_value();
           send SGB to pi;
           nextstate := ... ;
    ELSE nextstate := state1;
  ESAC STATE1;
```

b) CHILL

图 D-9.1.3

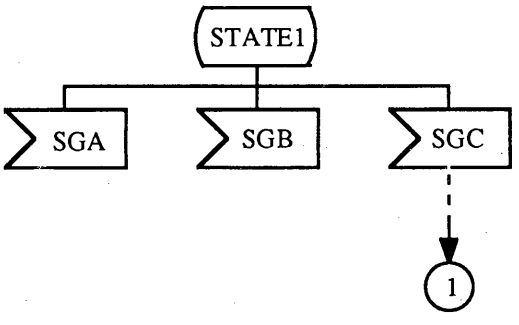
映射 OUTPUT 的例子

例如在图 D-9.1.3中，输入信号 SGA 一经被识别，就选择信号 SGB 的合适的目的进程实例，并发送信号 SGB。

在发送信号 SGB 之前，可能需要装填某些要被信号传送的信息字段。这或者可紧接在信号发送之前进行，或者可以事先装好。

当在图中(见图 D-9.1.4)遇到一汇接点时，就把适当的值赋给变量“JOIN”。按图 D-9.1.1 中解释，执行按照变量“join”的值所定的循环，便可确定待检测的下一个状态。从程序设计语言的观点看，一个汇接点可以看作是一个“goto”构件；把所有的汇接点收集到一起，便于观察并可使整个骨架程序不使用 goto 就能编制，同时也使得程序更易读了。

一个 SDL 判定可以直接翻译成 CHILL 的 case 构件，如图 D-9.1.5所示。



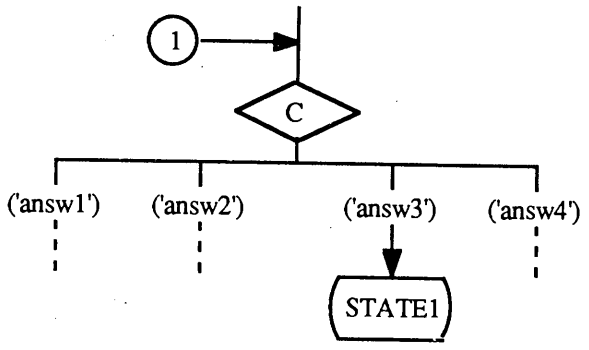
a) SDL

```
STATE1:
  RECEIVE CASE
    (S1 in m): case m.id of
                 (SGA): ... ;
                 (SGB): ... ;
                 (SGC): ... ; JOIN:=1;
                 ELSE nextstate := state1;
               esac;
  ESAC STATE1;
```

b) CHILL

图 D-9.1.4

映射 join 的例子



a) SDL

```
...
1: CASE C OF
    ('answ1'): ... ;
    ('answ2'): ... ;
    ('answ3'): nextstate := state1;
    ('answ4'): ... ;
   ESAC 1;
```

b) CHILL

图 D-9.1.5

映射 DECISION 的例子

D.9.2 GR 和 PR 之间的映射

把有关宏中提到的限制（§D.5.1）考虑进去，GR 形式总是可以被映射到 PR 形式，反过来也一样。

在本书中可以找到很多用这两种形式表示的等效规格的例子。

D.10 应用举例

D.10.1 引言

D.10一节包含了运用 SDL 的一些例子，这些例子取自电信应用领域，其中尽量采用了实际的例子并尽可能多的介绍了 SDL 概念。

这些例子是用来说明如何使用 SDL 的，而不是国际规格。

D.10.2 服务概念

下面的系统是服务概念的一个图示。在这个例子中，系统中的三种“系统功能”具有特别

的意义，它们是话务处理功能、运行操作功能和报警功能。下图 D-10.2.1展示了这些系统功能是如何由五个功能块内的五个不同进程中的一些服务组成的。

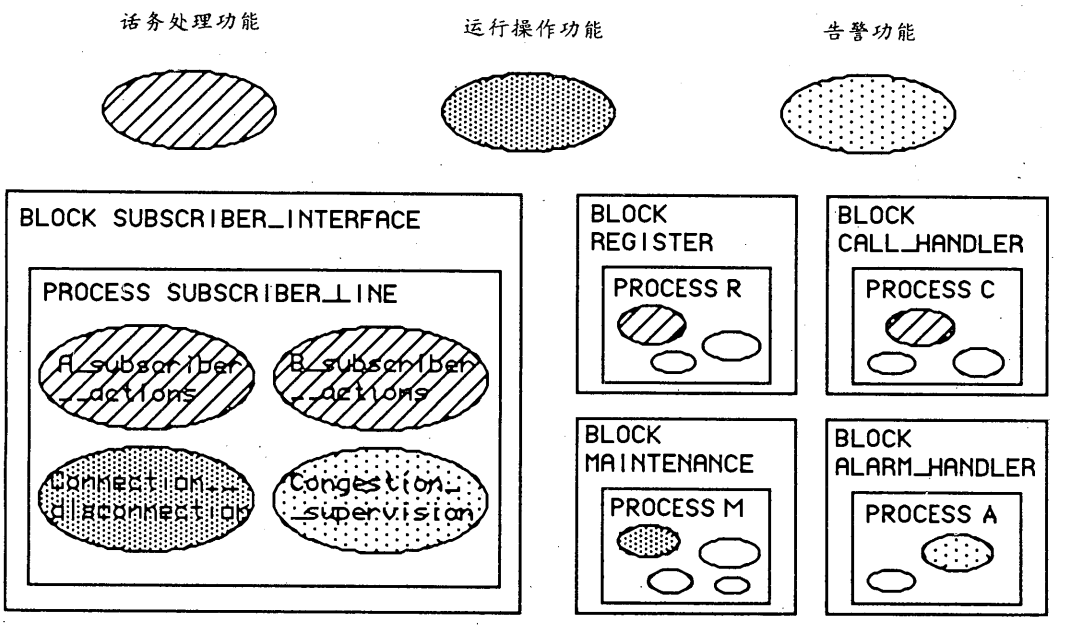


图 D-10.2.1

功能组成

话务处理功能包括电话呼叫的建立和结束。

这个例子不是一个完整的关于系统层功能的文件编制例子，它仅仅是“SUBSCRIBER_LINE”进程中四个服务的一个说明。

下面三个图是系统图 (D-10.2.2)，“SUBSCRIBER_INTERFACE”功能块图 (D-10.2.3) 和“SUBSCRIBER_LINE”进程图 (D-10.2.4)。必要的信道、信号路由和信号已在这些图中给出了。

```
SYSTEM Service_concept

/* 该系统是表示 SDL 中服务概念的一个例子，选择了功能块
SUBSCRIBER_INTERFACE 作为图例。系统图展示了相互配
合工作的各功能块和在 SUBSCRIBER_INTERFACE 中的服务
所需的信道*/

SIGNAL A_OFF_HOOK,A_ON_HOOK,DIALLED_DIGIT,B_ON_HOOK,B_OFF_HOOK,
  DIAL_TONE,CONG_TONE,DIAL_TONE_OFF,CONG_TONE_OFF,RING_TONE_TO_A,
  RING_TONE_A_OFF,RING_SIG_TO_B,RING_SIG_B_OFF,CONNECT_DIG_REC,DIGIT,
  DISCONN_DIG_REC,CONNECTION,CONGESTION,FETCH_NEXT_DIGIT,
  CONNECT_CALL_HANDLER,A_OFF,A_ON,LINE_CONNECTED,LINE_DISCONNECTED,
  SEND_RING_TONE,B_ANSWER,DISC_A,RING_SIG,DISC_B,SEND_ALARM,
  CEASE_ALARM,CONN_REQ_ACK1,CONN_REQ_ACK2,DISC_REQ_ACK1,
  DISC_REQ_ACK2,CONN_REQ,DISC_REQ,B_ON,B_OFF;

SIGNALLIST L1A = A_OFF_HOOK,A_ON_HOOK,DIALLED_DIGIT;
SIGNALLIST L1B = B_ON_HOOK,B_OFF_HOOK;
SIGNALLIST L2A = DIAL_TONE,CONG_TONE,DIAL_TONE_OFF,CONG_TONE_OFF,
                 RING_TONE_TO_A,RING_TONE_A_OFF;
SIGNALLIST L2B = RING_SIG_TO_B,RING_SIG_B_OFF;
SIGNALLIST LAL = SEND_ALARM,CEASE_ALARM;
SIGNALLIST L1MAIN = CONN_REQ,DISC_REQ;
SIGNALLIST L2MAIN = CONN_REQ_ACK1,CONN_REQ_ACK2,DISC_REQ_ACK1,
                    DISC_REQ_ACK2;
SIGNALLIST L1REG = CONNECT_DIG_REC,DIGIT,DISCONN_DIG_REC;
SIGNALLIST L2REG = CONNECTION,CONGESTION,FETCH_NEXT_DIGIT;
SIGNALLIST L1CALL = SEND_RING_TONE,B_ANSWER,DISC_A;
SIGNALLIST L2CALL = A_OFF,A_ON;
SIGNALLIST L3CALL = RING_SIG,DISC_B;
SIGNALLIST L4CALL = LINE_CONNECTED,LINE_DISCONNECTED,B_ON,B_OFF;
```

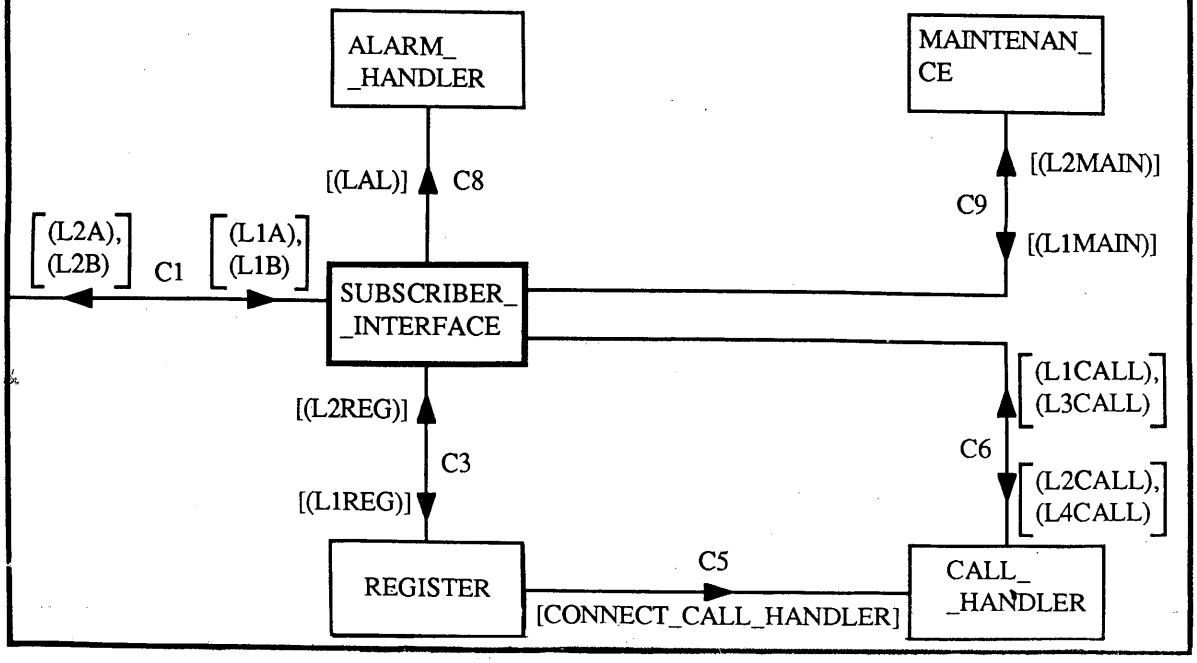


图 D-10. 2. 2

系统图

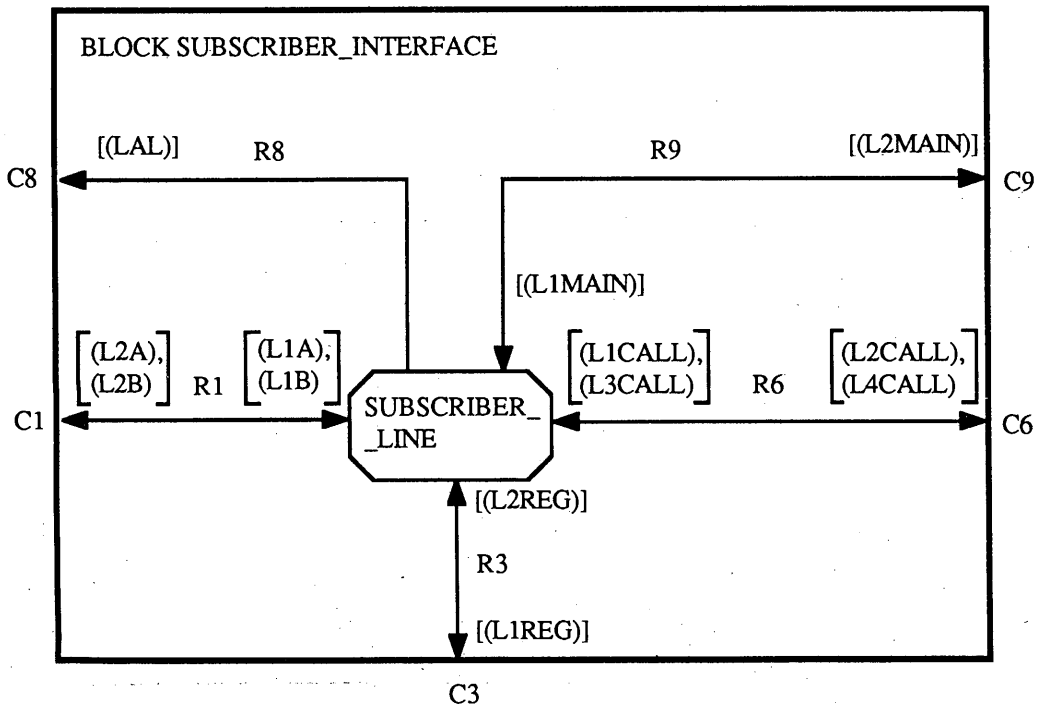


图 D-10.2.3

功能块图

从进程图（D-10.2.4）中可以见到，各个服务通过信号路由 IR01、IR02、IR03和 IR04与优先信号相互配合工作。但各服务也通过在进程中声明的“全局”变量“Connected”彼此相互作用，相互影响。

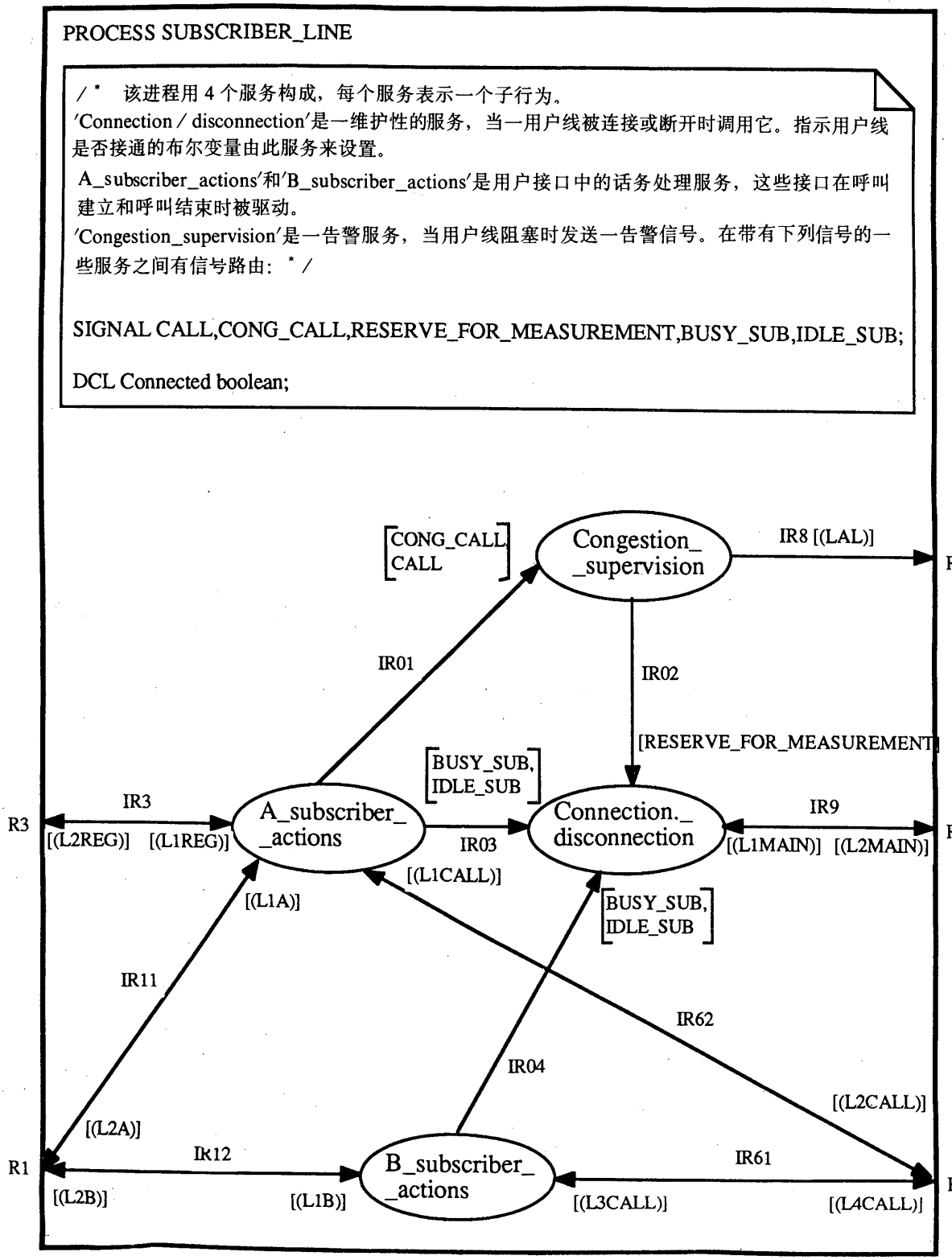

PROCESS SUBSCRIBER_LINE
/* 该进程用 4 个服务构成，每个服务表示一个子行为。
'Connection / disconnection'是一维护性的服务，当一用户线被连接或断开时调用它。指示用户线是否接通的布尔变量由此服务来设置。
A_subscriber_actions'和'B_subscriber_actions'是用户接口中的话务处理服务，这些接口在呼叫建立和呼叫结束时被驱动。
'Congestion_supervision'是一告警服务，当用户线阻塞时发送一告警信号。在带有下列信号的一些服务之间有信号路由: */
SIGNAL CALL,CONG_CALL,RESERVE_FOR_MEASUREMENT,BUSY_SUB,IDLE_SUB;
DCL Connected boolean;
CONG_CALL
CALL
Congestion_
_supervision
IR8 [(LAL)]
R8
IR01
IR02
[RESERVE_FOR_MEASUREMENT]
BUSY_SUB,
IDLE_SUB
IR3
R3
[(L2REG)] [(L1REG)]
A_subscriber_
_actions
IR03
[(L1CALL)]
Connection._
disconnection
IR9
[(L1MAIN)] [(L2MAIN)]
R9
[(L1A)]
BUSY_SUB,
IDLE_SUB
IR11
IR62
IR04
[(L2CALL)]
[(L2A)]
IR12
R1
[(L2B)]
[(L1B)]
B_subscriber_
_actions
IR61
[(L3CALL)]
[(L4CALL)]
R6

图 D-10. 2. 4

进程图

为了对服务以及对功能块之间的相互作用有更好的理解，例子中给出了若干顺序图。

前两个顺序图展示了在一次呼叫期间各功能块之间相互作用的正常情况。第三个图展示了在记发器阻塞时的相互作用。为了简化顺序图，假定发送与接收信号之间没有迟延。

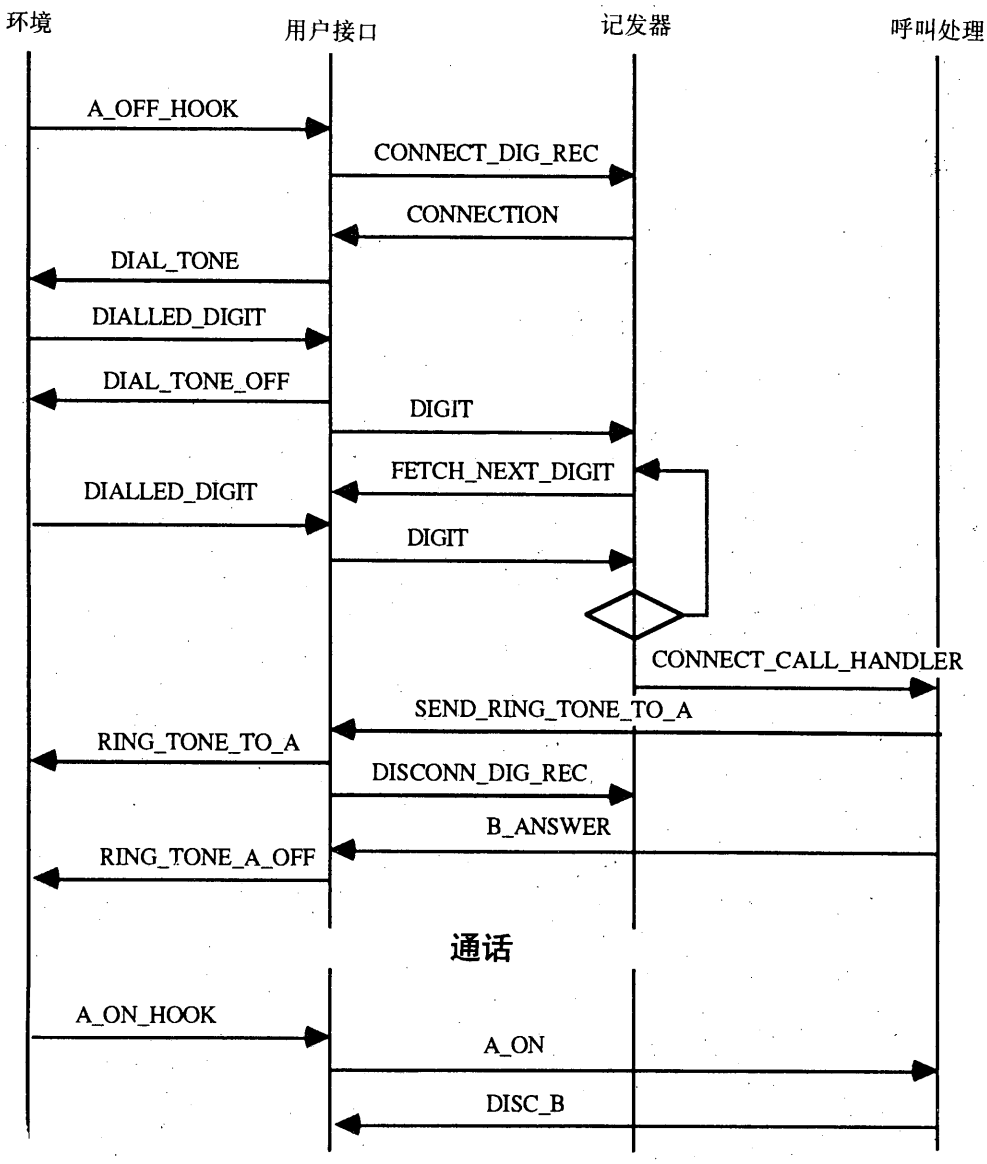
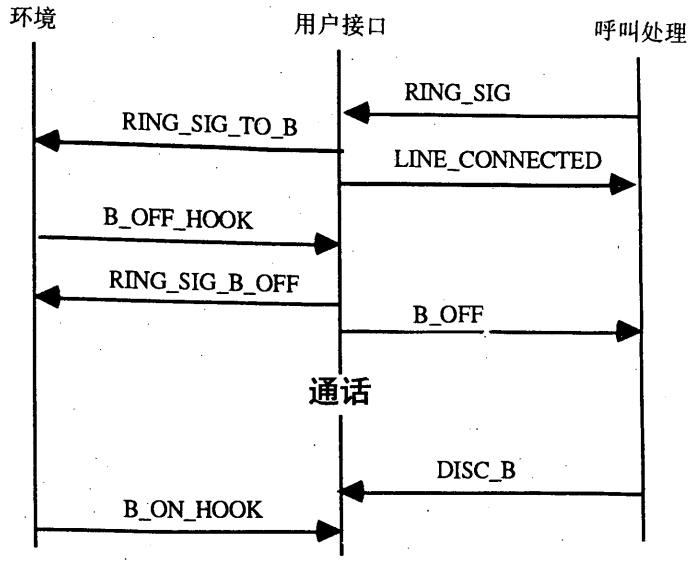


图 D-10. 2. 5

顺序图. 正常情况下的功能块相互作用. 对 A 用户方必需的信号



注：顺序图。正常情况下的功能块相互作用。B 用户所必需的信号。

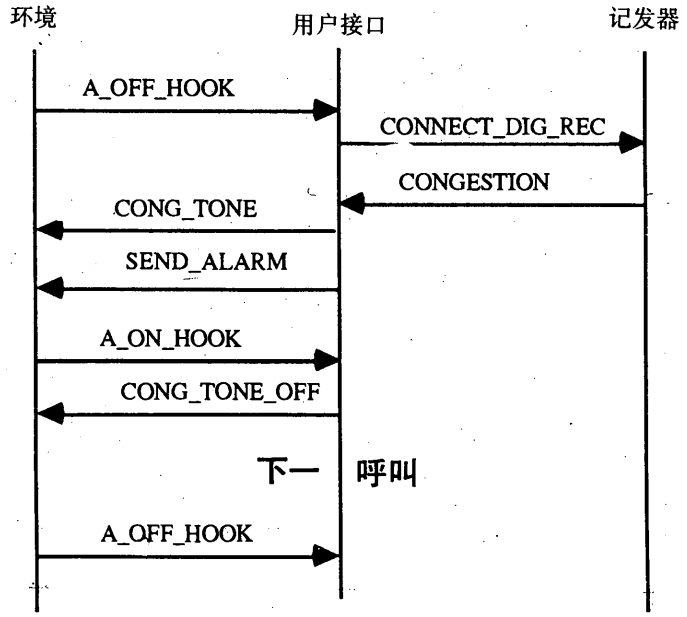
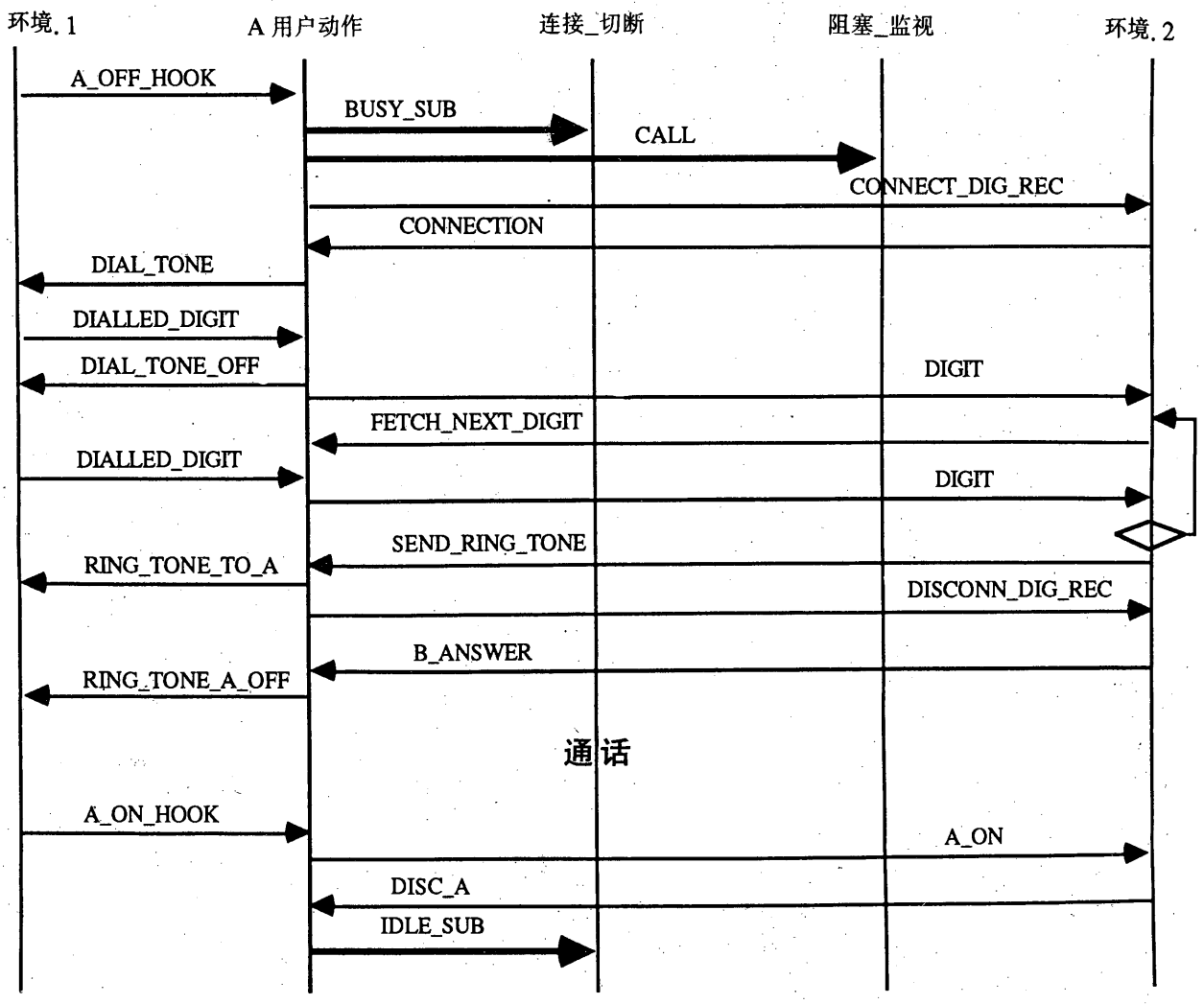


图 D-10.2.6

顺序图. 在记发器阻塞时的功能块相互作用

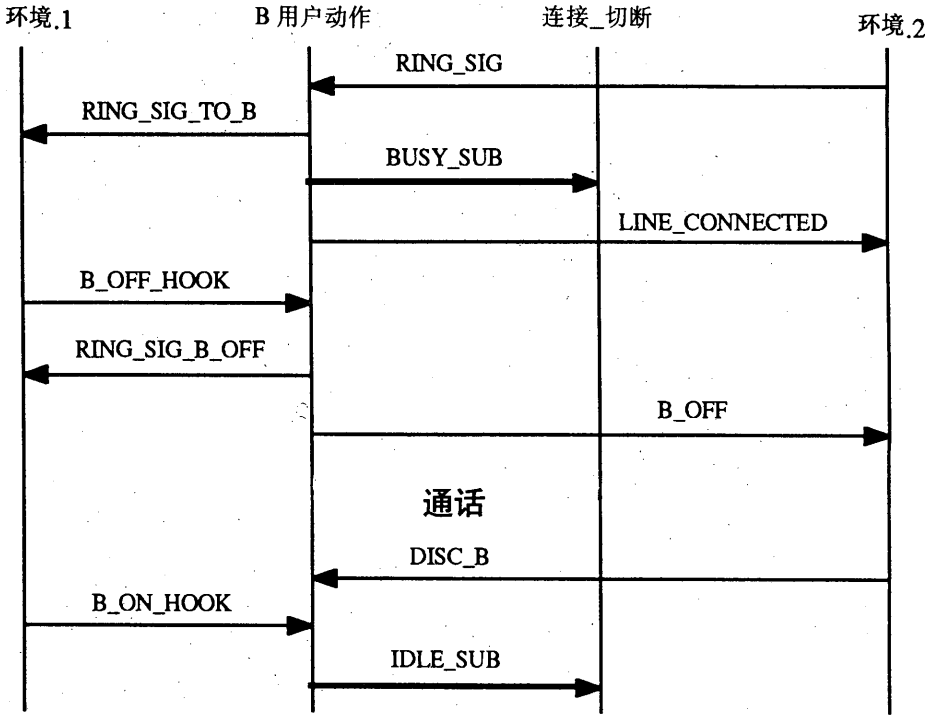
下面的几张顺序图描述了进程“SUBSCRIBER_LINE”中各服务之间的相互作用。



注: 用粗线指明优先信号。

图 D-10. 2. 7

顺序图. 正常情况下服务的相互作用. 对 A 用户方必需的信号



注：用粗线指明优先信号。

图 D-10.2.8

顺序图.正常情况下服务相互作用.对 B 用户方必需的信号

在四个服务图中，(D-10.2.9～15) 中，描述了进程“SUBSCRIBER_LINE”中的每个服务的行为。

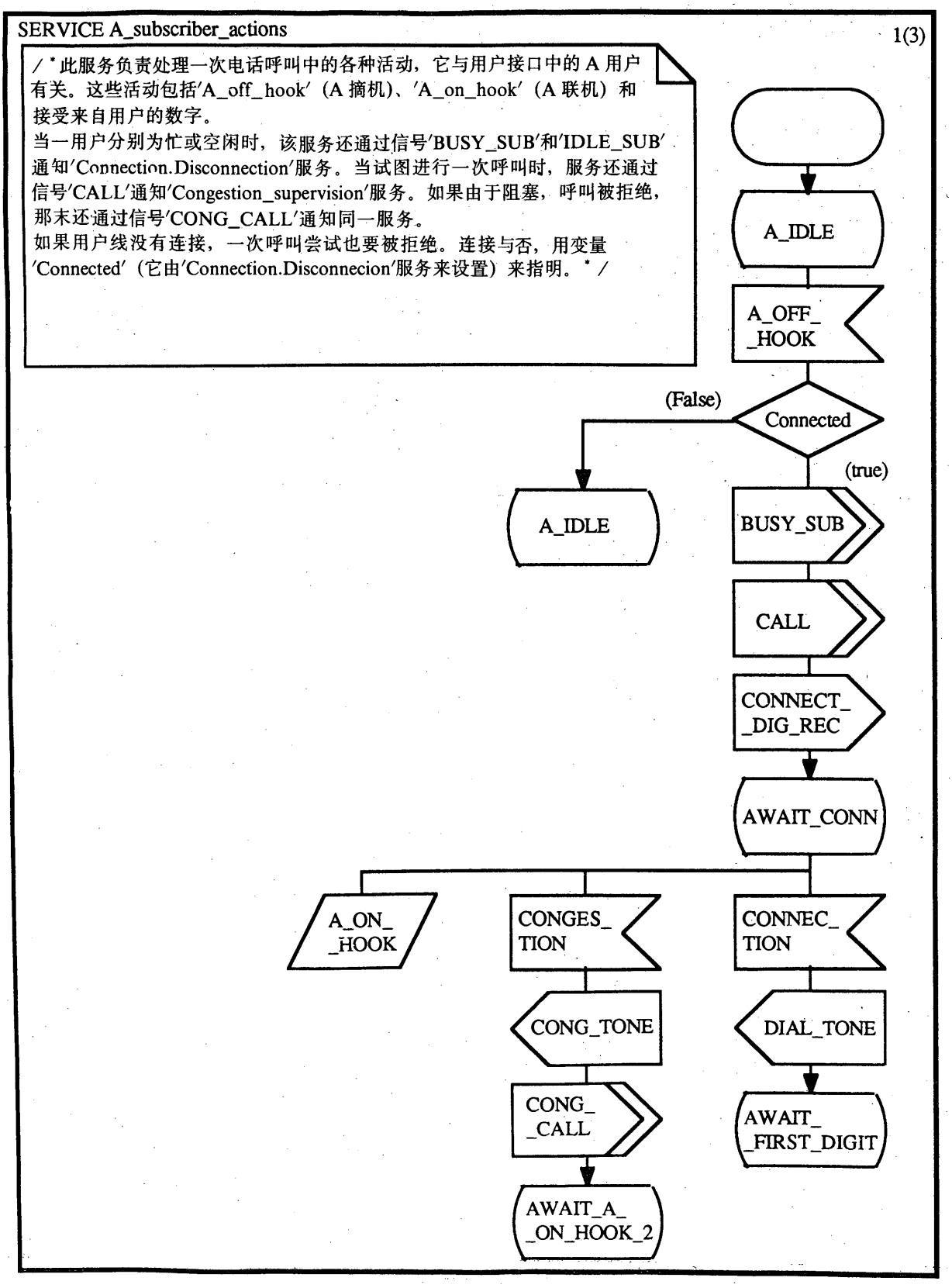

SERVICE A_subscriber_actions
1(3)
/ * 此服务负责处理一次电话呼叫中的各种活动，它与用户接口中的 A 用户有关。这些活动包括'A_off_hook'（A 摘机）、'A_on_hook'（A 联机）和接受来自用户的数字。
当一用户分别为忙或空闲时，该服务还通过信号'BUSY_SUB'和'IDLE_SUB'通知'Connection.Disconnection'服务。当试图进行一次呼叫时，服务还通过信号'CALL'通知'Congestion_supervision'服务。如果由于阻塞，呼叫被拒绝，那末还通过信号'CONG_CALL'通知同一服务。
如果用户线没有连接，一次呼叫尝试也要被拒绝。连接与否，用变量'Connected'（它由'Connection.Disconnecion'服务来设置）来指明。 * /
A_IDLE
A_OFF_ _HOOK
Connected
(False)
(true)
A_IDLE
BUSY_SUB
CALL
CONNECT_ _DIG_REC
AWAIT_CONN
A_ON_ _HOOK
CONGES_ TION
CONNEC_ TION
CONG_TONE
DIAL_TONE
CONG_ _CALL
AWAIT_ _FIRST_DIGIT
AWAIT_A_ _ON_HOOK_2

图 D-10. 2. 9

服务图

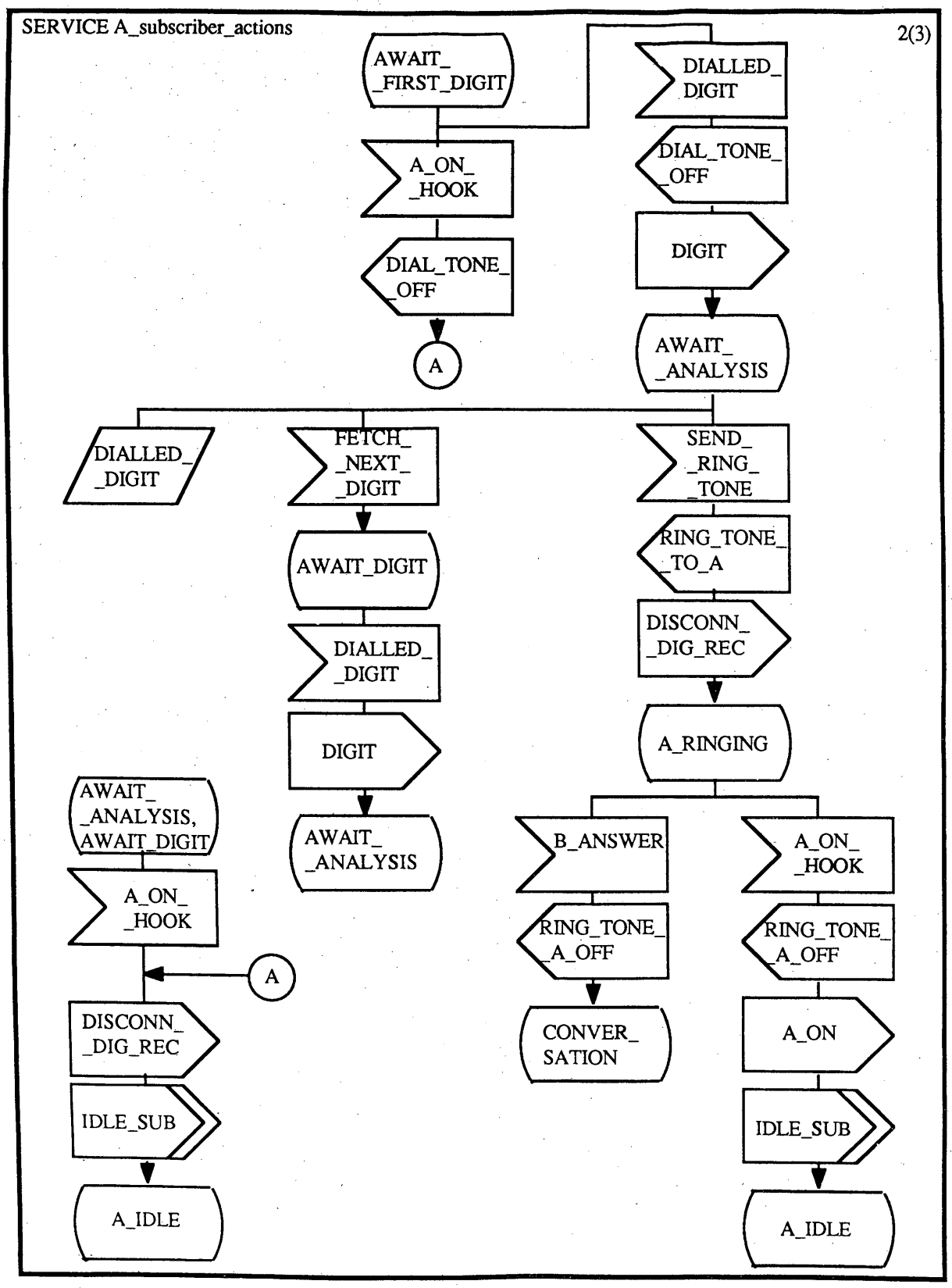

SERVICE A_subscriber_actions
2(3)
AWAIT_ _FIRST_DIGIT
A_ON_ _HOOK
DIAL_TONE_ _OFF
A
DIALLED_ DIGIT
DIAL_TONE_ _OFF
DIGIT
AWAIT_ _ANALYSIS
DIALLED_ _DIGIT
FETCH_ _NEXT_ _DIGIT
AWAIT_DIGIT
DIALLED_ _DIGIT
DIGIT
AWAIT_ _ANALYSIS
SEND_ _RING_ _TONE
RING_TONE_ _TO_A
DISCONN_ _DIG_REC
A_RINGING
B_ANSWER
RING_TONE_ _A_OFF
CONVER_ SATION
A_ON_ _HOOK
RING_TONE_ _A_OFF
A_ON
IDLE_SUB
A_IDLE
AWAIT_ _ANALYSIS, AWAIT_DIGIT
A_ON_ _HOOK
A
DISCONN_ _DIG_REC
IDLE_SUB
A_IDLE

图 D-10. 2. 10

服务图

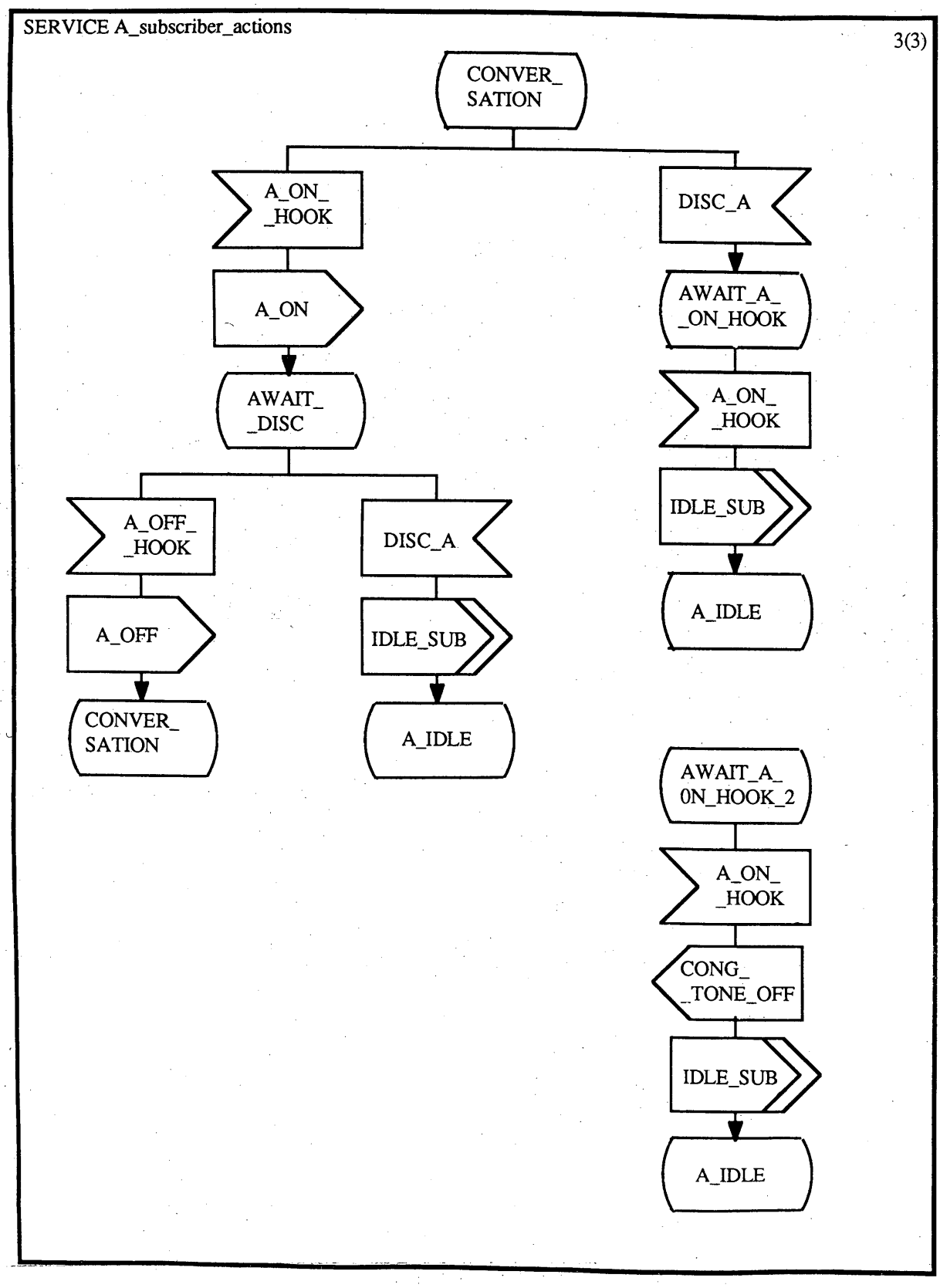

SERVICE A_subscriber_actions
3(3)
CONVER_
SATION
A_ON_
_HOOK
A_ON
AWAIT_
_DISC
A_OFF_
_HOOK
A_OFF
CONVER_
SATION
DISC_A
IDLE_SUB
A_IDLE
DISC_A
AWAIT_A_
_ON_HOOK
A_ON_
_HOOK
IDLE_SUB
A_IDLE
AWAIT_A_
0N_HOOK_2
A_ON_
_HOOK
CONG_
_TONE_OFF
IDLE_SUB
A_IDLE

图 D-10. 2. 11

服务图

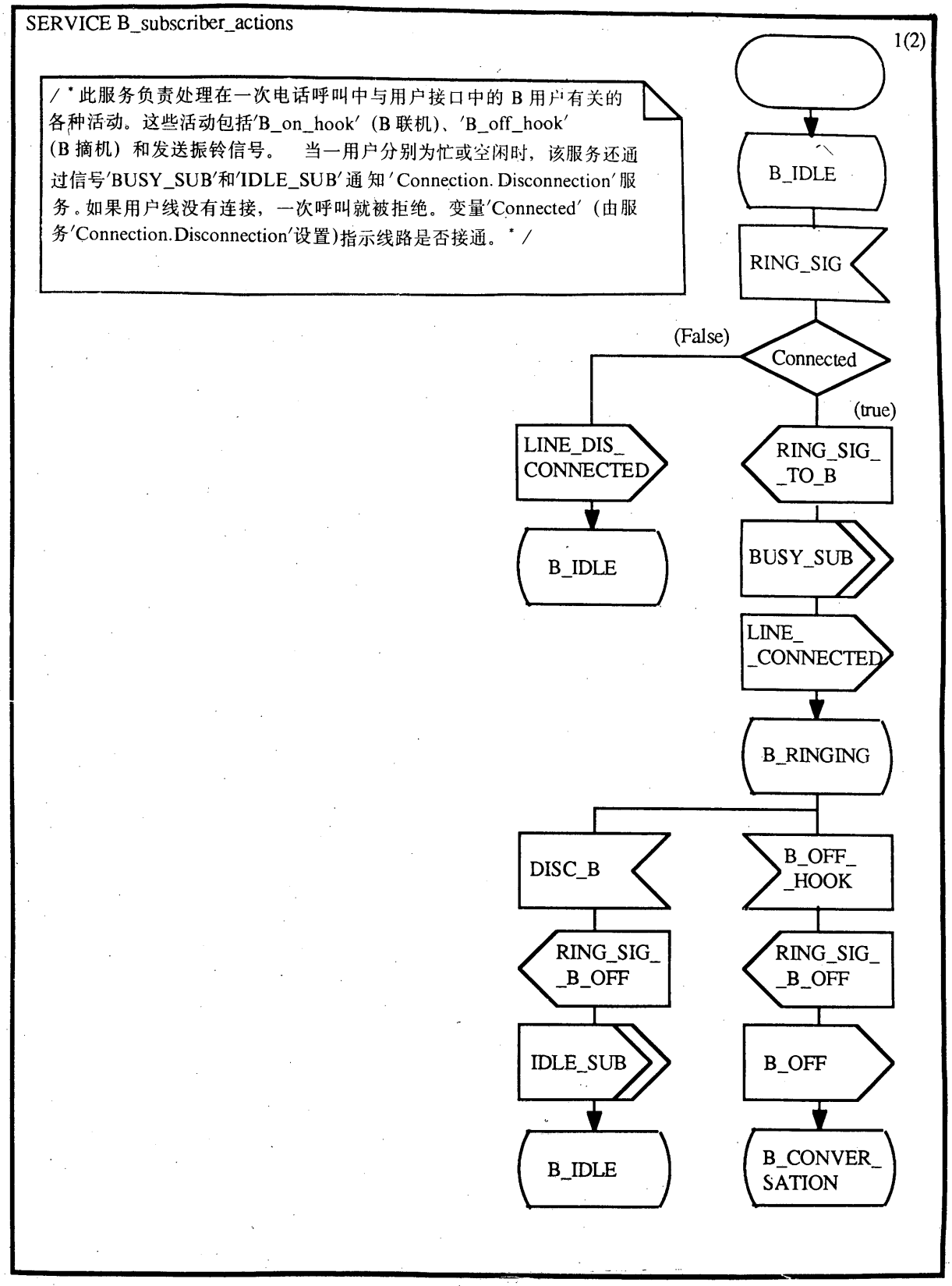

SERVICE B_subscriber_actions
1(2)
/ * 此服务负责处理在一次电话呼叫中与用户接口中的 B 用户有关的各种活动。这些活动包括'B_on_hook' (B 联机)、'B_off_hook' (B 摘机) 和发送振铃信号。 当一用户分别为忙或空闲时，该服务还通过信号'BUSY_SUB'和'IDLE_SUB'通知'Connection. Disconnection'服务。如果用户线没有连接，一次呼叫就被拒绝。变量'Connected' (由服务'Connection.Disconnection'设置)指示线路是否接通。 * /
B_IDLE
RING_SIG
(False)
Connected
(true)
LINE_DIS_ CONNECTED
RING_SIG_ _TO_B
B_IDLE
BUSY_SUB
LINE_ _CONNECTED
B_RINGING
DISC_B
B_OFF_ _HOOK
RING_SIG_ _B_OFF
RING_SIG_ _B_OFF
IDLE_SUB
B_OFF
B_IDLE
B_CONVER_ SATION

图 D-10. 2. 12

服务图

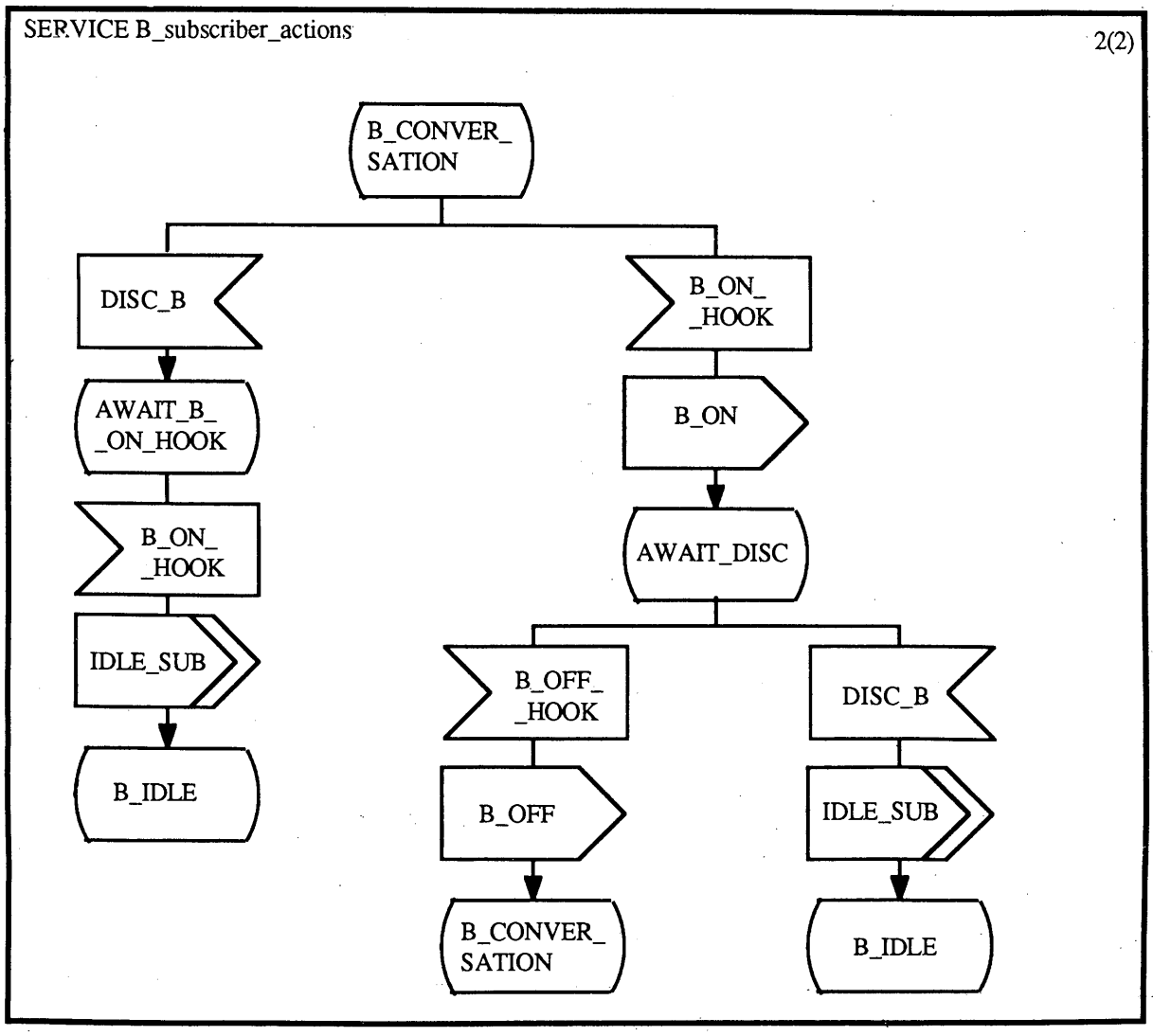

SERVICE B_subscriber_actions
2(2)
B_CONVER_
SATION
DISC_B
AWAIT_B_
_ON_HOOK
B_ON_
_HOOK
IDLE_SUB
B_IDLE
B_ON_
_HOOK
B_ON
AWAIT_DISC
B_OFF_
_HOOK
B_OFF
B_CONVER_
SATION
DISC_B
IDLE_SUB
B_IDLE

图 D-10. 2. 13

服务图

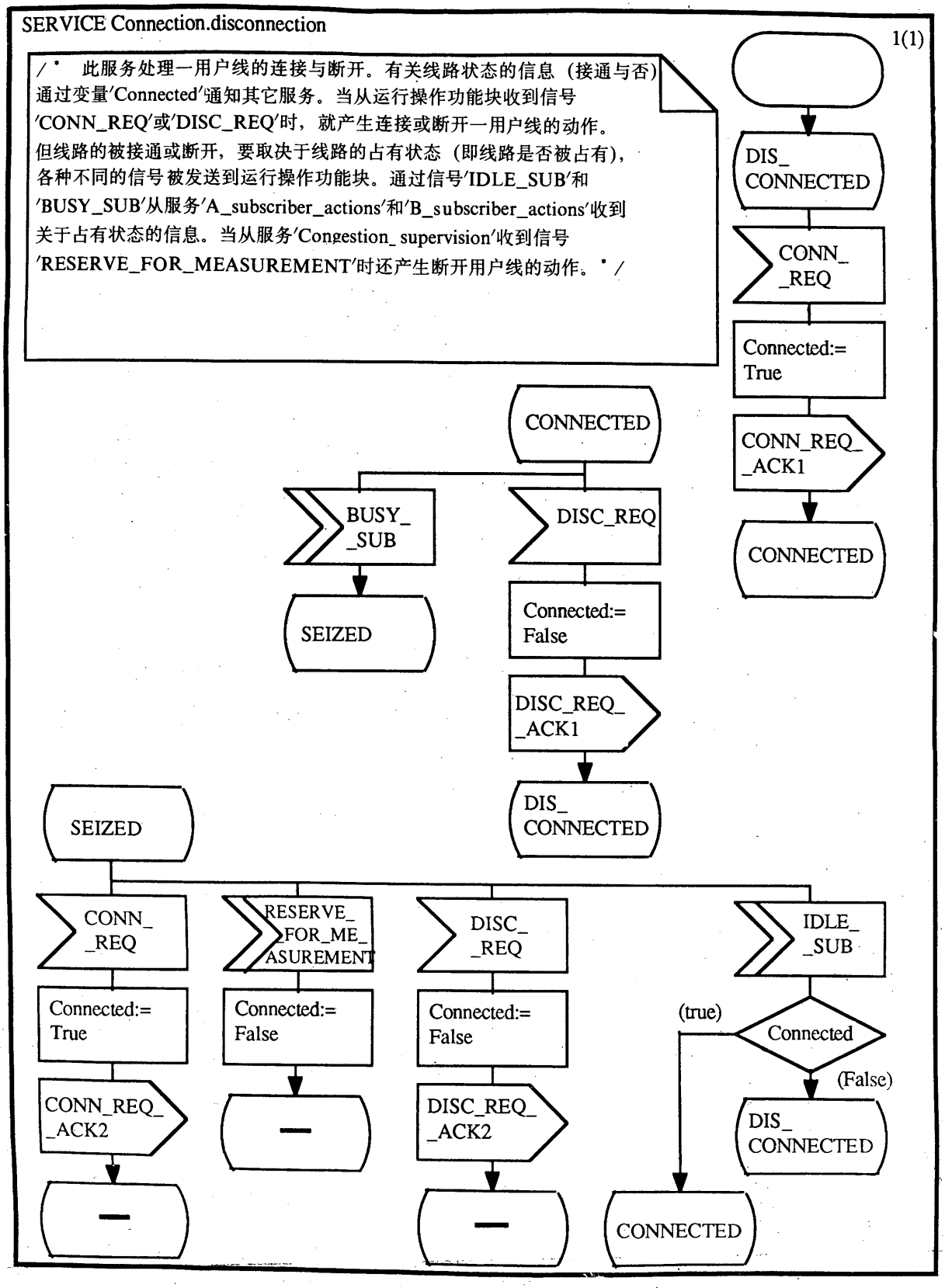

SERVICE Connection.disconnection
1(1)
/* 此服务处理一用户线的连接与断开。有关线路状态的信息（接通与否）通过变量'Connected'通知其它服务。当从运行操作功能块收到信号'CONN_REQ'或'DISC_REQ'时，就产生连接或断开一用户线的动作。但线路的被接通或断开，要取决于线路的占有状态（即线路是否被占有），各种不同的信号被发送到运行操作功能块。通过信号'IDLE_SUB'和'BUSY_SUB'从服务'A_subscriber_actions'和'B_subscriber_actions'收到关于占有状态的信息。当从服务'Congestion_supervision'收到信号'RESERVE_FOR_MEASUREMENT'时还产生断开用户线的动作。*/
DIS_ CONNECTED
CONN_ _REQ
Connected:= True
CONN_REQ_ _ACK1
CONNECTED
CONNECTED
BUSY_ _SUB
SEIZED
DISC_REQ
Connected:= False
DISC_REQ_ _ACK1
DIS_ CONNECTED
SEIZED
CONN_ _REQ
Connected:= True
CONN_REQ_ _ACK2
RESERVE_ _FOR_ME_ ASUREMENT
Connected:= False
DISC_ _REQ
Connected:= False
DISC_REQ_ _ACK2
IDLE_ _SUB
(true)
Connected
(False)
DIS_ CONNECTED
CONNECTED

图 D-10. 2. 14

服务图

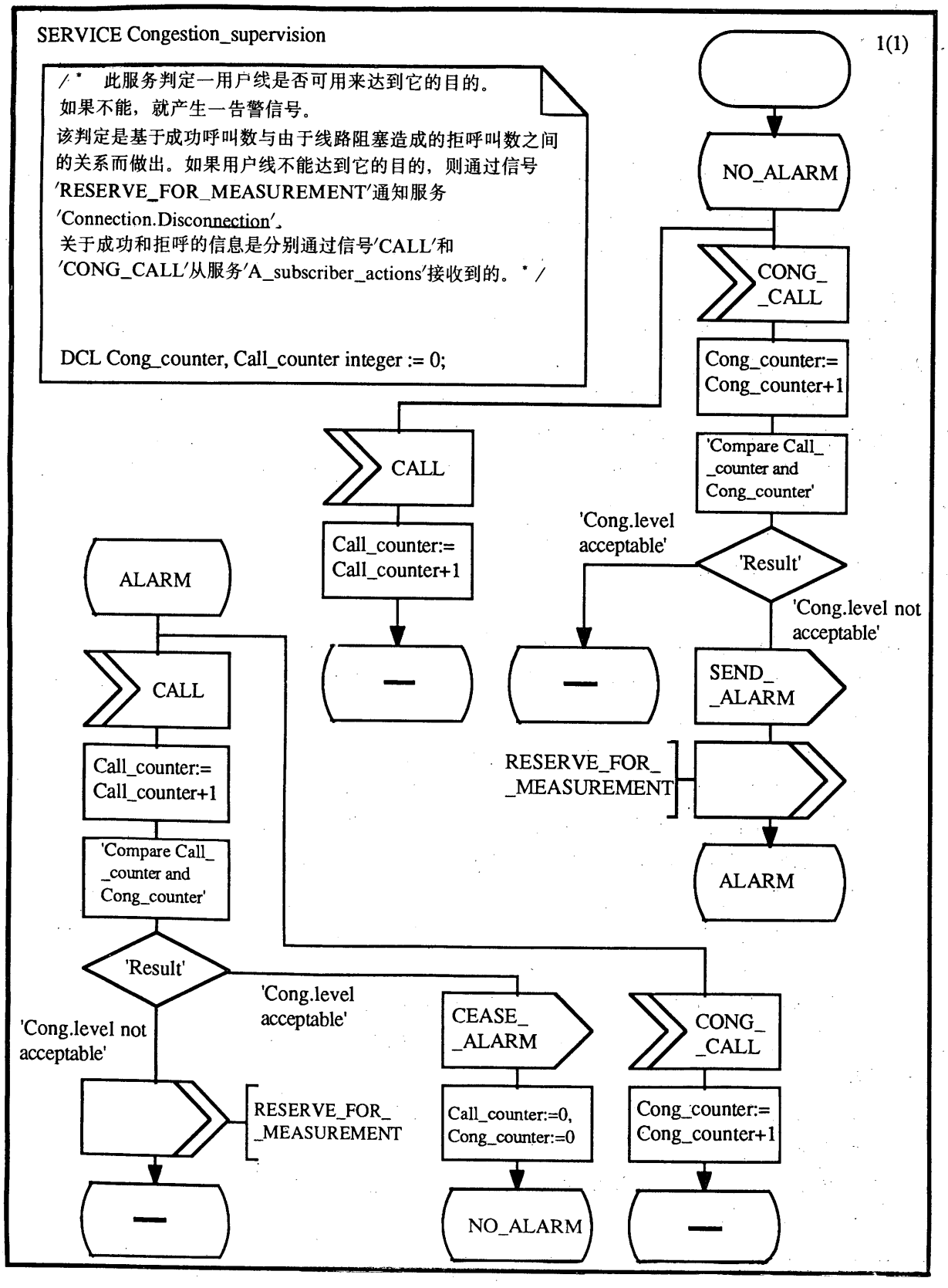

SERVICE Congestion_supervision
1(1)
/* 此服务判定一用户线是否可用来达到它的目的。
如果不能，就产生一告警信号。
该判定是基于成功呼叫数与由于线路阻塞造成的拒呼叫数之间
的关系而做出。如果用户线不能达到它的目的，则通过信号
'RESERVE_FOR_MEASUREMENT'通知服务
'Connection.Disconnection'。
关于成功和拒呼的信息是分别通过信号'CALL'和
'CONG_CALL'从服务'A_subscriber_actions'接收到的。*/
DCL Cong_counter, Call_counter integer := 0;
NO_ALARM
CONG_
_CALL
Cong_counter:=
Cong_counter+1
'Compare Call_
_counter and
Cong_counter'
'Result'
'Cong.level
acceptable'
'Cong.level not
acceptable'
SEND_
_ALARM
RESERVE_FOR_
_MEASUREMENT
ALARM
CALL
Call_counter:=
Call_counter+1
ALARM
CALL
Call_counter:=
Call_counter+1
'Compare Call_
_counter and
Cong_counter'
'Result'
'Cong.level
acceptable'
'Cong.level not
acceptable'
RESERVE_FOR_
_MEASUREMENT
CEASE_
_ALARM
Call_counter:=0,
Cong_counter:=0
NO_ALARM
CONG_
_CALL
Cong_counter:=
Cong_counter+1

图 D-10. 2. 15

服务图

D.11 SDL 工具

D.11.1 引言

本节介绍一组可用来支持 SDL 的工具，这些工具可以支持生成各种文件、SDL 图（SDL/GR）或语句（SDL/PR），和（或）验证 SDL 规格的正确性。

在本指南中没有把所有可能的工具列出一个完全的清单。所需要的工具取决于用户选择的方法。

原则上，在没有任何工具的情况下也能使用 SDL，然而，由于现代系统的内在复杂性，使得 SDL 规格往往很复杂。因此需要有一些自动工具来帮助对许多系统制定规格、进行设计和编制文件。例如，一个交换局的图形文件编制工作如果仅靠人工来制图及改进，则过程将是复杂的且费用高，如使用适当的计算机辅助手段将会大大降低其复杂性和成本。

由于以上的原因，在设计 SDL 时已考虑到使人们能够有效地使用一些工具来支持它的使用。

D.11.2 工具的分类

SDL 工具可以根据产生 SDL 文件时所进行的工作来分类，例如：

— 输入工具：按照已有的语法形式，将输入的辅助手段用于输入 SDL 流图、正文短语及图形等形式。

— 语法检查工具：包括两种语法的语法分析程序。

— 文件生成工具：SDL 文件一旦被贮存在计算机内，这些工具就可以对它们进行访问和复制，但可能要用到若干外部设备。所使用的语法形式可以与输入文件所用的语法不同。此外，有些工具也可以根据原先进入的文件类型产生出新的文件类型。

— 系统模拟和分析工具：表示一个系统的 SDL 文件可用来抽象出系统的模型。在该模型上可以进行检查。通过它们能够寻找死锁，能够在同一系统的各种模型之间（例如，规格和描述之间）作比较，或者进行系统行为的模拟等。

— 支持代码生成的工具：一些很详细的 SDL 规格可用来帮助软件编程。可以开发一些工具，使之能实现按准则设计的半自动的代码序列。

一类特殊而有用的工具是：

— SDL 训练工具：这些工具可以单独使用，或与其它工具结合在一起使用。后一种情况允许使用其它功能的用户在需要时得到帮助。

由于 SDL 被用于一个系统有效期间的各个不同阶段，因而在一个综合的工程环境中，人们会容易地见到所有各类工具的应用。

D.11.3 文件输入

从输入的观点看，由于SDL/PR是与任何字符串输入等效的，所以对输入SDL的正文短语形式没有特殊的要求，为方便起见，可以利用输入字符串的工具（行编辑程序）。然而如果采用SDL/GR语法就需要有图形处理功能。

显然，尽管对SDL/PR输入的支持可能是方便的，但如果期望用这两种语法作为输入，那么对SDL/GR输入的支持也是必需的。

图形编辑程序应具有的功能往往是：连接两个符号，把一组符号移到该页的另一区域或移到其它页上，以及链接删除（删除一个符号意味着删除与该符号的连接）。和SDL/PR工具一样，SDL/GR输入工具应当在SDL语义/语法上进行模拟，因此应当特别注意无效的连接，并提醒用户去填满所有不完全的部分等等。

这些工具面临着一些问题，这些问题是由图形设备的物理限制，例如“分辨能力”所引起的。几乎不可能要求这些设备既能读入一定数量的字符，同时又能在屏幕上显示一定数量的符号。

虽然已考虑了诸如图形变比窗口或上卷等问题的解决办法，但它们仍不能令人完全满意。如果用户只是复制图形，高分辨率就不是必须的。但若图形是由用户直接制作的，则高分辨率就是很需要的了。由于同样的原因，在图的显示中，既要求图的概貌又要求图的一定细节，因此也希望有高的分辨率。

辅助SDL/PR输入的工具是很有用的：它们可以用所期望的SDL/PR关键字提示用户。

它们能按照接收到的关键字立即安排SDL/PR的格式，自动地插入定界符，并向用户提供面向SDL/PR的功能键，提供一个清晰的布局等。

这类工具可以以现有的字符串编辑程序为基础来实现，这些编辑程序可以扩展，以包括上面提到的一些特性。

D.11.4 文件校验

文件一旦存入机器，下一步就是校验它们。首先应对它们分别进行校验，然后把有关联的图组合在一起再校验，直到整个系统校验完为止。

如果是采用面向SDL的工具进行输入的，则对每个单独文件的很多校验工作就随之完成了。

由“不可能的”操作（例如输入或保存所跟的符号不是状态）导致的错误，在输入阶段应全部被检测出来并进行校正。然而在一个独立的文件及各文件间存在矛盾的情况下，有些错误只有在输入阶段完成以后才能检测出来。

一些SDL规则可以自动得到检验，例如对于所有的输出应当有一个对应的输入这一规则。

在多层次规格的情况下，可以在一定程度上检验各层次之间的一致性。

形式的 SDL 模型可用来推导出一些校验过程。

D.11.5 文件复制

存储在计算机内的 SDL 文件必须能被检索、显示和复制。实现这些操作需要有执行所有这些功能的工具。能够只检索文件的一部分(即子集)是很有用的。检索可以是面向 SDL 的,例如“寻找所有发送一给定信号的进程”,或“在哪些状态”执行某一动作等等。当使用图形语法显示信息时,工具具有特殊的意义。在用 SDL/GR 语法输入文件时也是这样。文件的复制取决于待复制文件的类型,取决于该文件如何被存储及外部输出设备的特性。此外也可能取决于输入这些文件的方法。用户可能希望输入文件的语法有别于输出文件的语法。

文件复制受外部输出设备的限制,例如一个图可能太大,放不进一给定的页面积中,因而必须把图分割成若干块,必须加上连接符,并且应插入交叉引用的附注。人们希望把工具作出的“附加”内容与原来输入的特性区别开来,同时也希望有输出格式灵活的各种工具:其特性包括符号的各种不同尺寸、各种输出格式、垂直的或水平的显示等。

一个文件总可以按与其输入完全相同的方式再生。

D.11.6 文件生成

从用户输入的并贮存在计算机内的 SDL 文件出发,可以自动地生成若干其它文件,其中包括:

— 按每个进程,每个功能块或每个系统组织的信号表;
— 状态概览图,以一组代表跃迁的弧连接的状态来表示进程流图;
— 按每个进程、每个功能块或每个系统组织的交叉引用表;
— 功能块树图,指明各功能块和层次的结构;
— 系统行为,作为对环境动作序列的响应;
— 索引:对生成文件的再生,以上提供的考虑同样也是适用的。

以 SDL/GR 形式输入的 SDL 文件可自动地转换为等效的 SDL/PR 形式,反过来也是一样。

下面几点应予以考虑:

— SDL/GR 形式含有不能翻译成 SDL/PR 形式的直观信息(它在 SDL/PR 中不存在),例如,符号的座标在 SDL/PR 中是无意义的;
— 连接不同页上的各流线的连接符可以被删除。

然而,从 SDL/PR 到 SDL/GR 倒过来翻译则要复杂得多,并且不太可能完全满足所有的读者。

由于 SDL/GR 的二维表示,那些为了适应 SDL/PR 的顺序结构而插入的标号可以删去,用一连线就足够了。

这种翻译通常产生 SDL/GR 图的一个模型,该模型包含工具所必需的全部信息,用以在

某一图形设备上编辑和复制该图。

注意，两个不同的工具把某个 SDL/PR 翻译成 SDL/GR 时，可能会得出两种不同布局的 SDL/GR 规格。这样得到的两种 SDL/GR 规格都是正确的，只要它们保持原规格中的语义。

D.11.7 系统模拟和分析

SDL 文件不论它们是规定一个系统还是描述一个系统，基本上都是该系统的一个模型。

该模型主要用来把信息从一个人传递给另一个人，但也能用工具进行解释，检验其一致性、完备性（当规格的目的仅仅在于规定一个系统的某些部分时，不能满足完备性）和正确性，以及是否与 SDL 规则相符合（如文件检验一节所述）。

此外，可以研制一些工具，以便用模型来模拟各种系统功能的行为。模拟器可以与环境相互作用，并得出结论指明该模型与用户的期望相一致的程度。

如果加上附加信息，指明在执行每个动作时所耗费的时间，以及指明可利用的资源（队列、实例等）的大小，那么这种模拟也可用来研究系统的容量。

可以研制一些工具，从系统模型出发，来创建一个环境模型，并建立一个有意义的信号顺序，来检验实际系统。通过路径分析可以检测模型中的死锁。

系统模型也能用作联机文件编制。如果实际系统和文件存储器之间存在适当的链接，那么就可能研制一种工具，在模型上跟踪系统的实时事件。

为了实现这一点，应当提供从系统方面看到的物理事件与 SDL 文件所涉及的逻辑事件之间的对应关系。如果把文件组织成若干抽象层次，那么用户就可以选择要跟踪的层次。这一点是很有用的，因为这能使具有不同水平的用户观察到系统的活动。

解释 SDL 模型的工具也可用来挑选出同一系统的不同模型间的行巍—病☆异。它还可以用来比较各种不同的系统描述（由不同公司生产的系统），或者将系统规格与系统描述进行比较。这样就有可能了解一个系统描述是否符合原先的规格。

D.11.8 代码生成

提供了从形式上定义的语法和 SDL 的形式数学定义，有可能实现一些工具，用来把 SDL 规格的语义映射到编程语言的语义。这类工具也许不能提供完整的实现程序，但至少在为实际程序提供一个框架方面，它们会是很有用的。

本用户指南的 D.9.1节曾举了一个例子，概略地介绍了如何实现 SDL 和 CHILL 之间的映射。

D.11.9 训练

一套完整的关于 SDL 的训练教程已经编好。该教程包括了此语言的各个方面，同时还提供了一些例子以及关于使用 SDL 的几点建议。

SDL 教程可以从国际电信联盟总秘书处销售科得到。

通信地址：International Telecommunication Union，General Secretariat-Sales Section，Place des Nations，CH-1211 Geneve 20 (Switzerland)。

中国印刷 ISBN 92-61-03765-8