国 际 电 信 联 盟

ITU-T Z.100

(08/2002)

国际电信联盟
电信标准化部门

Z系列：电信系统使用的语言和一般性软件问题

正式描述技巧（FDT）— 规范和描述语言（SDL）

规范和描述语言（SDL）

ITU-T Z.100建议书

ITU-T Z系列建议书

电信系统使用的语言和一般性软件情况

正式描述技巧（FDT）	
规范和描述语言（SDL）	**Z.100-Z.109**
正式描述技巧的应用	Z.110-Z.119
信息排序表（MSC）	Z.120-Z.129
扩展的目标描述语言（eODL）	Z.130-Z.139
测试和测试控制记法（TTCN）	Z.140-Z.149
用户要求记法（URN）	Z.150-Z.159
编程语言	
CHILL: ITU-T 高级语言	Z.200-Z.209
人机语言	
总则	Z.300-Z.309
基本句法和对话程序	Z.310-Z.319
用于视频显示终端的扩展MML	Z.320-Z.329
人机接口规范	Z.330-Z.349
面向数据的人机接口	Z.350-Z.359
电信网络管理使用的人机接口	Z.360-Z.379
质量	
电信软件的质量	Z.400-Z.409
涉及协议的建议书中有关质量的内容	Z.450-Z.459
方法	
认证与测试的方法	Z.500-Z.519
中间件	
分布或处理环境	Z.600-Z.609

欲了解更详细信息，请查阅*ITU-T*建议书目录。

ITU-T Z.100建议书

规范和描述语言（SDL）

概　述

目标范围

本建议书定义SDL（**规范和描述语言**），旨在明确规范和描述电信系统。SDL的范围在第1节中详细叙述。本建议书是语言的一个参考手册。

覆盖范围

SDL包含有关行为、数据描述和（尤其对大型系统）构造方面的概念。行为描述基于扩展的有限状态机，它们通过消息进行通信。数据描述基于有关值和对象的数据类型。构造基于层次分解和类型层次。SDL的这些基础知识分别在本建议书的各主要章节中进行详细叙述。SDL的一个显著特征是图形化表示。

应用

SDL适用于标准实体和行业。计划中的SDL主要应用领域在第1.2节中描述，但SDL通常适于描述反应系统。应用范围从需求描述到实施。

状况/稳定性

本建议书是一份完整的语言参考手册，增补1中的使用指南为其提供支持。附件F给出了一个正式的SDL语义定义。本建议书的主体是稳定的，需要立即发布，以满足市场需要，但需要做进一步研究，以便完善附件F。附录I记录ITU-T Z.100建议书的状态，在完成进一步研究后应对其进行更新。虽然未来有望对语言做进一步扩展，但本建议书中定义的SDL-2000应在这几年内能满足大多数用户的需求。目前的版本基于广泛的SDL用户经验以及新近增加的用户需求。

建议书主体包括以下附件：

— 附件 A　非终结符索引
— 附件 B　留待将来使用（附件 B（03/93）不再有效）
— 附件 C　留待将来使用（附件 C（03/93）不再有效）
— 附件 D　SDL预定义数据
— 附件 E　留待例子使用
— 附件 F　SDL正式定义（单独发布）
— 附录 I　ITU-T Z.100建议书的状态、相关文档和建议书
— 附录 II　SDL维护指南
— 附录 III　SDL-92至SDL-2000的系统转化

ITU-T Z.100建议书还有一个单独发布的增补：

— Z.100增补1　SDL+方法：MSC和SDL（带ASN.1）的使用

相关工作

标准中SDL的一个使用方法在ITU-T Q.65建议书中描述。在ITU-T Z.110建议书中提供了一个建议的、有关引入正式描述技巧（如标准中的SDL）的策略。若想参考更多的有关SDL的资料以及有关SDL行业用法的信息，参见http://www.sdl-forum.org。

背景

1976年以来，CCITT和ITU-T推荐了不同版本的SDL。本版本是ITU-T Z.100（03/93）建议书的修订版，并收入了ITU-T Z.100（10/96）建议书的增补1以及ITU-T Z.105（03/95）建议书中的部分内容。本版本是对ITU-T Z.100（11/99）建议书所做的技术性更新，它收入了许多技术性的更正和补充，没有文本表达可选句法，它已移至ITU-T Z.106（2002）建议书中。

相比1992年定义的SDL，ITU-T Z.100（11/99）建议书中定义的版本扩展了面向对象数据的范围，溶入了许多特性，以使语言变得更加简单，并增强其他语言如ASN.1、ITU-T ODL（ITU-T Z.130建议书）、CORBA和UML对SDL的可用性。还包括了其他小的修改。虽然已注意不使现有的SDL文档无效，但一些修改可能需要对某些描述进行更新，以便使用本版本。在第1.5节中有对所做修改的详细描述。

来源

ITU-T第17研究组（2001-2004）按照WTSA第1号决议规定的程序，起草并于2002年8月6日批准了ITU-T Z.100建议书。

前　言

国际电信联盟（ITU）是从事电信领域工作的联合国专门机构。ITU-T（国际电信联盟电信标准化部门）是国际电信联盟的常设机构,负责研究技术、操作和资费问题，并且为在世界范围内实现电信标准化，发表有关上述研究项目的建议书。

每四年一届的世界电信标准化全会（WTSA）确定ITU-T各研究组的研究课题，再由各研究组制定有关这些课题的建议书。

WTSA第1号决议规定了批准建议书须遵循的程序。

属ITU-T研究范围的某些信息技术领域的必要标准，是与国际标准化组织（ISO）和国际电工技术委员会（IEC）合作制定的。

注

本建议书为简要而使用的“主管部门”一词，既指电信主管部门，又指经认可的运营机构。

知识产权

国际电联提请注意：本建议书的应用或实施可能涉及使用已申报的知识产权。国际电联对无论是其成员还是建议书制定程序之外的其他机构提出的有关已申报的知识产权的证据、有效性或适用性不表示意见。

至本建议书批准之日止，国际电联尚未收到实施本建议书可能需要的受专利保护的知识产权的通知。但需要提醒实施者注意的是，这可能不是最新信息，因此大力提倡他们查询电信标准化局（TSB）的专利数据库。

目 录

ITU-T Z.100建议书

规范和描述语言（SDL）

1 范围

推荐SDL（规范和描述语言）的目的是为了提供一种能确切规范和描述电信系统行为的语言。用SDL进行规范和描述的目的是使之显得正规，使得在分析和解释电信系统行为时能够做到明确无误。

规范和描述这两个术语具有下述意义：

a） 系统规范是对其要求的行为的描述；以及

b） 系统描述是对其实际的行为即它的实施情况的描述。

广义地说，系统规范既是系统行为的规范，也是系统一组一般参数的规范。不过，SDL旨在规范系统的行为；对于描述容量、重量等特性的一般参数需采用不同的方法来描述。

注 — 由于使用SDL进行规范和使用SDL进行描述之间没有什么区别，因此在本建议书中，术语“规范”既可用于要求的行为，也可用于实际的行为。

1.1 目标

当定义SDL时，总的目标是要提供这样一种语言：

a） 易于学习、使用和解释；

b） 为订购、投标和设计提供明确的规范，同时还允许开放某些事项；

c） 可以扩展，以覆盖新的发展；

d） 能够支持若干种系统规范和设计方法。

1.2 应用

本建议书是SDL的参考手册。给出了SDL用法的例子的方法框架文档，适用于1992-1996年研究期间生成的ITU-T Z.100建议书的增补1。首次于1993年3月发布的ITU-T Z.100建议书附录I还包含了方法指南，虽然这些指南未挖掘出SDL的所有潜能。

SDL的主要用途是用来规范实时系统的行为和设计。电信领域的应用包括：

a） 交换系统中的呼叫和连接处理（例如：呼叫处理、电话信令、测量）；

b） 一般电信系统中的维护和故障处理（例如：告警、自动故障排除、例行测试）；

c） 系统控制（例如：过载控制、更改和扩展程序）；

d） 操作和维护功能，网络管理；

e） 数据通信协议；

f） 电信业务。

当然，任何对象，只要其行为可以用离散模式来描述，也就是说，对象可用离散消息与其环境进行通信，那么功能上就可以用SDL来规范其行为。

SDL是一种丰富的语言，它不但可以用来描述高层非正式（和/或正式、非完全）规范，而且可以用来描述半正式和详细的规范。使用者需根据所要描述的通信层次以及语言的使用环境选择SDL的适当部分。根据规范的使用环境，在规范的源头与目标之间，可能有很多方面有待达成共识。

SDL可以用来产生：

a） 设备的要求；

b） 系统规范；

c） ITU-T建议书，或其他类似的标准（国际的、区域的或国家的）；

d） 系统设计规范；

e） 详细的规范；

f） 系统设计描述（高层的和详细的，足以直接促成实施）；

g） 系统测试描述（特别是联合MSC和TTCN）。

使用机构可以选择适当的SDL应用层次。

1.3 系统规范

假设系统的激励与响应均是离散的并携带信息，那么可以使用一个SDL规范，以激励/响应方式来定义系统的行为。在特殊情况下，系统规范可以看做是系统对任意给定激励序列的响应序列。

系统规范模型基于通信扩展有限状态机概念。

SDL还提供了构造概念，它有助于规范大型和/或复杂系统。这些构件允许将系统规范分解为易于管理的单元，使得可以独立处理和理解这些单元。分解过程可以分许多步来完成，这将导致单元的层次结构，以便在不同层次定义系统。

1.4 SDL-88与SDL-92之间的区别

在本建议书以往版本中定义的语言是ITU-T Z.100建议书（即1988年发布的蓝皮书）的扩展。在蓝皮书中定义的语言被称为SDL-88，在本建议书以往版本中定义的语言被称为SDL-92。在努力使SDL-92成为SDL-88的一个纯扩展，而不使语法失效或改变任何现有SDL-88用法的语义。另外，只接受基于需求的增强，需求应得到若干ITU-T成员实体的支持。

主要扩展在对象定向方面。SDL-88的基本模型是基于对象的，增加了一些语言构件，使得SDL-92能够更加全面、统一地支持对象范例：

a） 包；

b） 系统、功能块、进程和业务类型；

c） 系统、功能块、进程和基于类型的业务（业务集）实例；

d） 利用正文参数使类型参数化；

e） 类型特殊化以及虚拟类型重新定义与转换。

其他扩展是：自发转换、非确定性选择、用于与现有图表兼容的内部输入和输出符号、非确定性命令运算符**any** 、非延迟信道、远程程序呼叫和值返回程序、变量字段输入、运算符定义、与外部数据描述的结合、输出中的扩展寻址能力、转换中的自由行动、相同状态中以相同优先级进行的连续转换、信道的m:n连接以及在结构边界的信号路由。另外，还放宽了许多小的语法要求。

在一些情况下，对SDL-88所做的修改与SDL-88的定义是不一致的。引入的限制和修改可以通过一个自动的转换程序予以克服。转换SDL-92中的一个SDL-88文档也需要该程序，该文档包含由SDL-92关键字组成的名称。

对**output**构件，语义在SDL-88与SDL-92间进行了简化，这可能使SDL-88规范中**output**的一些特殊用法失效（当未提供**to**子句并存在若干种可能的信号路径时）。另外，还修改了类别等同属性的某些属性。

对**export/import**构件，引入了一个可选的远程变量定义，以便使变量出口与程序（远程程序）的引入出口相匹配。这需要对任何在入口表达式中包含限定词的或在相同范围中以不同类别引入若干入口名称的SDL-88文档进行修改。在某些情况下（很少），需要限定入口变量，以便通过正文解决决议问题。将SDL-88改为SDL-92引入了<远程变脸定义>，并用标识符限定了引入之远程变量的名称。

对**view**构件，视图定义已本地化为视图进程或业务。这需要修改在视图定义或视图表达式中包含限定符的SDL-88文档。将SDL-88改为SDL-92移去了这些限定符。这不改变视图表达式的语义，原因是，这些由其（未改变的）pid表达式决定。

service构件定义为一个原始概念，以代替简短形式，并不扩展其属性。业务的使用不受该改变影响，原因是，它本看起来就像是按一个原始概念来使用的。改变的理由是为了简化语言定义，以便将其与实际使用匹配起来，并减少业务的限制数量，它是由SDL-88中的规则转换而引起的。该修改的结果是，删去了业务信号路由构件；替代为可以使用信号路由了。这只是一个小的、概念上的改变，对具体使用没有影响（SDL-88业务信号路由的语法与SDL-92信号路由的语法相同）。

priority output构件从语言中移去了。该构件可以由带自动转换程序的**output to self**代替。

对基本SDL中的一些定义可以做很大扩展，如**signal**定义。应注意到，扩展是可选的，但旨在利用面向对象的扩展所引入的功能，如利用信号的参数化和特殊化功能。

以下是SDL-92的关键字而不是SDL-88的关键字：

any， **as**， **atleast**， **connection**， **endconnection**， **endoperator**， **endpackage**， **finalized**， **gate**，**interface**，**nodelay**，**noequality**，**none**，**package**，**redefined**，**remote**，**returns**，**this**，**use**，**virtual**。

1.5 SDL-92与SDL-2000之间的区别

做出的一个战略性的决定是在1992-1996年间保持SDL的稳定，使得在此期间结束时，只对SDL做了有限数量的修改。它们作为ITU-T Z.100建议书增补1（10/96）发布，而不更新SDL-92文档。虽然该版本的SDL有时被称为SDL-96，但相比从SDL-88到SDL-92的修改，这只是小的改动。改动包括：

a） 实现信号与远程程序和远程变量的协调；

b） 实现信号与信号路由的协调；

c） 增加外部程序和操作；

d） 允许功能块或进程作为系统使用；

e） 状态表达式；

f） 允许有关功能块和进程的包；

g） 无参数运算符。

这些现已纳入ITU-T Z.100建议书中，还有许多其他改动，生成了一个新的SDL版本，即SDL-2000。在本建议书中，ITU-T Z.100建议书（03/93）以ITU-T Z.100建议书增补1（10/96）定义的语言仍被称为SDL-92。2002年版的SDL-2000（名称未变）在ITU-T Z.100建议书（11/99）中增加了许多技术性改动，以便纠正错误或改善语言的描述，并做了一些小的扩展。该文档不再包括SDL-2000可选的文本语法，它现在在ITU-T Z.106建议书（08/2002）中定义。

更新SDL以支持和更好匹配其他语言（它们在与SDL的结合中经常用到）的好处开始超过语言稳定性（1992-1996年间保持稳定）的好处。另外，现代的工具和技术已使得能够更直接地从SDL规范中生成软件，不过通过在SDL使用中溶入更好的支持将获得更大的好处。SDL-2000很大程度上是对SDL-92的一个升级，不过应承认，它与SDL-92间存在某些不兼容性；否则生成的语言将太庞大、太复杂和太不一致。本节提供了有关改动的信息。关于大多数SDL-92描述如何能够系统地转换为SDL-2000将在附录 III中提供。

在众多方面进行了修改，焦点是简化语言，并调整新的应用领域：

a） 调整与其他语言的语法约定，这些语言使用SDL；

b） 协调基于“代理”的系统、功能块和进程概念，并将信号路由概念溶入信道概念；

c） 接口描述；

d） 异常处理；

e） 支持算法的文本符号；

f） 复合状态；

g） 以状态聚合构件代替业务构件；

h） 数据的新模型；

i） 支持以往在ITU-T Z.105建议书（03/95）中的SDL使用ASN.1的构件。

其他的变动还有：嵌套包、在功能块中直接包含功能块和进程、仅**out**的参数。

在语法层面，SDL-2000是大小写敏感的。提供了两种拼写形式的关键字：所有的大写形式或所有的小写形式。以下是SDL-2000的关键字而不是SDL-92的关键字：

abstract，aggregation，association，break，choice，composition，continue，endexceptionhandler，endmethod，endobject，endvalue，exception，exceptionhandler，handle，method，loop，object，onexception，ordered，private，protected，public，raise，value。

以下是SDL-92的关键字而不是SDL-2000的关键字：

all，axioms，constant，endgenerator，endnewtype，endrefinement，endservice，error，for，fpar，generator，imported，literal，map，newtype，noequal，ordering，refinement，returns，reveal，reverse，service，signalroute，view，viewed。

一小部分SDL-92的构件在SDL-2000中未提供：视图表达式、产生器、功能块分结构、信道分结构、信号细化、数据的公理定义、宏图表。这些构件很少用到（即使使用的话），将它们留在语言和工具中的管理费用太大，因此不值得将它们继续留在语言和工具中。

2 参考文献

下列ITU-T建议书和其他参考文献的条款，通过在本建议书中的引用而构成本建议书的条款。在出版时，所指出的版本是有效的。所有的建议书和其他参考文献都面临修订，使用本建议书的各方应探讨使用下列建议书和其他参考文献最新版本的可能性。当前有效的ITU-T建议书清单定期出版。本建议书中引用某个独立文件，并非确定该文件具备建议书的地位。

– ITU-T Recommendation T.50 （1992），*International Reference Alphabet （IRA） （Formerly International Alphabet No. 5 or IA5） – Information technology – 7-bit coded character set for information interchange.*

ISO/IEC 646:1991，*ISO 7-bit coded character set for information interchange.*

3 定义

在本建议书中定义了众多术语，本节中的定义清单将是建议书中许多文本的一个副本。因此，本节只给出了部分关键性的术语。

3.1 agent 代理：术语代理用于表示一个系统、功能块或进程，它包含一个或多个扩展的有限状态机。

3.2 block 功能块：功能块是一个代理，它包含一个或多个并行的功能块或进程，还可以包括一个扩展的有限状态机，它拥有并处理功能块中的数据。

3.3 body 实体：实体是代理、程序、复合状态或操作的状态机图。

3.4 channel 信道：信道是代理之间的通信路径。

3.5 environment 环境：系统环境指的是环境中的所有事物，它以类似SDL的方式与系统进行通信。

3.6 gate 通道：通道代表一个与代理类型的通信连接点，当类型实例化时，它决定代理实例与其他实例的连接。

3.7 instance 实例：实例是当类型实例化时创建的一个对象。

3.8 object 对象：术语对象用在关于值的数据项中。

3.9 pid：术语pid用在有关代理的数据项中。

3.10 procedure 程序：程序是代理部分行为的一个封装，它在代理中的某一处进行定义，但可以从代理中的若干处进行调用。其他代理可以调用一个远程程序。

3.11 process 进程：进程是一个代理，它包含一个扩展的有限状态机，并可以包含其他进程。

3.12 signal 信号：通信的主要方法是通过信号，它们通过发送代理输出，通过接收代理输入。

3.13 sort 类别：类别是数据项的一个集合，它们拥有共同的属性。

3.14 state 状态：当等待激励时，代理的一个扩展的有限状态机指的是一个状态。

3.15 stimulus 激励：激励指的是一个事件，它能产生一个代理，它处于进入转换的状态。

3.16 system 系统：系统指的是最外面的代理，它与环境进行通信。

3.17 timer 定时器：定时器是代理拥有的一个对象，它产生一个定时器信号激励，以便在某个特定时间发生。

3.18 transition 转换：转换指的是代理在进入一个状态前执行的一系列动作。

3.19 type 类型：类型是一个定义，它可用于创建实例，并可继承和特殊化，以形成其他类型。参数化的类型指的是一个拥有参数的类型。当为这些参数提供不同的真实参数时，定义不同的未参数化类型，即当实例化时，为实例提供不同的属性。

3.20 value 值：术语值用于数据类，它可直接访问。值可自由地在代理间传递。

4 缩写

本建议书采用下列缩写：

SDL-2000 本建议书定义的SDL

SDL-92 ITU-T Z.100建议书（03/93）增补1（10/96）定义的SDL

SDL-88 ITU-T Z.100建议书（1988）定义的SDL

5 约定

本节的文本不是标准的。相反，它定义了用于描述SDL的约定。在本节中，SDL的使用只是描述性的。引入元语言和约定的目的完全是为了清晰地描述SDL。

5.1 SDL语法

如果描述符合具体语法和抽象语法的要求，那么它只符合本建议书的要求：即描述必须被公认为SDL，并具有本建议书中所定义的相同含义。如果定义了进一步的具体语法，那么每个具体语法都拥有其自身语法的定义以及其自身与抽象语法之间关系的定义（即如何转换为抽象句法）。利用该方法，SDL的语义只有一种定义：每个具体语法通过其与抽象语法之间的关系继承语义。该方法还确保任何进一步的文法都是等同的。

提供的SDL正式定义用于定义如何将一个系统规范转换为抽象句法，并定义了如何根据抽象语法，解释一个规范。正式定义在附件F中给出（单独发布）。

对某些构件，没有直接等同的抽象句法。在这些情况下，为从具体句法向其他构件具体句法的转换提供了一个模型（直接地或间接地通过进一步的模型），模型具有一个抽象的语法。未与抽象句法（如注释）映射的项没有任何正式的含义。

5.2 基本定义

在本建议书中使用了一些一般的概念和约定，下面各子节给出它们的定义。

5.2.1 定义、类型和实例

在本建议书中，类型、实例及其他们之间的关系概念是最基本的概念。所采用的方案和术语定义如下，并示于图5-1中。

本子节介绍类型定义、实例定义、参数化的类型定义、参数化、文本参数的绑定、特殊化和实例化的基本语义。

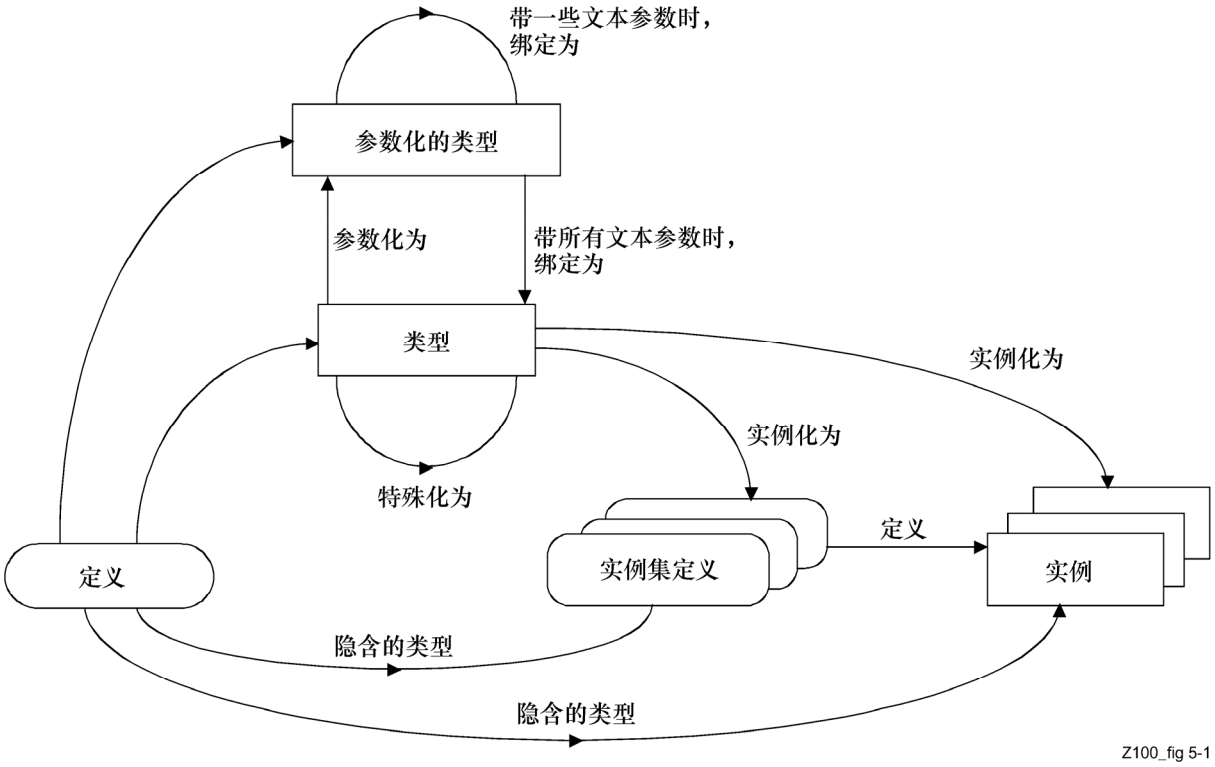


图5-1/Z.100－类型概念

定义引入了命名的实体，它们是类型或实例，具有隐含的类型或一个定义了行为实例的实例集。类型定义用于定义与该类型相关的所有属性。实例定义的一个例子是状态定义。类型定义的一个定义例子是信号定义。实例集定义的一个实例例子是进程定义。只有功能块或进程定义引入了实例集定义。

一个类型可以被实例化为任意多个实例。特定类型的一个实例拥有为该类型定义的所有属性。类型的一个例子是一个程序，它可以通过程序调用进行实例化。

参数化类型指的是这样一种类型，它的某些实体可代表为形式正文参数。类型定义的一个形式文本参数拥有一个约束。约束允许对参数化类型进行静态分析。绑定一个参数化类型的所有参数将获得一个普通类型。参数化类型的一个例子是一个参数化的信号定义，其中一个通过信号传送的类别可以通过形式类别正文参数进行特殊化；这使得在不同的正文中参数可以是不同的类别。

实例可以直接定义或通过类型的实例化进行定义。实例的一个例子是系统实例，它可以通过一个系统定义来定义，或是系统类型的一个实例化。

通过增加父类型的属性，或重新定义父类型的虚拟属性，特殊化使得一个类型、子类型可以基于另一个类型、其父类型。可以对一个虚拟属性进行限制，以便用于一般类型的分析。

绑定一个参数化类型的所有正文参数将获得一个非参数化的类型。在参数化类型与自其衍生的类型之间不存在任何父类型/子类型关系。

注 — 为避免繁琐的文本，约定中“实例”一词可以被省略。例如“系统可解释为……”指的是“系统实例可解释为……”。

5.2.2 环境

用SDL规定的系统根据与外界交流的激励来行动，这个外界被称为被规定系统的环境。

假设在环境中有一个或多个代理实例，因而从环境流向系统的激励具有与这些代理实例相关的标识，这些代理所具有的pid值不同于系统中的任何pid值（见第12.1.6节）。

尽管环境的行为是不确定的，但它必须遵循由系统规范所给定的限制。

5.2.3 有效性和错误

仅当一个系统规范满足SDL的句法规则和静态条件时，该系统规范才是一个合法的SDL系统规范。

如果一个合法的SDL规范在解释时违反了动态条件，那么会出现错误。在系统解释期间，若遇到一个错误，那么将出现预定义的异常（见D.3.16）。如果不对异常进行处理，那么系统的后续行为不能自规范生成。

5.3 表达格式

下列表达格式用于分隔每个主题下的不同语言要点。

5.3.1 文本划分

本建议书按主题进行组织，通过一段可选的引言进行描述，接着是有关以下几方面的、带有标题的列举项：

a) *抽象语法* — 通过抽象句法和静态条件进行描述，以便呈现良好形态。

b) *具体语法* — 通过图形化的语法、静态条件和具有良好形态的规则进行描述，以便呈现图形化的语法、该语法与抽象句法之间的关系，以及其他一些绘图规则（指的是第6.5节中的那部分内容）。

c) *语义* — 给出一个构件的含义、给出其所具有的属性、解释它的方法以及该构件为了以SDL方式良好运作而需满足的任何动态条件。

d) *模型* — 给出符号映射，这些符号没有直接的抽象句法，并依据其他具体句法构件进行建模。通过其他构件进行建模的符号即缩写，它被认为是衍生的语法，用于转换形式。

5.3.2 带有标题的列举项

当一个主题具有引言部分并随之有一带有标题的列举项时，则引言部分被认为是本建议书的非正式部分，该部分仅用于帮助理解，而不是为了使本建议书完整。

若带有标题的列举项没有正文，则该项整个被略去。

本节的剩余部分描述用于每一个带标题的列举项及所用标题的其他特殊形式。它也可以被认为是上面所定义的第一级带标题的列举项的一个印刷版面的例子，其中的正文是引言节的一部分。

a) *抽象语法*

在第5.4.1节中给出了抽象句法符号的定义。

如果省略了带标题的列举项“抽象语法”，那么对所介绍的主题来说，没有额外的抽象句法，并且具体句法将映射至由另一个带序号的正文节所定义的抽象句法。

可以从任何带标题的列举项引用抽象句法中的规则，办法是用斜体字来书写规则的名称。

正式符号中的规则后可以跟有数段文字，用于定义具有良好形态的SDL定义必须满足的条件，它们可以无需某个实例的解释就能进行校验。这里的静态条件指的只是抽象句法。仅与具体句法有关的静态条件以具体句法进行定义。抽象句法的静态条件与抽象句法一起定义了语言的抽象语法。

b) 具体语法

具体句法用语法描述中的扩展型巴科斯—瑙尔（Backus-Naur）范式进行说明，它在第5.4.2节中定义。

具体句法后跟有数段文字，用于定义在具有良好形态的正文中必须满足的静态条件，它们可以无需某个实例的解释就能进行校验。静态条件（如果有的话）对抽象语法也是适用的。

在很多情况下，具体句法与抽象句法之间有一种简单的关系，原因是，具体句法规则可以简单地用抽象句法中的一条单一规则来表示。当在抽象句法和具体句法中使用相同名字以便表明它们代表同样概念时，那么在语言描述中就意味着含有这样的文本内容："具体句法中的〈x〉代表抽象句法中的X"，而这通常是被省略掉的。在这个意义上，大小写将被忽略，而带下划线的语义子类（见第5.4.2节）将具有重要意义。

非缩写形式的具体句法是严格的具体正文语法。仅对严格的具体句法定义了从具体句法至抽象句法的关系。

如果所定义的主题是由其他SDL构件（见下面的模型一段）建模的缩写形式，那么具体句法与抽象句法之间的关系可以被省略。

当在具体语法中一个非终结符的名称是以"diagram"字结束并且在抽象语法中存在一个名称其仅有的不同是以"definition"字结束，那么这两个规则代表的是相同的概念。例如，在具体语法中的<system diagram>和在相应抽象语法中的<system-definition>。

当在具体语法中一个非终结符的名称是以"area"字结束并且在抽象语法中存在一个名称其仅有的不同是删去了"area"字，那么这两个规则代表的是相同的概念。例如，在具体语法中的<state partition area>和在相应抽象语法中的<state-partition>。

c) 语义

属性指的是SDL中不同概念之间的关系。属性用在具有良好形态的规则中。

例如，一个进程合法输入信号标识符的集合是一种属性。该属性用在下述静态条件中："对每一个State-node，所有Signal-identifier（在合法输入信号集内）或出现在一个Save-signalset中，或出现在一个Input-node中"。

除以某种特殊方法形成的外，所有实例都具有标识属性，此标识属性的确定如第6.3节中关于标识的一般规则一节中所定义的那样。作为一种标识属性，它通常是不提及的。也不必提及包含在定义中的子定义部分，原因是，这种子定义部分的所有权从抽象语法的角度来看，是显而易见的。例如，功能块定义显然"已经"包含了进程和/或功能块。

如果属性可以在无需解释SDL系统规范的情况下确定，那么属性是静止的；如果为了确定属性需要解释SDL系统规范，那么属性是动态的。

解释可以用运行的方式来描述。每当在抽象句法中存在一个列表时，那么对此列表的解释要按给定的顺序进行。这就是说，本建议书描述了怎样从系统定义创建实例，以及在"抽象的SDL机"中怎样解释这些实例。列表在抽象句法中用后缀"*"和"＋"表示（见第5.4.1节）。

动态条件是这样一种条件，它在解释期间必须得到满足，并且若不进行解释则不能进行校验。动态条件可能导致错误（见第5.2.3节）。

注 — 系统行为通过"解释"SDL产生。"解释"一字需显性选择（而不是可选的，如"执行"），以便包括通过人工进行的SDL解释和通过计算机进行的SDL解释。

d) 模型

某些构件被认为是"衍生的具体句法"（或缩写符号），用于其他等同的具体句法构件。例如，删除一个信号的输入是该信号输入的衍生具体句法，后跟一个回到同一状态的空转换。

缩写符号的属性源自其依据原始概念（或转变为原始概念）进行建模的方式。为了确保缩写符号使用的简便、明确，并减少当有若干个缩写符号相结合时产生的副作用，这些概念将按附件F中所定义的特定顺序进行转变。

衍生具体句法片段转变的结果通常或者是衍生具体句法的另一个片段，或者是具体句法的一个片段。转变的结果还可以是空的。对后一种情况，原件将从规范中删去。

转变可以是相互依赖的，因此各种不同转变所用的顺序决定了一个SDL规范的合法性和含义。有关转变顺序的详细内容参见附件F。

5.4 元语言

对属性定义和SDL语法，可以根据特定需求使用不同的元语言。

本建议书中给出的语法用于帮助在本建议书中的表述，以便上下文中给出的规则名称含义明确，能在文本中使用。这意味着可以通过系统地重写语法规则或语义规则应用，来方便地解决诸多明显的含糊之处。

下面将介绍所用的元语言。

5.4.1 抽象语法的元语言

下面描述SDL的抽象句法。

抽象句法中的一个定义可以看做是一个命名的复合对象（一棵树），用来定义一组子部件。

例如，关于信道定义的抽象句法为：

Channel-path :: *Originating-gate*
Destination-gate
*Signal-identifier-**set***

它为名为*Channel-path*的复合对象（树）定义了域。此对象由3个子部件构成，这3个子部件又可以是树。

定义

Agent-identifier = *Identifier*

表示*Agent-identifier*是一个*Identifier*，因此，从语法上不能将它与其他标识符区分开来。

一个对象也可能是某种基本的（非复合的）域。在SDL上下文中，它们是：

a） 自然对象

例如：

Number-of-instances :: *Nat* [*Nat*]

*Number-of-instances*表征一个复合域，它包含一个强制性的自然数（*Nat*）值和一个可选的自然数（[*Nat*]）值，分别表示实例的初始数与可选的最大数。

b） 引用对象

黑体大写字母和数字的任意序列表示引用。

例如：

Channel-definition :: *Channel-name*
[NODELAY]
Channel-path-set

信道不可以是延迟的。这可以通过一个可选的引用**NODELAY**来表示。

c） 标记对象

*Token*表示标记的域。可以认为该域包含一个潜在的无穷集合，它由各不相同的基本对象组成。这些基本对象不需要描述。

例如：

Name :: *Token*

一个名字由一个基本对象构成，以便使任意*Name*都能够与任何其他名字相区分开来。

d） 未定对象

未定对象指的是可能有某种描述的域，但在本建议书中，不关注此类描述。

例如：

Informal-text :: ...

*Informal-text*包括一个没有解释的对象。

BNF中的以下运算符（构件符）（见第5.4.2节）与抽象句法中的用法相同，“*”表示一个可能为空的列表，“+”表示一个非空列表，“|”表示二选一，“[”“]”表示可选项。

圆括号用来分组逻辑上相关的域。

最后，抽象句法使用另一个后缀运算符“**-set**”来产生一个集合（各不相同对象的无序集合）。

例如：

Agent-graph :: *Agent-start-node State-node-**set***

一个*Agent-graph*由一个*Agent-start-node*和一个*State-nodes*集合构成。

5.4.2 具体语法的元语言

在有关词汇规则的巴科斯—瑙尔范式（BNF）中，终结符为<space>，打印字符见第6.1节。

在有关非词汇规则的巴科斯—瑙尔范式（BNF）中，终结符为第6.1节中所定义的词汇单元之一（<name>，<quoted operation name>，<character string>，<hex string>，<bit string>，<special>，<composite special>或 <keyword>=。在非词汇规则中，终结符可以用下列之一来表示：

a） 关键字（如state）；

b） 有关词汇单元的字符，如果它由一个单个字符组成（如“=”）；

c） 词汇单元名称（如<quoted operation name>或<bit string>）；

d） <composite special>词汇单元名称（如<implies sign>）。

为避免与巴科斯—瑙尔范式（BNF）文法混淆，总使用词汇单元名称<asterisk>、<plus sign>、<vertical line>、<left square bracket>、<right square bracket>、<left curly bracket> 以及 <right curly bracket>，而不使用相当的字符。注意，两个特殊的终结符<name>和<character string>还可以具有语义，如以下定义所强调的那样。

尖括号及其所括起来的词或者是非终结符，或者是词汇单元之一。由括在尖括号中的一个或多个词表示的非终结符属语法范畴。在具体语法中，为每一个非终结符都给出了一种产生规则。例如：

<block reference> ::=
block <block name> referenced <end>

非终结符的产生规则由符号“::=”左边的非终结符和右边的一个或多个构件组成，构件由右边的非终结符和/或终结符组成。例如，上面例子中的<block reference>、<block name>和<end>都是非终结符；**block** 和 **referenced**都是终结符。

有时符号含有带下划线的部分，此带下划线部分用于强调那个符号的语义。例如，<block name>在语法上是与<name>是相同的，但在语义上，它要求名称为一个功能块的名称。

在符号“::=”右边，可以给出几种供选择的终结符产生方式，它们由一竖杆（“|”）进行分隔。例如：

<diagram in package> ::=
<package diagram>
| <package reference area>
| <entity in agent diagram>
| <data type reference area>
| <signal reference area>
| <procedure reference area>
| <interface reference area>
| <create line area>
| <option area>

表示<diagram in package>是 <package diagram>或<package reference area>或<entity in agent diagram>或<data type reference area>或<signal reference area>或<procedure reference area>或<interface reference area>或<create line area>或<option area>。

可以用花括号（“{”和“}”）把语法元素组合起来，此处的花括号类似于Meta Ⅳ中（见第5.4.1节）的圆括号。一对花括号中可以含有一个或多个竖杠，用来表示可选的语法元素。例如，

<operation definitions> ::=

{ <operation definition>
| <operation reference>
| <external operation definition> }+

语法元素或花括号组合的重复由星号（“*”）或加号（“+”）表明。星号表示此花括号组合是可选的，并可进一步重复任意多次。加号表示该花括号组合必须出现，并可进一步重复任意多次。上面的例子表示<operation definitions>可以包含零个或多个<operation definition>、<operation reference> 或 <external operation definition>定义，并可以包含它们中任意一个的多个定义。

如果语义元素是用方括号（“[”和“]”）分组的，那么组是可选的。例如：

<valid input signal set> ::=

signalset [<signal list>] <end>

表示<valid input signal set> 可以但不是必须包含 <signal list>。

为了支持图形化语法，元语言拥有下列元符号：

a） ***set***

b） ***contains***

c） ***is associated with***

d） ***is followed by***

e） ***is connected to***

f） ***is attached to***

元符号set是一个后缀运算符，它作用于紧靠其前的、花括号内的语法元素，表示的是一个（无序的）项集合。每一个项都可以是任意一组语法元素，在此情况下，它必须在应用set元符号之前进行扩展。

例如：

{ <operation text area>* <operation body area> } ***set***

是一组包含零个或多个<operation text area>和一个<operation body area>的集合。当图中语法元素相互间位置无关且元素可以以任何次序进行考虑时，使用元符号***set***。

所有其他元符号均为中缀运算符，它们有一个图形非终结符作为左边的变元。右边变元或为位于花括号中的一组语法元素或为单一语法元素。如果一个产生规则的右边有一图形非终结符作为第一个元素，并含有这些中缀运算符中的一个或多个，那么图形非终结符是每一个这种中缀运算符的左边变元。一个图形非终结符是一个非终结符，它以“symbol”字结束。

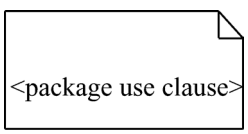
元符号***contains***表示，如果有的话，其右边的变元应放在其左边变元与附加的<text extension symbol>之间。右边变元在符号内进行扩展，不得跨越符号边界，并区别于任何在另一规则中的具有相同语法的事件。例如，

<package use area> ::=

<text symbol> ***contains*** <package use clause>

<text symbol> ::=

指的是以下意思：



元符号***is associated with***表示其右边变元在逻辑上与其左边变元关联（仿佛它“包含”在那个变元之中，其明确的关联关系由适当的绘图规则来保证）。对右边变元进行扩充，并区别于任何在另一规则中具有相同语法的事件。

元符号***is followed by***的意思是：其右边变元（在逻辑上和画法上）跟随在其左边变元后面，并表示一个流程线符号（见第6.5节）。在流程线符号末尾对右边变元进行扩充，并区别于任何在另一规则中具有相同语法的事件。

元符号***is connected to***的意思是：其右边变元（在逻辑上和画法上）连接到其左边的变元。对右边变元进行扩充，并区别于任何在另一规则中具有相同语法的事件（相比以下***is attached to***）。

元符号 ***is attached to*** 表示语法要求而非语法产生。元符号***is attached to*** 要求其右边变元和左边变元互相隶属（逻辑上和绘图上），但一个变元不以另一个变元的语法进行扩充，不过每一个变元应作为语法规则（相比上面的***is connected to***）的独立扩充存在。由于相互隶属，因此A ***is attached to*** B 在语法上总与另一规则B ***is attached to*** A匹配，虽然这无需在B上直接表示出来。例如，B可以具有可选项B1和B2，每个***is attached to*** A。隶属通常意味着每边的抽象句法都包含另一边的标识符。

6 一般规则

6.1 词汇规则

词汇规则定义词汇单元。词汇单元是**具体语法**的终结符。

```
<lexical unit> ::=
                        <name>
                    |   <quoted operation name>
                    |   <character string>
                    |   <hex string>
                    |   <bit string>
                    |   <note>
                    |   <comment body>
                    |   <composite special>
                    |   <special>
                    |   <keyword>
<name> ::=
                        <underline>* <word> {<underline>+ <word>}* <underline>*
                    |   {<decimal digit>}+ { {<full stop>} <decimal digit>+ }*
<word> ::=
                        {<alphanumeric>}+
<alphanumeric> ::=
                        <letter>
                    |   <decimal digit>
<letter> ::=
                        <uppercase letter> | <lowercase letter>
<uppercase letter> ::=
                        A   | B   | C   | D   | E   | F   | G   | H   | I   | J   | K   | L   | M
                    |   N   | O   | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   | Z
<lowercase letter> ::=
                        a   | b   | c   | d   | e   | f   | g   | h   | i   | j   | k   | l   | m
                    |   n   | o   | p   | q   | r   | s   | t   | u   | v   | w   | x   | y   | z
<decimal digit> ::=
                        0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9
<quoted operation name> ::=
                        <quotation mark> <infix operation name> <quotation mark>
                    |   <quotation mark> <monadic operation name> <quotation mark>
```

<infix operation name> ::=
or | **xor** | **and** | **in** | **mod** | **rem**
| <plus sign> | <hyphen>
| <asterisk> | <solidus>
| <equals sign> | <not equals sign>
| <greater than sign> | <less than sign>
| <less than or equals sign> | <greater than or equals sign>
| <concatenation sign> | <implies sign>

<monadic operation name> ::=
<hyphen> | **not**

<character string> ::=
<apostrophe> { <general text character>
| <special>
| <apostrophe> <apostrophe>
}* <apostrophe>

<apostrophe> <apostrophe> represents an <apostrophe> within a <character string>.

<hex string> ::=
<apostrophe> { <decimal digit>
| a | b | c | d | e | f
| A | B | C | D | E | F
}* <apostrophe> { H | h }

<bit string> ::=
<apostrophe> { 0 | 1
}* <apostrophe> { B | b }

<note> ::=
<solidus> <asterisk> <note text> <asterisk>+ <solidus>

<note text> ::=
{ <general text character>
| <other special>
| <number sign>
| <asterisk>+ <not asterisk or solidus>
| <solidus>
| <apostrophe> }*

<not asterisk or solidus> ::=
<general text character> | <other special> | <apostrophe> | <number sign>

<text> ::=
{ <general text character> | <special> | <apostrophe> }*

<general text character> ::=
<alphanumeric> | <other character> | <space>

<comment body> ::=
<solidus> <number sign> <note text> <number sign>+ <solidus>

<comment text> ::=
{ <general text character>
| <other special>
| <asterisk>
| <number sign>+ <not number or solidus>
| <solidus>
| <apostrophe> }*

<not number or solidus> ::=
<general text character> | <other special> | <apostrophe> | <asterisk>

<composite special> ::=
<result sign>
| <range sign>
| <composite begin sign>
| <composite end sign>
| <concatenation sign>
| <history dash sign>
| <greater than or equals sign>
| <implies sign>
| <is assigned sign>
| <less than or equals sign>
| <not equals sign>
| <qualifier begin sign>
| <qualifier end sign>

<result sign> ::=
<hyphen> <greater than sign>

<range sign> ::=
<full stop> <full stop>

<composite begin sign> ::=
<left parenthesis> <full stop>

<composite end sign> ::=
<full stop> <right parenthesis>

<concatenation sign> ::=
<solidus> <solidus>

<history dash sign> ::=
<hyphen> <asterisk>

<greater than or equals sign> ::=
<greater than sign> <equals sign>

<implies sign> ::=
<equals sign><greater than sign>

<is assigned sign> ::=
<colon> <equals sign>

<less than or equals sign> ::=
<less than sign> <equals sign>

<not equals sign> ::=
<solidus> <equals sign>

<qualifier begin sign> ::=
<less than sign> <less than sign>

<qualifier end sign> ::=
<greater than sign> <greater than sign>

<special> ::=
<solidus> | <asterisk> | <number sign> | <other special>

<other special> ::=
<exclamation mark>
| <left parenthesis> | <right parenthesis>
| <plus sign> | <comma> | <hyphen>
| <full stop> | <colon> | <semicolon>
| <less than sign> | <equals sign> | <greater than sign>
| <left square bracket> | <right square bracket>
| <left curly bracket> | <right curly bracket>

<other character> ::=
<quotation mark> | <dollar sign> | <percent sign>
| <ampersand> | <question mark> | <commercial at>
| <reverse solidus> | <circumflex accent> | <underline>
| <grave accent> | <vertical line> | <tilde>

<exclamation mark> ::= **!**
<quotation mark> ::= **"**
<left parenthesis> ::= **(**
<right parenthesis> ::= **)**
<asterisk> ::= *****
<plus sign> ::= **+**
<comma> ::= **,**
<hyphen> ::= **-**
<full stop> ::= **.**
<solidus> ::= **/**
<colon> ::= **:**
<semicolon> ::= **;**
<less than sign> ::= **<**
<equals sign> ::= **=**
<greater than sign> ::= **>**
<left square bracket> := **[**
<right square bracket> ::= **]**
<left curly bracket> ::= **{**
<right curly bracket> ::= **}**
<number sign> ::= **#**
<dollar sign> ::= **$**
<percent sign> ::= **%**
<ampersand> ::= **&**
<apostrophe> ::= **'**
<question mark> ::= **?**
<commercial at> ::= **@**
<reverse solidus> ::= ****
<circumflex accent> ::= **^**
<underline> ::= **_**
<grave accent> ::= **`**
<vertical line> ::= **|**
<tilde> ::= **~**
<keyword> ::=

abstract | **active** | **adding**
| **aggregation** | **alternative** | **and**
| **any** | **as** | **association**
| **atleast** | **block** | **break**
| **call** | **channel** | **choice**
| **comment** | **composition** | **connect**
| **connection** | **constants** | **continue**
| **create** | **dcl** | **decision**
| **default** | **else** | **endalternative**
| **endblock** | **endchannel** | **endconnection**
| **enddecision** | **endexceptionhandler** | **endinterface**
| **endmacro** | **endmethod** | **endobject**
| **endoperator** | **endpackage** | **endprocedure**
| **endprocess** | **endselect** | **endstate**
| **endsubstructure** | **endsyntype** | **endsystem**
| **endtype** | **endvalue** | **env**
| **exception** | **exceptionhandler** | **export**
| **exported** | **external** | **fi**
| **finalized** | **from** | **gate**
| **handle** | **if** | **import**
| **in** | **inherits** | **input**
| **interface** | **join** | **literals**
| **loop** | **macro** | **macrodefinition**
| **macroid** | **method** | **methods**
| **mod** | **nameclass** | **nextstate**
| **nodelay** | **none** | **not**
| **now** | **object** | **offspring**
| **onexception** | **operator** | **operators**
| **optional** | **or** | **ordered**
| **out** | **output** | **package**
| **parent** | **priority** | **private**
| **procedure** | **protected** | **process**
| **provided** | **public** | **raise**
| **redefined** | **referenced** | **rem**
| **remote** | **reset** | **return**
| **save** | **select** | **self**
| **sender** | **set** | **signal**
| **signallist** | **signalset** | **size**
| **spelling** | **start** | **state**
| **stop** | **struct** | **substructure**
| **synonym** | **syntype** | **system**
| **task** | **then** | **this**
| **timer** | **to** | **try**
| **type** | **use** | **value**
| **via** | **virtual** | **with**
| **xor**

<space> ::=

<lexical unit>和<note>中的字符以及字符<space>和控制字符由国际参考字母表国际参考版（ITU-T T.50建议书）定义。词汇单元<space>代表T.50 SPACE字符（只取首字母的缩写为SP），它不能显示（因为很显然的原因）。

<text>在<comment area>中使用，它相当于<character string>，在<text extension area>中，它必须作为其他词汇单元的一个序列来对待。

完全忽略T.50删除字符。如果使用一个扩展的字符集，那么不由T.50定义的字符只能出现在<comment area>中的<text>中，或<comment>中的<character string>中，或<note>中。

当一个<underline>字符后跟一个或多个<space>字符或控制字符时，将忽略所有这些字符（包括<underlin>），例如A_ B表示的<name>等同于AB。这样使用<underline>允许将<lexical unit>分成多行。该规则在任何其他词汇规则之前使用。

在<space>可以出现的地方，可以出现（非space）控制字符，其含义同<space>。

在<informal text>和<note>中出现控制字符影响不大。为了构造一个包含控制字符的字符串表达式，<concatenation sign>运算符和控制字符文字必须使用。字符串中的所有空格是有意义的：一串空格不得看待为一个空格。

在任何<lexical unit>之前或之后都可以插入任何数量的<space>。插入的<space>或 <note>在语法上没有相关性，但有时需要使用一个<space>或 <note>来将两个<lexical unit>分开。

除了关键字，在所有的<lexical unit>中，大写<letter>和小写<letter>是有区别的。因此，AB、aB、Ab和ab代表的是四个不同的<word>。一个所有字母都大写的<keyword>与一个所有字母都小写的<keyword>具有相同的拼法（即忽略大小写），但在一个混合有大小写的字母串且其拼法与某个<keyword>相同时，代表的是一个<word>。

为使词汇规则和具体语法简明起见，小写的、作为终结符的<keyword>与大写的<keyword>具有相同拼法，并可在同一地方使用。例如，关键字

default

代表词汇可选项：

{ default | DEFAULT }

注 — 黑体小写用于本建议书中的关键字。通过字体属性来区分不是强制性的要求，但对规范的读者来说是有用的。

通过首字符来终结一个<lexical unit>，根据上面规定的语法，它不能是<lexical unit>的一部分。如果一个<lexical unit>既可以是一个<name>，也可以是一个<keyword>，那么它是一个<keyword>。如果两个<quoted operation name>只是大小写不同，那么使用小写名称的语义，因此（例如）表达式"OR"(a，b)的含义与表达式(a **or** b)的含义相同。

注 — SDL的某些关键字只能在ITU-T Z.106建议书中使用。

特殊的词汇规则在<macro body>中使用。

6.2 宏

一个宏定义包含词汇单元的一个集，它可以被包括在一个<sdl specification>具体语法文本部分中的一个或多个位置上。每个这样的位置都通过一个宏调用来指明。在可以对<sdl specification>进行分析之前，每个宏调用都必须用相应的宏定义进行替换。

6.2.1 额外的词汇规则

<formal name> ::=
[<name>**%**] <macro parameter>
{ [**%**<name>] **%**<macro parameter> }*
[**%**<name>=

6.2.2 宏定义

<macro definition> ::=
macrodefinition <macro name>
[<macro formal parameters>= <end>
<macro body>
endmacro [<macro name>] <end>

<macro formal parameters> ::=
(<macro formal parameter> { **，** <macro formal parameter>}* **)**

<macro formal parameter> ::=
<name>

<macro body> ::=
{<lexical unit> | <formal name>}*

<macro parameter> ::=
<macro formal parameter>
| **macroid**

<macro formal parameter>必须是有区别的。一个宏调用的<macro actual parameter>必须与其相应的<macro formal parameter>一对一匹配。

<macro body>不得包含关键字**endmacro**和**macrodefinition**。

<macro definition>包含词汇单元。

<macro name>在整个系统定义中是可见的，不论宏定义在哪里出现。一个宏调用可以在相应的宏定义之前出现。

宏定义可以包含宏调用，但宏定义不得在其他宏定义中通过宏调用来调用它自身，不论是通过直接调用，还是通过间接调用。

在每个宏定义中，关键字**macroid**都可以用做伪宏形式参数。没有<macro actual parameter>可以提供给它，通过一个惟一的<name>来替换它，来实现对一个宏定义的每个扩充（在一个扩充中，对**macroid**的每个事件使用相同的<name>=。

例子

下面是<macro definition>的一个例子：

```
macrodefinition Exam (alfa, c, s, a);
          block alfa referenced;
          dcl exported c as s Integer := a;
endmacro Exam;
```

6.2.3 宏调用

<macro call> ::=

macro <macro name> [<macro call body>] <end>

<macro call body> ::=

(<macro actual parameter> {**,** <macro actual parameter>}* **)**

<macro actual parameter> ::=

<lexical unit>*

<lexical unit>不能是逗号“，”或右括号“）”。如果在<macro actual parameter>中需要用到这两个字符中的任意一个，那么<macro actual parameter>必须是<character string>。如果<macro actual parameter>是<character string>，那么当<macro actual parameter>替换<macro formal parameter>时，使用<character string>的结果。

<macro call>可以出现在任何允许<lexical unit>的地方。

模型

<sdl specification>可以包含宏定义和宏调用。在可以对这样一个<sdl specification>进行分析之前，必须对所有的宏调用进行扩充。扩充宏调用指的是对具有相同<macro name>（在宏调用中给出）的一个宏定义拷贝进行扩充，以替换该宏调用。这意味着创建一个宏实体的拷贝，并且该拷贝的<macro formal parameter>的每一个事件将由宏调用相应的<macro actual parameter>进行代替，然后，如果有的话，对拷贝中的各宏调用进行扩充。当<macro formal parameter>由<macro actual parameter>代替时，移去<formal name>中的所有百分比字符（%）。

在<macro formal parameter>与<macro actual parameter>之间，必须存在一对一的通信联系。

例子

下面是在<block diagram>片断中的<macro call>的一个例子。

```
.........
block A referenced;
macro Exam (B, C1, S1, 12);
.........
```

利用第6.2.2节中的例子对此宏调用进行扩充，得到以下结果。

```
.........
block A referenced;
block B referenced;
dcl exported C1 as S1 Integer := 12;
.........
```

6.3 视见度规则、名称和标识符

抽象语法

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item +*
Path-item	=	*Package-qualifier*

| *Agent-type-qualifier*
| *Agent-qualifier*
| *State-type-qualifier*
| *State-qualifier*
| *Data-type-qualifier*
| *Procedure-qualifier*
| *Signal-qualifier*
| *Interface-qualifier*

Package-qualifier :: *Package-name*
Agent-type-qualifier :: *Agent-type-name*
Agent-qualifier :: *Agent-name*
State-type-qualifier :: *State-type-name*
State-qualifier :: *State-name*
Data-type-qualifier :: *Data-type-name*
Procedure-qualifier :: *Procedure-name*
Signal-qualifier :: *Signal-name*
Interface-qualifier :: *Interface-name*
Package-name = *Name*
Agent-type-name = *Name*
Agent-name = *Name*
State-type-name = *Name*
Data-type-name = *Name*
Interface-name = *Name*
Name :: *Token*

具体语法

<identifier> ::=
[<qualifier>] <name>

<qualifier> ::=
<qualifier begin sign> <path item> { / <path item> }* <qualifier end sign>

<string name> ::=
<character string>
| <bit string>
| <hex string>

<path item> ::=
[<scope unit kind>] <name>

<scope unit kind> ::=
package
| **system type**
| **system**
| **block**
| **block type**
| **process**
| **process type**
| **state**
| **state type**
| **procedure**
| **signal**
| **type**
| **operator**
| **method**
| **interface**

范围单元通过下列具体语法的非终结符进行定义。一些范围单元种类既具有文本形式的定义，又具有图形形式的定义。它们显示在同一条线上，文本定义显示在左边栏。

	＜package diagram＞
	＜agent diagram＞
	＜agent type diagram＞
＜procedure definition＞	＜procedure diagram＞
＜data type definition＞	
＜interface definition＞	
＜operation definition＞	＜operation diagram＞
	＜composite state area＞
	＜composite state type diagram＞
＜sort context parameter＞	
＜signal definition＞	
＜signal context parameter＞	
＜compound statement＞	＜task area＞

一个范围单元附有一个定义列表，每一个定义用于定义一个或多个术语某个实体种类的实体，并具有一个相关的名称，包括包含在范围单元中的通道定义、＜formal context parameter＞、＜agent formal parameters＞和＜formal variable parameters＞。

尽管＜quoted operation name＞和＜string name＞有它们自己的语法符号，实际上它们是＜name＞，代表抽象句法中的名称。在下面，对待它们就好像它们语法上也是＜name＞。

实体可分为实体种类。存在以下一些实体种类：

a） 包；

b） 代理（系统、功能块、进程）；

c） 代理类型（系统类型、功能块类型、进程类型）；

d） 信道、通道；

e） 信号、计时器、接口、数据类型；

f） 过程、远程过程；

g） 变量（包括形式变量）、同义词；

h） 文字、运算符、方法；

i） 远程变量；

j） 类别；

k） 状态类型；

l） 信号列表；

m） 异常。

形式正文参数是相同实体种类的一个实体，作为相应的实际正文参数。

参考定义是在＜referenced definition＞转化步骤后的实体（见第7.3节和附件F）。

每个实体在定义它的范围单元内拥有它自己的定义正文。

实体通过＜identifier＞引用。＜identifier＞内的＜qualifier＞惟一规定了实体的定义文本。

＜qualifier＞或者指的是一个父类，或者反映的是从系统或包层到定义正文的层次结构，因此系统或包层是最左边的文本部分。而后实体的*Name*通过限定词、实体名称和特征（只对h）类实体=进行描述（见第12.1.7.1、12.1.4节）。相同种类的所有实体必须具有不同的*Name*。

注1 — 因此，在相同范围单元并属于相同实体种类的两个定义不得拥有相同的＜name＞。惟一的例外是在相同＜data type definition＞中定义的操作，条件是只要它们至少有一个变元＜sort＞或结果＜sort＞不同。

出现在信道定义中的＜state name＞、＜connector name＞和＜gate name＞，＜macro formal parameter＞和＜macro name＞具有特殊的视见度规则，不得限定。其他的特殊视见度规则在适当的章节中进行解释。

注2 — 没有对应范围单元的＜scope unit kind＞，范围单元由＜task area＞和＜compound statement＞定义。因此，不可能指向在定义中引入的标识符，定义通过限定词附于这些范围单元上。

如果实体是可见的，那么可以通过使用<identifier>来引用实体。如果满足以下条件，那么在范围单元中实体是可见的：

a） 在范围单元中它有自己的定义正文；或

b） 范围单元是一个特殊化，实体在基类型中是可见的；并且

1） 通过在第12.1.9.3节中定义的特殊构件，它不受视见度保护；并且

2） 未运用数据特殊化重命名（第12.1.3节）；并且

3） 它不是一个形式正文参数，已经绑定至一个实际的正文参数（第8.2节）；或

c） 范围单元有一个<package use clause>，它指的是一个<package diagram>，因此：

1） 或者<package use clause>删去<definition selection list>，或者在<definition selection>中提到实体的<name>；并且

2） 作为实体定义正文的<package diagram>或者删去<package interface>，或者在<package interface>中提到实体的<name>；或者

d） 范围单元包含一个<interface definition>，它是实体的定义正文（见第12.1.2节）；或

e） 范围单元包含一个<data type definition>，它是实体的定义正文，通过在第12.1.9.3节中定义的特殊构件，它不受视见度保护；或

f） 实体在范围单元中是可见的，它定义了范围单元。

允许删去一些最左边的<path item>，或如果删去的<path item>能够惟一地扩充为一个完整的<qualifier>，那么可以删去<identifier>的整个<qualifier>。

当<identifier>的<name>部分表示的是一个非g）实体种类的实体时，<name>绑定至一个这样实体，即在最近包括的范围单元中它具有自己的定义正文，其中<identifier>的<qualifier>等同于完整<qualifier>（表示该范围单元）的最右边部分（通过存储器解决）。如果<identifier>不包含<qualifier>，那么有关<qualifier>匹配的要求不适用。

接下来是通过存储器将<name>绑定至定义，以通过部分<qualifier>表示的范围单元开始：

a） 如果在范围单元中存在一个惟一的实体，其<name>和实体种类相同，那么<name>绑定至该实体；否则

b） 如果范围单元是一个特殊化，那么递归地重复步骤a），直至<name>能绑定至一个实体；否则

c） 如果范围单元有一个<package use clause>，并且存在一个惟一的实体，并且在<package diagram>中是可见的，那么<name>绑定至该实体；否则

d） 如果范围单元有一个<interface definition>，并且存在一个惟一的实体，并且在<interface definition>中是可见的，那么<name>绑定至该实体；否则

e） 尝试在范围单元中运用存储器解决方案，它定义了当前的范围单元。

关于视见度和限定符的使用，与范围单元相关的<package use clause>被看做是代表一个包定义，它直接嵌套范围定义，并在范围单元中定义，在范围单元中定义该范围单元。如果<identifier>不包含一个<qualifier>，那么<package use clause>被看做是该范围单元的最近的嵌套范围单元，与之相关，并包含包可见的各实体。

注3 — 在具体句法中，包不能在其他范围单元内部进行定义。上述规则只能用于定义适用于包的视见度规则。该规则的结果是，可引用包中的名称，以便使用不同的限定符，一个对应包的一个嵌套<package use clause>。

当<identifier>中的<name>部分表示一个g＝实体种类的实体时，<name>至定义的绑定必须通过正文解决。正文解决方案在存储器解决方案之后尝试；也就是说，如果一个<name>可以通过存储器解决方案绑定至一个实体，那么使用该绑定，即使正文解决方案也能将该<name>绑定至一个实体。用于解决<name>的正文是一个<assignment>（如果<name>出现在<assignment>中＝，或者是一个<decision area>（如果<name>出现在<question>中，或<decision area>的<answer>中＝，或者是一个<expression>（它不是任何其他<expression>的一部分）。正文解决方案按以下步骤进行：

a） 对每个出现在正文中的<name>，找到<identifier>集，这样<name>部分就是可见的，具有相同的<name>，并考虑对部分<qualifier>进行重命名。

b） 构建与每个<name>相关的<identifier>集的产物。

c） 只考虑产物中那些不违反任何静态类别限制条件的元素，也考虑包中那些在<package use clause>中未变得可见的类别。每个剩余的元素代表一种可能的、静态正确的、<expression>中<name>至实体的绑定。

d） 出于<assignment>中可能的多态性（见第12.3.3节），<expression>的静态类别不能等同于<variable>的静态类别，对参数中的隐含指配也一样。此类不匹配的数量按元素逐个计算。

e） 按对比较元素，丢弃那些更不匹配的元素。

f） 如果有一个以上的剩余元素，那么所有非惟一的<identifier>都必须代表相同的*Dynamic-operation-signature*；否则正文中的<name>不能绑定至某个定义。

如果<name>或<quoted operation name>惟一地确定范围单元，那么只允许删去<path item>中可选的<scope unit kind>。

没有相应的、以**operator**或**method**表示的<scope unit kind>的抽象句法。

6.4 非正式文本

抽象语法

Informal-text :: ...

具体语法

<informal text> ::=
<character string>

语义

如果在一个规范中使用了非形式文本，那么这意味着该文本不具有任何SDL定义的语义。非正式文本的语义可以通过其他方式进行定义。

6.5 绘图规则

图形符号的尺寸可由使用者选择。

符号边界不能重叠或交叉。线形符号是这一规则的一个例外，它们可以相互交叉。在相互交叉的符号之间，没有逻辑上的联系。下面是线形符号：

<association symbol>
<channel symbol>
<create line symbol>
<dashed association symbol>
<dependency symbol>
<flow line symbol>
<solid association symbol>
<solid on exception association symbol>
<specialization relation symbol>

元符号***is followed by***意味着一个<flow line symbol>。

线形符号可以包含一段或多段直线。

当一个<flow line symbol>进入另一个<flow line symbol>、<out connector symbol>或<nextstate area>时，<flow line symbol>上需要有一个箭头。在其他情况下，<flow line symbol>上的箭头是可选的。<flow line symbol>应画成垂直的或水平的。

<input symbol>、<output symbol>、<internal input symbol>、<internal output symbol>、<priority input symbol>、<raise symbol>、<handle symbol>、<comment symbol>和<text extension symbol>的垂直镜像是允许的。

元符号***is associated with***的右边变元必须靠近其左边的变元，而不是靠近任何其他图形符号。右边变元的语法元素必须是相互可区分的。

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

6.6 图形划分

下面的图形划分定义不是具体语法的一部分，但使用了相同的元语言。

<page> ::=

<frame symbol> ***contains***
{ <heading area> <page number area> { <symbol> | <lexical unit> }* }

<heading area> ::=
<implicit text symbol> ***contains*** <heading>

<heading> ::=
<kernel heading> [<extra heading>]

<kernel heading> ::=
[<virtuality>]
<drawing kind> <drawing qualifier> <drawing name>

<drawing kind> ::=
package | **system** [**type**] | **block** [**type**] | **process** [**type**]
| **state** [**type**] | [**exported**] **procedure** | **operator** | **method**

<extra heading> ::=
图标题部分不在核心标题中

<page number area> ::=
<implicit text symbol> ***contains*** [<page number> [**(**<number of pages>**)**]]

<page number> ::=
<literal name>

<number of pages> ::=
<Natural literal name>

<symbol> ::=
以规则名称定义的任何终结符均在“*symbol*”中结束

<page>是一个起始非终结符，因此，在任何产生规则中都不引用它。一个图可以被划分成许多<page>，在这种情况下，限定该图范围的<frame symbol>和图<heading>将被每个<page>的<frame symbol>和<heading>所取代。

一个<symbol>是一个图形非终结符（见第5.4.2节）。

为使<heading area>和<page number area>之间有一清晰的间隔，<implicit text symbol>没有显示出来，而是隐含起来。<heading area>位于<frame symbol>的左上角，<page number area>位于〈frame symbol〉的右上角。允许出现在页面上的<heading>和语法单元（<symbol>和<lexical unit>）取决于图的类型。

<extra heading>必须至少在图的一页中显示，在其他页的显示可选。为了本建议书单个章节中的特殊图形，需要对<heading>和<drawing kind> 进行精心设计。本建议书不对<extra heading>做进一步定义。

<virtuality>表示由图定义的类型的虚拟性（见第8.3.2节），**exported**表示一个过程是否作为远程过程输出（见第10.5节）。

SDL的图形为<specification area>、<package diagram>、<agent diagram>、<agent type diagram>、<procedure diagram>、<operation diagram>、<composite state area>和<composite state type diagram>。

6.7 注释

注释是一个记号，用来表示与符号或文本有关的注解。

具体语法

对文本，使用了两种形式的注释。第一种形式为<note>。

第二种形式的具体句法为：

<end> ::=
[<comment>] <semicolon>

<comment> ::=
comment <comment body>

<package text area>、<agent text area>、<procedure text area>、<composite state text area>、 <operation text area>和<statement list>中的 <end>不得包含<comment>。

对符号，使用以下语法：

<comment area> ::=
<comment symbol> ***contains*** <text>
is connected to <dashed association symbol>

<comment symbol> ::=

<dashed association symbol> ::=

<dashed association symbol>的一端必须连接到<comment symbol>垂直段的中点。

借助于<dashed association symbol>，<comment symbol>可以连接到任何图形符号。通过（在想像上）完成一个长方形框以包围正文，可以把<comment symbol>看成是一个闭合的符号，它含有与该图形符号有关的注释文本。

6.8 正文扩展

具体语法

<text extension area> ::=
<text extension symbol> ***contains*** <text>
is connected to <solid association symbol>

<text extension symbol> ::=

<solid association symbol> ::=

<solid association symbol>的一端必须连接到<text extension symbol>垂直段的中点。

借助于<solid association symbol>，一个<text extension symbol>可以连接到任何图形符号。通过（在想象上）完成一个长方形框，可以把<solid association symbol>看成是一个闭合的符号。

<text extension symbol>中所包含的文本是图形符号中文本的继续，并且可认为是包含在该符号中，因此可当做许多词汇单元来处理。

6.9 文本符号

<text symbol>在任何<diagram>中使用。内容取决于图。

具体语法

<text symbol> ::=

7 SDL规范的组织

一个SDL系统通常不能简单地用一个图来描述。因此，该语言支持规范的划分，并支持从别处使用SDL。

7.1 框架

<sdl specification>可以描述成一个单一的<system specification>（可能带有许多<package diagram>变元），或描述成<package diagram>和 <referenced definition>的集合。通过在这些正文（即系统或独立的包）中“使用”包的方法，<package diagram>允许在不同的正文中使用定义。<referenced definition>是从其定义的正文中移走的定义，以获得某个系统描述的概貌。它通过引用准确地“插入”到某个地方（定义的正文）。<specification area>允许使用图形化方式来描述<system specification> 与<package diagram>之间的关系。

抽象语法

SDL-specification :: [*Agent-definition*]
*Package-definition-**set***

Agent-definition（如果有的话）必须要有一个有关带Agent-kind **SYSTEM的**Agent-type-definition 的Agent-type-identifier。

具体语法

<sdl specification> ::=
{[<specification area>]
{ <package diagram> | <system specification> } <package diagram>* <referenced definition>* }***set***

<system specification> ::=
<agent diagram>
| <typebased agent definition>[***is associated with*** <package use area>]

<specification area> ::=
<frame symbol> ***contains*** {
{ <agent reference area>
| <typebased agent definition>
[***is connected to*** {<package dependency area>+ =***set***=
}
{<package reference area>* {***set***}

语义

*SDL-specification*具有系统代理（如果有的话）和包的组合语义。若没有指定系统代理，则该规范会给出在其他规范中使用的一组定义。

对带*Agent-definition*的*SDL-specification*，如果它在*Agent-definition*中被实例化，或在潜在地实例化的类型中被实例化，那么类型是被隐含地实例化的。

模型

作为<process diagram>或<typebased process definition>的<system specification> ，是与进程具有相同名称的<system diagram>的衍生语法，含有隐含的信道，还含有作为其惟一定义的<process diagram> 或 <typebased process definition>。

作为< block diagram>或<typebased block definition>的<system specification> ，是与功能块具有相同名称的<system diagram>的衍生语法，含有隐含的信道，还含有作为其惟一定义的<block diagram> 或 <typebased block definition>。

与<system specification>的<typebased agent definition>相关的<package use area>，是与源自<typebased agent definition>的<system diagram>相关的<package use area>的衍生语法。

7.2 包

为了使类型定义能在不同的系统使用，必须将其定义成包的一部分。

可定义为包定义类型、信号列表、远程规范和同义词的一部分。

通过包使用子句，使包内的定义能被另一个范围单元见到。

抽象语法

Package-definition :: *Package-name*
*Package-definition-**set***
*Data-type-definition-**set***

*Syntype-definition-**set***
*Signal-definition-**set***
*Exception-definition-**set***
*Agent-type-definition-**set***
*Composite-state-type-definition-**set***
*Procedure-definition-**set***

具体语法

<package diagram> ::=
<frame symbol> ***contains***
{ <package heading>
{ {<package text area>}*
{<diagram in package>}* } ***set*** }
[***is associated with*** <package use area>]

<package heading> ::=
package [<qualifier>] <package name>
[<package interface>]

<package use area> ::=
<text symbol> ***contains*** {<package use clause>}*

<package text area> ::=
<text symbol> ***contains***
{ <agent type reference>
| <package reference>
| <signal definition>
| <signal reference>
| <signal list definition>
| <remote variable definition>
| <data definition>
| <data type reference>
| <procedure definition>
| <procedure reference>
| <remote procedure definition>
| <exception definition>
| <select definition>
| <macro definition>
| <interface reference> }*

<diagram in package> ::=
<package diagram>
| <package reference area>
| <entity in agent diagram>
| <data type reference area>
| <signal reference area>
| <procedure reference area>
| <interface reference area>
| <create line area>
| <option area>

<package reference> ::=
package [<qualifier>] <package name> **referenced** <end>

<package reference area> ::=
<package symbol> ***contains*** <package identifier>
[***is connected to*** {<package dependency area>+ }***set***]

<package dependency area> ::=
<dependency symbol> ***is connected to*** { <package diagram> | <package reference area> }

<package use clause> ::=
use <package identifier> [/ <definition selection list>] <end>

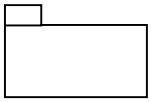
<definition selection list> ::=
<definition selection> { , <definition selection>}*

<definition selection> ::=
[<selected entity kind>] <name>

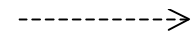
<selected entity kind> ::=
system type
| **block type**
| **process type**
| **package**
| **signal**
| **procedure**
| **remote procedure**
| **type**
| **signallist**
| **state type**
| **synonym**
| **remote**
| **exception**
| **interface**

<package interface> ::=
public <definition selection list>

<package symbol> ::=



<dependency symbol> ::=



<package use area>必须放在<frame symbol>的最上面。<package reference area>的可选的<qualifier>和 <package name>必须包含在<package symbol>的下面那个长方形中。

<package reference area>中的<package dependency area>是相应的<package diagram>中的<package use clause>的部分规范（或是<specification area>中的<package reference area>的<package> 或<system specification>），并必须与该<package use clause>一致。

对<package use clause>中提到的每个<package identifier>，都必须存在一个相应的<package diagram>。该包可以是<sdl specification>的一部分，也可以是另一个包中包含的包，要不然就必须存在访问被引用之<package diagram>的机制，使该包就像是<sdl specification>的一部分。本建议书中没有定义该机制。

如果包是<sdl specification>的一部分，或如果存在访问被引用之<package diagram>的机制，那么在<package identifier>中不必有<qualifier>。

如果相应的<package diagram>包含在另一个包中，那么<package identifier>能反映出从最外层<package diagram>到所定义的<package diagram>的层次结构。最左侧的<path item>可以被省略。

<package identifier>必须表示一个可见的包。在完全限定的<package identifier>的<qualifier>中的所有<package diagram>必须都是可见的。一个包是可见的：如果它或是 <sdl specification>的一部分，或根据 <identifier>的SDL视见度规则，其<identifier>是可见的。SDL的视见度规则意指<package identifier>可通过某个<package use clause> 使之变得可见，并且包在包含它的范围内是可见的。该范围还可以扩展至存储器包的<package use clause>。

同样，如果在<sdl specification>中省略<system specification> ，那么必须在其他<sdl specification>中存在使用该<package diagram>的机制。在其他<sdl specification>使用该<package diagram>之前，需使用宏模型和引用的定义。本建议书中并没有定义这种机制。

<selected entity kind> **procedure**用于选择（普通）过程和远程过程。如果普通过程和远程过程都有给定的<name>，那么**procedure**表示普通过程。为强制使<definition selection>表示远程过程，则可在**procedure**关键字前加上**remote**。

关键字**type**用于选择包中的类别名称和共型名称。**remote**用于选择远程变量定义。

语义

<package diagram>中定义的实体名字的视见度在第6.3节进行解释。

在**use**子句中没有变得可见的信号可以通过<signal list identifier>成为信号列表的一部分，使其在**use**子句中变得可见，从而使这些信号影响代理的完整有效输入信号集。

如果<definition selection>中有一个名称表示<sort>，那么该<definition selection>也隐含地表示定义该<sort>的数据类型，并且所有的文字和操作都由该数据类型进行定义。如果<definition selection>中有一个名字表示共型，那么该<definition selection>也隐含地表示定义该<parent sort identifier>的数据类型，并且所有的文字和操作都由该数据类型进行定义。

<definition selection>中的<selected entity kind>表示<name>的实体种类。任何一对（<selected entity kind>，<name>）都必须在<definition selection list>中严格区分。对<package interface>中的<definition selection>，只有当<package diagram>中没有其他拥有其直接定义的事件的名称时，才可以省略<selected entity kind>。对<package use clause>中的<definition selection>，当且仅当只有一个该名称的实体出现在包的<definition selection list>中，或者包没有<definition selection list>且直接包含该名称的惟一定义时，才可以省略<selected entity kind>。

模型

<system diagram>和每个<package diagram>都有一个隐含的<package use clause>：

use Predefined;

其中，Predefined表示某个包包含在附件 D中定义的预定义数据。如果没有与该图相关的<package use area>，那么创建一个<package use area>，并插入该<package use clause>。

7.3 引用的定义

具体语法

```
<referenced definition> ::=
                        <definition> | <diagram>
<definition> ::=
                        <procedure definition>
                    |   <operation definition>
                    |   <macro definition>
<diagram> ::=
                        <package diagram>
                    |   <agent diagram>
                    |   <agent type diagram>
                    |   <composite state area>
                    |   <composite state type diagram>
                    |   <procedure diagram>
                    |   <operation diagram>
```

对每个<referenced definition>，<macro definition>除外，在相关的<package diagram>或<system specification>中，必须都有一个引用。文本和图形引用分别定义为<... reference>和<… reference area>（例如，<block reference>和<block reference area>）。

在第一个关键字后，会在<referenced definition>中出现可选的<qualifier>和<name>。每个引用都会有一个与引用具有相同实体类型的<referenced definition>，并且如果有的话，其<qualifier>表示一条从包含该引用的范围单元到该引用的路径。如果相同实体类型的两个<referenced definition>拥有相同的<name>，那么其中一个的<qualifier>不可以是另一个<qualifier>的最左部分，并不能省略<qualifier>。如果<referenced definition>是一个<package diagram>，那么必须要有<qualifier>。

对<referenced definition>之外的其他定义，在第一个关键字之后均不得指定<qualifier>。

模型

在<system specification>的属性被衍生之前，每个引用都要由相应的<referenced definition>来替换。在该替换中，<referenced definition>的<qualifier>被移去。

8 结构概念

本子句引入了许多支持根据实例和应用特定概念、根据类型构建应用特定现象模型的语言机制。继承通过实例和特定应用的概念来支持。继承旨在表示概念的一般化和特殊化。

引入的语言机制可提供：

a） （纯）类型定义，它可以在系统或包的任何地方定义；

b） 基于类型的实例定义，它根据类型来定义实例或实例集；

c） 参数化的类型定义，它通过正文参数实现与封装范围的独立，并可以绑定至特定的范围；

d） 父类型定义到子类型定义的特殊化，它通过增加属性和重新定义虚类型与变迁来实现。

8.1 类型、实例和通道

在SDL描述中，实例（或实例集）的定义与类型的定义之间是有区别的。在第8.1.1节中介绍了代理的类型定义，在第8.1.3节中介绍了相应的实例规范，其他类型则：在第9.4节中介绍过程、在第10.3节中介绍信号、在第11.15节中介绍计时器、在第12.1节中介绍类别、在第12.1.2节中介绍接口。代理类型定义不连接到（通过信道）任何实例上；相反地，代理类型定义引入了通道（第8.1.5节）。在基于类型的信道实例上有连接点。

一个类型定义一组属性。类型的所有实例（第5.2.1节）都拥有这组特性。

实例（或实例集）通常有类型，如果某实例没有显式地基于某个类型，那么它是隐含的。例如，一个进程图具有一个隐含的等价匿名进程类型。

8.1.1 结构类型定义

这些是用在规范结构中的实体类型定义。相反地，过程定义也是类型定义，但组织行为更像结构。

8.1.1.1 代理类型

一个代理类型是一个系统、功能块或进程类型。当该类型用于定义代理时，该代理就具有相应的种类（系统、功能块或进程）。

抽象语法

Agent-type-definition	::	*Agent-type-name* *Agent-kind* [*Agent-type-identifier*] *Agent-formal-parameter** *Data-type-definition-**set*** *Syntype-definition-**set*** *Signal-definition-**set*** *Timer-definition-**set*** *Exception-definition-**set*** *Variable-definition-**set*** *Agent-type-definition-**set*** *Composite-state-type-definition-**set*** *Procedure-definition-**set*** *Agent-definition-**set*** *Gate-definition-**set*** *Channel-definition-**set*** [*State-machine-definition*]
Agent-kind	=	***SYSTEM*** | ***BLOCK*** | ***PROCESS***
Agent-type-identifier	=	*Identifier*
Agent-formal-parameter	=	*Parameter*
State-machine-definition	::	*State-name* *Composite-state-type-identifier*

具体语法

<agent type diagram> ::=

<system type diagram> | <block type diagram> | <process type diagram>
[***is associated with*** <package use area>

<type preamble> ::=
[<virtuality> | <abstract>]

<agent type additional heading> ::=
[<formal context parameters>] [<virtuality constraint>]
<agent additional heading>

<package use area>必须置于<frame symbol>最上面。

语义

*Agent-type-definition*定义代理类型。某种代理类型的所有代理具有那种代理类型所定义的所有属性。

一个代理类型的定义隐含着在该代理类型相同范围内的一个接口定义（见第12.1.2节）。该接口隐含定义的pid 类别以*Agent-type-name*确定，并在代理类型定义的相同范围单元内是可见的。

代理类型的全部输出集是所有信号、远程过程和远程变量的并集，或者在代理类型通道相关的输出信号列表中直接提到，或者作为接口和信号列表的一部分。

注 — 因为每个代理类型具有一个同名的隐含定义的接口，所以代理类型的名称必须与在相同范围内定义的每个显式定义的接口和每个代理（它们也有隐含接口）的名称不同，否则会存在名称冲突。

*Agent-type-definition*中定义的其他属性，如*Procedure-definition-**set***、*Agent-definition-**set***和 *Gate-definition-**set***确定了基于该类型的任何*Agent-definition*的属性，因此在第9节中进行描述。

模型

带有一个<agent body area>的代理类型是只有一种状态机而不含代理的代理类型的缩写形式。这种代理机通过用一个复合状态定义替换<agent body area>得到。该复合状态定义与代理类型名称相同，并且其*State-transition-graph*通过<agent body area>表示。

带有以下内容的代理类型：

— 带<composite state reference area> 的<state partition area>；或者

— <composite state area>

是带有基于虚拟隐含复合状态类型的状态机的代理类型的缩写形式。隐含的状态类型带有<composite state reference area>或<composite state area>的主体。如果该代理类型是一个子类型，并且如果父类型有一个<state partition area>，那么该隐含的状态类型是一个子类型，它隐含地继承父类型的隐含状态类型。

每个隐含的类型都有一个为其自身的约束（见第8.3.1节）。

8.1.1.2 系统类型

系统类型定义是顶层的代理类型定义，用关键字**system type**表示。系统类型定义不必包含在任何其他代理或代理类型定义中。系统类型既不能是抽象的，也不能是虚拟的。

具体语法

<system type diagram> ::=
<frame symbol> ***contains*** {<system type heading> <agent structure area> }
is connected to {{ <gate on diagram>* }***set*** }

<system type heading> ::=
system type [<qualifier>] <system type name>
<agent type additional heading>

<formal context parameters>的<formal context parameter>不得为<agent context parameter>、<variable context parameter> 或<timer context parameter>。

<system type diagram>中的<agent type additional heading>不可包括<agent formal parameters>。

语义

一个<system type diagram>定义一个系统类型。

8.1.1.3 功能块类型

具体语法

<block type diagram> ::=
<frame symbol> ***contains*** {<block type heading> <agent structure area> }
is connected to {{ <gate on diagram>* }***set*** }

<block type heading> ::=
<type preamble>
block type [<qualifier>] <block type name>
<agent type additional heading>

<block type diagram>中的<gate on diagram>必须置于图框之外。

语义

一个<block type diagram>定义一个功能块类型。

8.1.1.4 进程类型

具体语法

<process type diagram> ::=
<frame symbol> ***contains*** {<process type heading> <agent structure area> }
is connected to {{ <gate on diagram>* }***set*** }

<process type heading> ::=
<type preamble>
process type [<qualifier>] <process type name>
<agent type additional heading>

<process type diagram>中的<gate on diagram>必须置于图框之外。

语义

一个< process type diagram>定义一个进程类型。

8.1.1.5 复合状态类型

抽象语法

Composite-state-type-definition :: *State-type-name*
[*Composite-state-type-identifier*]
*Composite-state-formal-parameter********
*State-entry-point-definition****-set***
*State-exit-point-definition****-set***
*Gate-definition****-set***
*Data-type-definition****-set***
*Syntype-definition****-set***
*Exception-definition****-set***
*Composite-state-type-definition****-set***
*Variable-definition****-set***
*Procedure-definition****-set***
[*Composite-state-graph* | *State-aggregation-node*]

Composite-state-type-identifier = *Identifier*

具体语法

<composite state type diagram> ::=
<frame symbol>
contains {
{ <composite state type heading> | <state aggregation type heading> }
<composite state structure area>
is associated with { <state connection point>* }***set***
is connected to {{ <gate on diagram>* }***set*** }
[***is associated with*** <package use area>]

<composite state type heading> ::=
[<virtuality>]
state type [<qualifier>] <composite state type name>
[<formal context parameters>] [<virtuality constraint>]
[<specialization>]
[<agent formal parameters>]

<state aggregation type heading> ::=
[<virtuality>]
state aggregation type [<qualifier>] <composite state type name>
[<formal context parameters>] [<virtuality constraint>]
[<specialization>]
[<agent formal parameters>]

<package use area>必须置于<frame symbol>最上面。

< composite state type diagram>中的<gate on diagram>必须置于图框之外。

语义

Composite-state-type-definition 定义一个复合状态类型。复合状态类型的所有复合状态都具有该复合状态类型所定义的相同属性。其语义将在第11.11节中进行详细定义。

8.1.2 类型表达式

类型表达式用于根据一种类型来定义另一种类型，就像第8.3节中通过特殊化来定义那样。

具体语法

<type expression> ::=
<base type> [<actual context parameters>]

<base type> ::=
<identifier>

当且仅当<base type>表示一个参数化的类型时，才可规定<actual context parameters>。正文参数在第8.2节中定义。

在参数化的类型之外，参数化的类型只能在<type expression>中通过引用其<identifier>来使用。

模型

在没有实际正文参数的情况下，<type expression>产生以<base type>标识符标识的类型，或者通过将实际正文参数应用于以<base type>标识符表示的参数化类型的形式正文参数，来定义一个匿名类型。

省略了某些实际的正文参数，该类型仍会被参数化。

除了要满足<base type>表示的有关定义的任何静态条件外，<type expression>的使用还必须满足有关结果类型的任何静态条件。

注 — 在以下情况中可能会违反有关<type expression>使用的静态属性，例如：

— 当范围单元有信号正文参数或计时器正文参数时，必须依据使用的实际正文参数对激励某个状态的条件进行脱节。

— 当范围单元中的某个输出指的是一个通道或信道时，而在有通道的最近封装类型内又未做定义时，则在没有指向通道的通信路径情况下，该类型的实例化会导致一个错误的规范。

— 当某过程包含对信号标识符、远程变量和远程过程的引用，而在过程内部此类标识符的使用又违反进程正确用法时，在代理内部对该过程的特殊化会引起一个错误的规范。

— 当状态类型实例化成相同状态聚集的组成部分时，若在输入信号集中两个或更多个组成部分具有相同的信号时，则结果的复合状态是错误的。

— 当范围单元在输出动作中使用了代理正文参数时，是否存在可能的通信路径取决于使用的实际正文参数。

— 当范围单元带有类别正文参数时，若在特殊化类型中试图进行多态赋值，则实际的类别正文参数的应用会引起一个错误的规范。

— 如果在规范中增加了某过程的形式参数带有<parameter kind>的输入/输出或输出，那么父类型对子类型（用this）的调用会导致省略一个实际的输入/输出或输出参数（即导致一个错误的规范）。

— 如果某个形式的过程正文参数是用一个atleast约束定义的，并且实际的正文参数增加了一个<parameter kind>的输入/输出或输出参数，那么在参数化类型中对形式的过程正文参数的调用可能会导致省略一个实际的输入/输出或输出参数（即导致一个错误的规范）。

如果范围单元包含<specialization>，并且<type expression>中省略了所有<actual context parameter>，那么<formal context parameter>会被拷贝（仍保持原有顺序）并插入到范围单元的<formal context parameter>的前面（如果有的话）。在省略<actual context parameter>的地方，会插入对应<formal context parameter>的名称，作为<actual context parameter>。这样，这些<actual context parameter>在当前范围单元中就有了定义正文。

8.1.3 基于类型的定义

基于类型的代理定义是根据用<type expression>表示的类型来定义代理实例集。定义的实体继承了其所基于类型的所有属性。

具体语法

<typebased agent definition> ::=
<typebased system definition>
| <typebased block definition>
| <typebased process definition>

<inherited agent definition> ::=
<inherited block definition>
| <inherited process definition>

在<typebased agent definition>或<inherited agent definition>的类型表达式中用<base type>表示的代理类型在其状态机中必须包含一个未打标签的start转换。

在<typebased agent definition>中，<gate definition>和<interface gate definition>必须置于<block symbol>或 <process symbol>之外。

8.1.3.1 基于系统类型的系统定义

具体语法

<typebased system definition> ::=
<block symbol> ***contains*** <typebased system heading>

<typebased system heading> ::=
system <system name> <colon> <system type expression>

一个<typebased system definition>定义一个带*Agent-kind* **SYSTEM**的*Agent-definition*，它是以<system type expression>表示的系统类型的一个实例化。

语义

一个基于类型的系统定义可解释为是一个使用系统类型显性或衍生*Agent-type-definition*的代理。

8.1.3.2 基于功能块类型的功能块定义

具体语法

<typebased block definition> ::=
<block symbol> ***contains*** { <typebased block heading> { <gate>* }***set*** }
is connected to { {<gate property area>*}***set*** }

<typebased block heading> ::=
<block name> [<number of instances>] <colon> <block type expression>

<inherited block definition> ::=
<dashed block symbol> ***contains*** { <block identifier> { <gate>* }***set*** }
is connected to { {<gate property area>*}***set*** }

<dashed block symbol> ::=

<gate>位于符号边界附近，并与信道连接点相关。

<inherited block definition>只能出现在子类型定义中，它表示在子类型定义的父类型中定义的功能块。

注 — 允许指定连接至继承功能块通道的其他信道。

<typebased block definition>定义*Agent-kind* **BLOCK**的*Agent-definition*，它是以<block type expression>表示的功能块类型的一个实例化。

语义

一个基于类型的功能块定义可解释为是一个使用功能块类型显性或衍生*Agent-type-definition*的代理。

8.1.3.3 基于进程类型的进程定义

具体语法

<typebased process definition> ::=
<process symbol> ***contains*** { <typebased process heading> { <gate>* }***set*** }
is connected to { {<gate property area>*}***set*** }

<typebased process heading> ::=
<process name> [<number of instances>] <colon> <process type expression>

<inherited process definition> ::=
<dashed process symbol> ***contains*** { <process identifier> { <gate>* }***set*** }
is connected to { {<gate property area>*}***set*** }

<dashed process symbol> ::=

<gate>位于符号边界附近，并与信道连接点相关。

<inherited process definition>只能出现在子类型定义中，它表示在子类型定义的父类型中定义的进程。

注 — 允许指定连接至继承进程的通道的其他信道。

<typebased process definition>定义一个带*Agent-kind* **PROCESS** 的*Agent-definition*，它是以<process type expression>表示的进程类型的一个实例化。

语义

一个基于类型的进程定义可解释为是一个使用进程类型显性或衍生*Agent-type-definition*的代理。

8.1.3.4 基于复合状态类型的复合状态定义

具体语法

<typebased composite state> ::=
<state name> [<actual parameters>] <colon> <composite state type expression>

语义

一个基于类型的复合状态定义和一个文本的基于类型的状态划分定义用于定义一个通过转化从一个复合状态类型衍生出的复合状态。

模型

<typebased composite state>被转化成<composite state area>，<composite state area>具有以<composite state type expression>定义的复合状态类型的定义。

8.1.4 抽象类型

具体语法

<abstract> ::=
abstract

<abstract>是类型定义的一部分。参见第8.1.1.1、8.4、9.5、10.3、12.1.1和12.1.9.4节。

若某类型的定义包括<abstract>，则它是一个抽象类型。

抽象过程不能被调用。

基于类型的代理（见第8.1.3节）不能被指定一个抽象代理类型为类型。

抽象类型不能被实例化。不过，抽象数据类型的子类型可以被实例化，只要子类型不是它自身的抽象。

8.1.5 通道

通道在代理类型（功能块类型、进程类型）或状态类型中定义，它表示信道的连接点，将在第8.1.3节中定义的类型实例与其他实例相连，或与包含的框符号相连。

还可以在代理和复合状态中定义通道，它代表一种标记，用于规定所考虑的实体具有一个命名的连接点。

抽象语法

Gate-definition :: *Gate-name*
*In-signal-identifier**-set***
*Out-signal-identifier**-set***
Gate-name = *Name*
In-signal-identifier = *Signal-identifier*
Out-signal-identifier = *Signal-identifier*

具体语法

<gate on diagram> ::=
<gate definition> | <interface gate definition>

<gate definition> ::=
{ <gate symbol> | <inherited gate symbol> }
is associated with { <gate> [<signal list area>] [<signal list area>] }***set***
[***is connected to*** <endpoint constraint>]

<endpoint constraint> ::=
{ <block symbol> | <process symbol> | <state symbol> }
contains <textual endpoint constraint>

<textual endpoint constraint> ::=
[**atleast**] <identifier>

<interface gate definition> ::=
<gate symbol 1>
is associated with <interface identifier>

<gate symbol> ::=
<gate symbol 1> | <gate symbol 2>

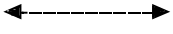
<gate symbol 1> ::=

<gate symbol 2> ::=

<inherited gate symbol> ::=
<inherited gate symbol 1> | <inherited gate symbol 2>

<inherited gate symbol 1> ::=

<inherited gate symbol 2> ::=



<gate> ::=
<gate name>

<gate on diagram>置于图框之外。

作为<gate property area>一部分的<gate definition>不能包含<endpoint constraint>。

<signal list area>的元素与通道符号的方向有关。

与<inherited gate symbol>相关的<signal list area>和<endpoint constraint>被认为是对父类型中通道定义的补充。

<inherited gate symbol>只能出现在父类型定义中，它代表具有相同<gate name>的通道（在子类型定义的父类型中规定）。

<gate symbol>上的每个箭头，可以有一个<signal list area>。<signal list area>必须明确地足够靠近与其相关的箭头。箭头指明<signal list area>是代表*In-signal-identifier-**set***还是代表*Out-signal-identifier-**set***，并指出<signal list>的方向（自某类型或到某类型）。*In-signal-identifier*代表到通道的<signal list>中的一个元素。*Out-signal-identifier*代表自通道的<signal list>中的一个元素。

带有<block symbol>（<process symbol>、<state symbol>）的<endpoint constraint>的<identifier>必须指明功能块类型的定义（分别为进程类型、状态类型）。

连接到通道的信道必须与通道的端点约束相一致。满足下列条件就认为通道与约束相一致，即如果信道的另一端是一个代理或类型状态（由端点约束中的<identifier>表示），或是该类型的一个子类型（当它包含带有**atleast**的<textual endpoint constraint>时），并且信道上的信号集（如果规定的话）与各自方向上的通道规定的信号集相等或是它的一个子集。

如果<typebased block definition>或<typebased process definition>中以<base type>表示的类型包含信道，那么应使用下列规则：对该类型定义的通道的<signal list>的通道、信号和方向的每一个组合，该类型必须在给定的方向上至少包含一个信道，该信道连接到该通道的框架上，并且或者提到信号，或者没有显性相关的<signal list>。

如果类型包含提及远程过程或远程变量的信道，那么上述规则同样适用。

若指定了两个<gate constraint>，则其中一个肯定在另一个的反方向上，并且这两个<gate constraint>的<textual endpoint constraint>必须相同。

如果为父类型中的通道指定了<textual endpoint constraint>，那么（新增的）<textual endpoint constraint>的<identifier>必须与父类型的<textual endpoint constraint>中表示的类型相同或者是它的一个子类型。

<textual endpoint constraint>的<identifier>必须表示一个功能块、进程类型或状态类型定义。

<interface gate definition>的<interface identifier>不能是该通道连接的实体隐含定义的接口（见第12.1.2节）。

语义

类型定义中通道的使用对应实例规范（集）中封装范围中的通信路径的使用。

模型

<interface gate definition>是接口名为<gate name>、<interface identifier>为<gate constraint>或<signal list area>的<gate definition>的缩写形式。

8.2 文本参数

为了使类型定义能在不同的正文中使用，即不但可以在相同的系统规范内使用，也可以在不同的系统规范内使用，规定类型可以通过正文参数进行参数化。正文参数由实际的正文参数替换，如第8.1.2节中定义。

以下类型定义可以有形式的正文参数：系统类型、功能块类型、进程类型、过程、信号、复合状态、接口和数据类型。

正文参数可以是给定的约束（即由相应的实际标识符表示的任何实体必须要有必要的属性）。正文参数在类型内部具备这些属性。

具体语法

<formal context parameters> ::=
<context parameters start> <formal context parameter list> <context parameters end>

<formal context parameter list> ::=
<formal context parameter> {<end> <formal context parameter> }*

<actual context parameters> ::=
<context parameters start> <actual context parameter list> <context parameters end>

<actual context parameter list> ::=
[<actual context parameter>] {，[<actual context parameter>] }*

<actual context parameter> ::=
<identifier> | <constant primary>

<context parameters start> ::=
<less than sign>

<context parameters end> ::=
<greater than sign>

<formal context parameter> ::=
<agent type context parameter>
| <agent context parameter>
| <procedure context parameter>
| <remote procedure context parameter>
| <signal context parameter>
| <variable context parameter>
| <remote variable context parameter>
| <timer context parameter>
| <synonym context parameter>
| <sort context parameter>
| <exception context parameter>
| <composite state type context parameter>
| <gate context parameter>
| <interface context parameter>

带有形式正文参数的类型定义的范围单元定义了形式正文参数的名字。因此，这些名字在类型定义中是可见的，并且在形式正文参数的定义中也是可见的。

<actual context parameter>不能是一个<constant primary>，除非它是一个同义词正文参数。<constant primary>是一个<primary>，它是一个有效的<constant expression>（见第12.2.1节）。

形式的正文参数既不能在<type expression>中用做<base type>，也不能用在<formal context parameters>的**atleast**约束中。

约束通过约束规范来规定。约束规范引入了后面跟着约束特征或**atleast**子句的形式正文参数的实体。约束特征直接引入了形式正文参数的足够属性。**atleast**子句表示形式的正文参数必须要用实际的正文参数替换，它与**atleast**子句中确定的类型相同或是它的一个子类型。该子句中跟在关键字**atleast**后面的标识符必须确定正文参数的实体种类的类型定义，并且既不能是形式的正文参数，也不能是参数化的类型。

一个类型的形式正文参数只能绑定到一个满足形式参数约束条件的相同实体种类的实际正文参数上。

除第8.1.2节中列出的那些情况外，参数化类型只能使用正文参数的属性，它们通过约束给出。

在约束中使用其他正文参数的正文参数在其他参数被绑定之前不能被绑定。

<actual context parameters>中可以省略拖尾的逗号。

模型

如果某个类型定义既不是子类型定义，也不是通过在<type expression>中绑定的形式正文参数进行定义，那么该类型定义的形式正文参数就是在<formal context parameters>中规定的参数。

一个类型的正文参数在<type expression>定义中被绑定到实际的正文参数上。在这种绑定中，参数化类型内的各形式正文参数事件都被实际的参数所替换。在<formal context parameter>中所包含的标识符绑定到定义期间（即衍生出其限定词，见第6.3节），除<formal context parameters>之外的其他局部定义均被忽略。

参数化类型不能是实际的正文参数。为了使定义可以作为实际的正文参数，它必须与形式参数具有相同的实体种类，并满足形式参数的约束要求。

如果某范围单元包含＜specialization＞，那么＜specialization＞中任何省略的实际正文参数都被＜type expression＞中＜base type＞的相应＜formal context parameter＞所替换，并且该＜formal context parameter＞成为范围单元的形式正文参数。

8.2.1 代理类型正文参数

具体语法

＜agent type context parameter＞ ::=
{**process type** | **block type**} ＜agent type name＞ [＜agent type constraint＞]

＜agent type constraint＞ ::=
atleast ＜agent type identifier＞ | ＜agent signature＞

一个实际的代理类型参数必须是约束代理类型（**atleast** ＜agent type identifier＞）的子类型，那些约束类型没有其他的形式参数；或者它必须兼容于形式的代理特征。

如果一个代理类型定义与形式代理特征具有相同的种类，并且代理类型定义的形式参数与＜agent signature＞的相应＜sort＞具有相同的类别，那么认为该代理类型定义兼容于形式代理特征。

8.2.2 代理正文参数

具体语法

＜agent context parameter＞ ::=
{ **process** | **block** } ＜agent name＞ [＜agent constraint＞]

＜agent constraint＞ ::=
{ **atleast** | ＜colon＞ } ＜agent type identifier＞ | ＜agent signature＞

＜agent signature＞ ::=
＜sort list＞

一个实际的代理参数必须确定一个代理定义。它的类型必须是约束代理类型（**atleast** ＜agent type identifier＞）的子类型，那些约束类型没有其他的形式参数；或者它必须是以＜agent type identifier＞（＜colon＞ ＜agent type identifier＞）表示的类型；或者它必须与形式的＜agent signature＞兼容。

如果代理定义的形式参数与＜agent signature＞的相应＜sort＞或＜agent formal parameters＞具有相同的类别，并且两个定义具有相同的*Agent-kind*，那么认为该代理定义兼容于形式的＜agent signature＞。

8.2.3 过程正文参数

具体语法

＜procedure context parameter＞ ::=
procedure ＜procedure name＞ ＜procedure constraint＞

＜procedure constraint＞ ::=
atleast ＜procedure identifier＞ | ＜procedure signature in constraint＞

＜procedure signature in constraint＞ ::=
[**(**＜formal parameter＞ { **,** ＜formal parameter＞ }* **)**] [＜result＞]

一个实际的过程参数必须确定一个过程定义，它或者是约束过程的特殊化（**atleast** ＜procedure identifier＞），或者是兼容于形式的过程特征。

一个过程定义兼容于形式的过程特征应满足以下条件：

a） 如果过程定义的形式参数与相应的特征参数具有相同的＜parameter kind＞，并且二者具有相同的结果，即都为＜sort＞，或者都不返回结果，那么认为过程定义的形式参数与相应的特征参数应有相同的类别；或

b） 过程定义中的每个**in**/**out**和**out**参数与相应的特征参数具有相同的＜sort identifier＞ 或＜syntype identifier＞。

8.2.4 远程过程正文参数

具体语法

<remote procedure context parameter> ::=
remote procedure <procedure name> <procedure signature in constraint>

一个**remote**过程正文参数的实际参数必须确定一个具有相同特征的<remote procedure definition>。

8.2.5 信号正文参数

具体语法

<signal context parameter> ::=
signal <signal name> [<signal constraint>]
{ , <signal name> [<signal constraint>] }*

<signal constraint> ::=
atleast <signal identifier> | <signal signature>

<signal signature> ::=
<sort list>

一个实际的信号参数必须确定一个信号定义，它或者是约束的信号类型的子类型（**atleast** <signal identifier>），或者兼容于形式的信号特征。

语义

如果信号类别与类别约束列表中的类别相同，那么认为该信号定义兼容于形式的信号特征。

8.2.6 变量正文参数

具体语法

<variable context parameter> ::=
dcl <variable name> { , <variable name>}* <sort>
{ , <variable name> { , <variable name>}* <sort> }*

一个实际的参数必须是一个变量、一个形式代理或与约束类别具有相同类别的过程参数。

8.2.7 远程变量正文参数

具体语法

<remote variable context parameter> ::=
remote <remote variable name> { , <remote variable name>}* <sort>
{ , <remote variable name> { , <remote variable name>}* <sort> }*

一个实际的参数必须确定一个具有相同类别的<remote variable definition>。

8.2.8 计时器正文参数

具体语法

<timer context parameter> ::=
timer <timer name> [<timer constraint>]
{ , <timer name> [<timer constraint>] }*

<timer constraint> ::=
<sort list>

一个实际的计时器参数必须确定一个兼容于形式的类别约束列表的计时器定义。如果计时器的类别与类别约束列表中的类别相同，那么认为该定时器定义兼容于形式类别的约束列表。

8.2.9 同义词正文参数

具体语法

<synonym context parameter> ::=
synonym <synonym name> <synonym constraint>
{, <synonym name> <synonym constraint> }*

<synonym constraint> ::=
<sort>

一个实际的同义词必须是一个与约束类别具有相同类别的约束表达式。

模型

如果实际的参数是一个<constant expression>（而不是一个<synonym identifier>），那么在该类型周围的文本中会存在一个匿名同义词的隐含定义，类型用文本参数定义。

8.2.10 类别文本参数

具体语法

<sort context parameter> ::=
[{ **value** | **object** }] **type** <sort name> [<sort constraint>]

<sort constraint> ::=
atleast <sort> | <sort signature>

<sort signature> ::=
literals <literal signature> { **,** <literal signature> }*
[**operators** <operation signature in constraint> { **,** <operation signature in constraint> }*]
[**methods** <operation signature in constraint> { **,** <operation signature in constraint> }*]
| **operators** <operation signature in constraint> { **,** <operation signature in constraint> }*
[**methods** <operation signature in constraint> { **,** <operation signature in constraint> }*]
| **methods** <operation signature in constraint> { **,** <operation signature in constraint> }*

<operation signature in constraint> ::=
<operation name>[**(**<formal parameter> { **,** <formal parameter> }* **)**] [<result>]
| <name class operation> [<result>]

如果省略<sort constraint>，那么实际的类别可以是任何类别。否则一个实际的类别必须是约束类别的不带<renaming>的子类型，或者兼容于形式的类别特征。

如果一个类别的文字包含在形式类别特征中的文字，引入该类别的数据类型所定义的操作包含形式类别特征中的操作，并且操作具有相同的特征，那么认为该类别兼容于形式的类别特征。

<literal signature>不可以包含<named number>。

模型

如果给出了关键字**value**，并且实际的类别是对象类别，那么实际的参数被当做扩充的类别**value** <sort identifier>来看待。如果给出了关键字**object**，并且实际的类别是值类别，那么实际的参数被当做引用类别**object** <sort identifier>来看待。

8.2.11 异常正文参数

具体语法

<exception context parameter>::=
exception <exception name> [<exception constraint>]
{ **,** <exception name> [<exception constraint>] }*

<exception constraint>::=
<sort list>

一个实际的异常参数必须确定一个带相同特征的异常。

8.2.12 复合状态类型正文参数

具体语法

<composite state type context parameter> ::=
state type <composite state type name> [<composite state type constraint>]

<composite state type constraint> ::=
atleast <composite state type identifier> | <composite state type signature>

<composite state type signature> ::=
<sort list>

一个实际的复合状态类型参数必须确定一个复合状态类型定义。它的类型必须是约束复合状态类型（**atleast** <composite state type identifier>）的子类型，那些约束类型没有其他的形式参数；或者它必须兼容于形式的复合状态类型特征。

如果复合状态类型定义的形式参数与相应的<composite state type constraint>的<sort>具有相同的类别，那么认为该复合状态类型定义兼容于形式的复合状态类型特征。

8.2.13 通道正文参数

具体语法

<gate context parameter> ::=
gate <gate> <gate constraint> [<gate constraint>]

<gate constraint> ::=
{ **out** [**to** <textual endpoint constraint>] | **in** [**from** <textual endpoint constraint>] }
[**with** <signal list>]

<gate constraint>中的**out**或**in**表示<signal list>的方向，分别自或到该类型。来自实例被定义处的类型在<gate constraint>中必须要有一个<signal list> 。

一个实际的通道参数必须确定一个通道定义。它向外的通道约束必须包含相应的形式通道文本参数的<signal list>中提到的所有元素。它形式通道正文参数的向内通道约束必须包含实际通道参数的<signal list>中的所有元素。

8.2.14 接口文本参数

具体语法

<interface context parameter> ::=
interface <interface name> [<interface constraint>]
{ **,** <interface name> [<interface constraint>] }*

<interface constraint> ::=
atleast <interface identifier>

一个实际的接口参数必须确定一个接口定义。该接口类型必须是该约束（**atleast** <interface identifier>）的接口类型的一个子类型。

8.3 特殊化

一个类型可以定义成另一个类型（父类型）的特殊化，获得一个新的子类型。一个子类型除了具有父类型的属性外，还可以有其他属性，并且它可以重新定义虚拟局部类型和转换。除接口情况外，最有可能的是一个父类型。

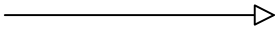
虚拟类型可以是给定的约束（即虚拟类型的任何重新定义都必须具备的属性）。这些属性用于保证任何重新定义的属性。虚拟类型在第8.3.2节中定义。

8.3.1 增加的特性

具体语法

<specialization> ::=
inherits <type expression> [**adding**]

<specialization area> ::=
<specialization relation symbol>
[***is associated with*** <actual context parameters>]
is connected to <type reference area>

<specialization relation symbol> ::=


<specialization relation symbol>的箭头指向<type reference area>。连接到箭头的类型是父类型，而其他类型是子类型。被连接的引用都必须具有相同的种类。正文参数的相关绑定对应作为带有实际正文参数的类型表达式的父类型。

<type expression>表示基类型。基类型被称为特殊化类型的父类型，而特殊化类型被称为基类型的子类型。子类型的任何特殊化也是基类型的子类型。

如果类型subT是一个（父）类型T的子类型（直接地或间接地），那么：

a） T必须包含subT；

b） T不可以是subT的一个特殊化；

c） 由T包含的定义不可以是subT的特殊化。

对代理类型情况，这些规则也必须适用于包含在T中的各定义；另外，直接或间接地被T包含的定义不可以是subT的基于类型的定义。

在下列中的<specialization>的<type expression>：

a） 在<agent additional heading>中的<specialization>的<type expression>代表第8.1.1.1节中的*Agent-type-definition*的*Agent-type-identifier*。

b） 在<composite state type heading>或 <state aggregation type heading>中的<specialization>的<type expression>代表第8.1.1.5节中的*Composite-state-type-definition*的*Composite-state-type-identifier*。

c） 在<procedure heading>中的<specialization>的<type expression>代表第9.4节中的*Procedure-definition*的*Procedure-identifier*。

数据类型特殊化的具体句法见第12.1.3节。

语义

带局部定义的一个特殊化类型定义的结果内容由父类型的内容跟着特殊化定义的内容组成。这意味着特殊化定义的定义集是特殊化定义自身内容和父类型内容的结合。定义的结果集必须遵守在第6.3节中给出的有关不同名称的规则。不过，该规则的例外有：

a） 一个虚拟类型的重新定义是一个与虚拟类型同名的定义；

b） 在子类型中可以为父类型的通道给出一个扩展的定义（根据传送的信号和端点约束） — 这通过与父类型同名的通道定义来指定。

c） 如果<type expression>包含<actual context parameters>，那么用子类型的名称替换<type expression>的<base type>的任何事件。

d） 如果特殊化运算符的特征等同于基类型运算符的特征，那么不继承父类型的运算符。

e） 如果运算符或方法的特征等同于特殊化运算符或子类型中已存在之方法的特征，那么不继承父类型的运算符或非虚拟方法。

子类型的形式正文参数是非绑定的，父类型定义的形式正文参数后跟特殊化类型的形式正文参数（见第8.2节）。

特殊化代理类型的形式参数是代理父类型的形式参数，后跟在特殊化中增加的形式参数。

特殊化过程的形式参数是带形式参数（在特殊化中增加）的过程的形式参数。如果在特殊化之前过程有一个<procedure result>，那么在特殊化中增加的参数被插在最后一个参数（结果的**out**参数）之前，否则插入在最后一个参数之后。

特殊化代理类型的完整合法输入信号集是该特殊化代理类型的完全合法输入信号集与代理父类型的全部合法输入信号集的联合。

特殊化代理类型、过程定义或状态类型的结果图由其父类型定义的图构成，后跟特殊化代理类型、过程定义或状态类型的图。

一个给定代理类型、过程定义或状态类型的状态转换图至多只能有一个没有标签的起始转换。

特殊化信号定义可以向父类型的类别列表添加（通过增加）类别。

特殊化数据类型定义可以向继承的类型构造器添加文字、字段或选择，还可以增加运算符和方法，还可以增加缺省初始化或缺省指配。

特殊化复合状态类型的形式参数是带形式参数（在特殊化中增加）的复合状态类型的形式参数。

注 — 当子类型中的一个通道是从父类型继承之通道的一个扩展时，在具体语法中使用＜inherited gate symbol＞。

8.3.2 虚拟类型

当代理类型、过程或状态类型定义成另一类型的局部代理类型、过程或状态类型时（以*enclosing*类型表示），则该代理类型、过程或状态类型可以被规定为虚拟类型。虚拟类型可以在嵌套类型的特殊化中进行重新定义。

具体语法

＜virtuality＞ ::=

virtual | redefined | finalized

＜virtuality constraint＞ ::=

atleast ＜identifier＞

＜virtuality＞和＜virtuality constraint＞是类型定义的一部分。

虚拟类型是带**virtual**或**redefined**的类型，如＜virtuality＞。重新定义的类型是带**redefined**或**finalize**的类型，如＜virtuality＞。只有虚拟类型可以被重新定义。

每个虚拟类型都关联一个虚拟约束，它有一个与虚拟类型具有相同实体种类的＜identifier＞。如果规定了＜virtuality constraint＞，那么虚拟约束就是所包含的＜identifier＞；否则虚拟约束是衍生的，如下所述。

虚拟类型及其约束不能带有正文参数。

只有虚拟类型可以规定＜virtuality constraint＞。

如果＜virtuality＞既出现在引用中，又出现在被引用的定义中，那么它们必须相同。如果＜procedure preamble＞既出现在过程引用中，又出现在被引用的过程定义中，那么它们必须相同。

虚拟代理类型必须与其约束具有完全相同的形式参数，至少是相同的通道和接口以及定义。虚拟状态类型必须至少要与其约束具有相同的状态连接点。虚拟过程必须与其约束具有完全相同的形式参数。在第8.3.4节给出了有关虚拟运算符和方法之变元的限制。

如果既使用了**inherits**又使用了**atleast**，那么继承的类型必须与约束相同或是约束的子类型。

在隐性约束情况下，有关**inherits**的重新定义必须是约束的一个子类型。

语义

虚拟类型可以在该虚拟类型嵌套类型的子类型定义中进行重新定义。在子类型中，它是来自子类型的定义，用于定义该虚拟类型的实例类型，当将虚拟类型用于自父类型继承的子类型的一部分时也如此。没有在子类型定义中重新定义的虚拟类型具有父类型定义中给出的定义。

不过，通过一个表示其中一个父类型的限定词来访问虚拟类型意味着：以该限定词表示的实际父类型中给出的虚拟类型的（重新）定义的应用。它的名称通过T的重新定义而藏在一个嵌套子类型中的类型T，可以通过带父类型名称（即在继承链中的类型名称）的限定词变为可见。该限定词将只包含一个表示特定父类型的路径项。

没有显性给出＜specialization＞的虚拟类型或重新定义类型可以有一个隐含的＜specialization＞。虚拟约束和可能的隐性＜specialization＞通过以下方式获得。

对虚拟类型V和V的重新定义类型R，有以下规则适用（以给定的次序使用所有规则）：

a） 如果虚拟类型V没有＜virtuality constraint＞，那么类型V的约束VC与虚拟类型V相同，并表示类型V；否则，约束VC用＜virtuality constraint＞（用类型V给出）来标识；

b） 如果虚拟类型V没有＜specialization＞，并且约束VC为类型V，那么类型V 没有隐含的特殊化；

c） 如果虚拟类型V没有＜specialization＞，并且约束VC不是类型V，那么隐含的特殊化类型VS与约束VC相同；

d） 如果虚拟类型V的＜specialization＞出现，那么特殊化类型VS必须与约束VC相同或是约束VC的一个子类型；

e） 如果重新定义的类型R没有＜virtuality constraint＞，那么类型R的约束RC与类型R相同；否则，约束RC用＜virtuality constraint＞（用类型R给出）来标识；

f） 如果重新定义的类型R没有<specialization>，那么R的隐含特殊化类型RS与类型V的约束VC相同；否则特殊化类型RS用带类型R的显性<specialization>来标识；

g） 约束RC必须与约束VC相同或是约束VC的一个子类型；

h） R的特殊化类型RS必须与约束RC相同或是约束VC的一个子类型；

i） 如果R是一个虚拟类型（重新定义的而不是最终确定的），那么适用于V的规则同样适用于R。

虚拟类型的子类型是初始虚拟类型的子类型，不可能被重新定义。

8.3.3 虚拟转换/保存

通过关键字**virtual**，将进程类型、状态类型或过程的转换或保存规定为虚拟转换或虚拟保存。虚拟转换或保存可以在特殊化中进行重新定义。它们分别用相同状态或信号的转换或保存，以及用关键字**redefined**或**finalized**来指明。

具体语法

虚拟转换和保存的语法将在第9.4节（虚拟过程开始）、第10.5节（虚拟远程过程输入和保存）、第11.1节（虚拟进程开始）、第11.3节（虚拟输入）、第11.4节（虚拟优先级输入）、第11.5节（虚拟连续信号）、第11.7节（虚拟保存）、第11.9节（虚拟自发转换）和第11.16.3节（虚拟句柄）中进行介绍。

虚拟转换或保存不得出现在代理（实例集）定义中，或出现在复合状态定义中。

一个状态不得有一个以上的虚拟自发转换。

以**redefined**标记的特殊化中的重新定义可能会在进一步的特殊化中被不同地定义，而以**finalized**标记的重新定义不得在进一步的特殊化中给出新的定义。

带<virtuality>的输入或保存不得包含<asterisk>。

语义

虚拟转换/保存的重新定义紧密对应虚拟类型的重新定义（见第8.3.2节）。

虚拟起始转换可被重新定义为新的起始转换。

虚拟优先级输入或输入转换可被重新定义为新的优先级输入或输入转换，或重新定义为保存。

虚拟保存可被重新定义为优先级输入、输入转换或保存。

虚拟自发转换可被重新定义为新的自发转换。

虚拟句柄转换可被重新定义为新的虚拟句柄转换。

虚拟连续转换可被重新定义为新的连续转换。该重新定义以与重新定义的连续转换相同的状态和优先级（如果有的话）来指明。如果在一个状态中存在若干个虚拟连续转换，那么这些转换的优先级必须不同。如果在一个状态中只存在一个虚拟连续转换，那么可以忽略优先级。

虚拟远程过程输入转换的一个转换可被重新定义为一个新的远程过程输入转换或一个远程过程保存。

虚拟远程过程保存可被重新定义为一个远程过程输入转换或一个远程过程保存。

有关虚拟输入转换的转化适同样用于虚拟远程过程输入转换。

在子类型中，虚拟转换或保存是用子类型定义来定义的。在子类型定义中没有被重新定义的虚拟转换或保存具有父类型定义中给出的定义。

8.3.4 虚拟方法

通过<virtuality>中的关键字**virtual**将数据类型方法规定为虚拟方法。虚拟方法可以在特殊化中被重新定义。这通过带相同<operation name>以及<virtuality>中带关键字**redefined**或**inalized**的方法来指明。

如果衍生类型对重新定义的方法只包含一个<operation signature>而不包含<operation definition>、<operation reference>或 <external operation definition>，那么只改变重新定义方法的特征。

具体语法

虚拟方法的语法在第12.1.4节中介绍。

当方法在特殊化中被重新定义时，其特征在类别上必须兼容于基础类型中的对应特征，此外，如果*Operation-signature*中的*Result*表示类别A，那么重新定义方法的*Result*可以只表示类别B，这样B将在类别上兼容于A。

虚拟方法的重新定义不得改变被继承<operation signature>的任何<argument>中的<parameter kind>。

虚拟方法的重新定义不得向<operation signature>的任何<argument>增加<argument virtuality>。

语义

没有<virtuality constraint>的虚拟方法对重新定义没有限制，也仅在这种情况下如此。

虚拟方法的重新定义与虚拟类型的重新定义紧密对应（见第8.3.2节）。

8.3.5 虚拟缺省初始化

该子句描述了虚拟缺省初始化，如第12.3.3.2节中介绍。

通过<virtuality>中的关键字**virtual**将数据类型实例的缺省初始化规定为虚拟的。虚拟缺省初始化可以在特殊化中被重新定义。这通过<virtuality>中带关键字**redefined**或**finalized**的缺省初始化来指明。

如果衍生类型在其缺省初始化中不包含<constant expression>，那么该衍生类型没有缺省初始化。

具体语法

虚拟缺省初始化的语法在第12.3.3.2节中介绍。

语义

虚拟缺省初始化的重新定义与虚拟类型的重新定义紧密对应（见第8.3.2节）。

8.4 类型参考

类型图和实体类型定义中可以有类型引用。一个类型引用既规定了类型在包含定义或图的范围单元内进行定义（但在被引用的定义或图中做充分描述），又规定了该类型具有作为类型引用一部分的部分指定的属性。被引用的定义或图定义了该类型的属性，而类型引用只有部分定义。这就要求作为类型引用一部分的部分规范要符合该类型定义或图的规范的要求。例如，一个变量的部分规范可以给出变量名称但不能给出变量类别。在被引用的定义中，必须存在具有该名称的一个变量，并且在该定义中必须规定一个类别。

同一类型定义可以有若干个类型引用。不带限定词的引用都必须在同一范围单元中，并且该类型定义被插入到该范围单元中。

具体语法

<agent type reference> ::=
<system type reference>
| <block type reference>
| <process type reference>

<agent type reference area> ::=
{ <system type reference area>
| <block type reference area>
| <process type reference area> }
is connected to { <gate property area>* }***set***

如果代理的某个<agent type reference area>是通过<agent type diagram>进行定义的，那么与<agent type reference area>相关的<gate property area>对应与<agent type diagram>相关的<gate on diagram>。如果没有<gate property area>与<agent type reference area>相关，那么必须包含<signal list item>，<signal list item>不能出现在与<agent type diagram>相关的<gate on diagram>中。

<system type reference> ::=
system type <system type identifier> <type reference properties>

<system type reference area> ::=
<type reference area>

作为<system type reference area>一部分的<type reference area>必须有一个带<system type name>的<type reference heading>。

<block type reference area> ::=
<type reference area>

<block type reference> ::=
<type preamble>
block type <type reference heading> <type reference properties>

作为<block type reference>一部分的<type reference heading>必须有一个<block type name>。

作为<block type reference area>一部分的<type reference area>必须有一个带<block type name>的<type reference heading>。

<process type reference> ::=
<type preamble>
process type <type reference heading> <type reference properties>

<process type reference area> ::=
<type reference area>

作为<process type reference>一部分的<type reference heading>必须有一个<process type name>。

作为<process type reference area>一部分的<type reference area>必须有一个带<process type name>的<type reference heading>。

<composite state type reference> ::=
<type preamble>
state type <type reference heading> <type reference properties>

<composite state type reference area> ::=
<type reference area>
is connected to {<gate property area>*}***set***

作为<composite state type reference>一部分的<type reference heading>必须有一个<composite state type name>。

作为<composite state type reference area>一部分的<type reference area>必须有一个带<composite state type name>的<type reference heading>。

如果复合状态的某个<composite state type reference area>是通过<composite state type diagram>进行定义的，那么与<composite state type reference area>相关的<gate property area>对应与<composite state type diagram>相关的<gate on diagram>。如果没有<gate property area>与<composite state type reference area>相关，那么必须包含<signal list item>，<signal list item>不能出现在与<composite state type diagram>相关的<gate on diagram>中。

<procedure reference> ::=
<type preamble> [**exported** [**as** <remote procedure identifier>]]
procedure <type reference heading> <type reference properties>

<procedure reference area> ::=
<type reference area>

作为<procedure reference area>一部分的<type reference area>必须有一个带<procedure name>的<type reference heading>。

作为<procedure reference> 一部分的<type reference heading>必须有一个<procedure name>。

<signal reference> ::=
<type preamble>
signal <type reference heading> <type reference properties>

<signal reference area> ::=
<type reference area>

作为<signal reference>一部分的<type reference heading>必须有一个<signal name>。

作为<signal reference area>一部分的<type reference area>必须有一个带<signal name>的<type reference heading>。

<data type reference> ::=
<type preamble>
{ **value** | **object** } **type** <data type identifier> <type reference properties>

<data type reference area> ::=
<type reference area>

作为<data type reference>一部分的<type reference heading>必须有一个<data type name>。

作为<data type reference area>一部分的<type reference area>必须有一个带<data type name>的<type reference heading>。

<interface reference> ::=
[<virtuality>]
interface <type reference heading> <type reference properties>

<interface reference area> ::=
<type reference area>

作为<interface reference>一部分的<type reference heading>必须有一个<interface name>。

作为<interface reference area>一部分的<type reference area>必须有一个带<interface name>的<type reference heading>。

<operation reference> ::=
{ **operator** | **method** } <operation signature> **referenced** <end>

如果同名的相同类别内没有其他<operation reference>，并且存在一个<operation signature>，那么可以省略<operation reference>中的<arguments>和<result>。在这种情况下，<arguments>和<result>衍生自<operation signature>。

<type reference area> ::=
{ <basic type reference area> | <iconized type reference area> }
is connected to { <package dependency area>* <specialization area>* } ***set***

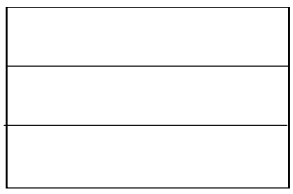
<type reference area>中的<package dependency area>是类型图对应<package use clause>的部分规范，并必须符合它的要求。

应利用不带箭头的<specialization relation symbol>末端，将<specialization area>连接至<basic type reference area>或<iconized type reference area>的上面部分。除接口引用外，所有的<type reference area>只能有一个<specialization area>。

<specialization area>对应于被引用类型的<specialization>。被连接的<type reference area>必须对应<specialization>的<type expression>中的<base type>。<specialization area>中的<actual context parameters>必须对应<type expression>中的<actual context parameters>。

<basic type reference area> ::=
<class symbol>
contains {
<graphical type reference heading>
<attribute properties area>
<behaviour properties area> }

<class symbol> ::=



将<class symbol>分成三格的两条直线的相对位置可以与显示的不同。

<graphical type reference heading> ::=

{ <type reference kind symbol> ***contains*** **system** | <system type symbol> }
<system type type reference heading>
| { <type reference kind symbol> ***contains*** **block** | <block type symbol> }
<block type type reference heading>
| { <type reference kind symbol> ***contains*** **process** | <process type symbol> }
<process type type reference heading>
| { <type reference kind symbol> ***contains*** **state** | <composite state type symbol> }
<composite state type type reference heading>
| { <type reference kind symbol> ***contains*** **procedure** | <procedure symbol> }
<procedure type reference heading>
| <type reference kind symbol> ***contains*** **signal**
<signal type reference heading>
| { <type reference kind symbol> ***contains*** { **value** | **object** } | <data symbol> }
<data type type reference heading>
| { <type reference kind symbol> ***contains*** **interface** | <data symbol> }
<interface type reference heading>

<graphical type reference heading>应置于包含的<class symbol>的最上面那个格中。

<type reference heading> ::=
<type preamble>
[**exported** [**as** <remote procedure identifier>]]
[<qualifier>] <name> [<formal context parameters>]

<type preamble>必须对应被引用类型的<type preamble>。如果该引用是虚拟的，那么被引用类型必须是虚拟的。如果引用是抽象的，那么被引用类型必须是抽象的。如果在<type reference heading>中给出了**exported**，那么被引用类型必须是一个输出过程，如果还给出了<remote procedure identifier>，那么过程还必须标识相同的远程过程定义。

<formal context parameters>对应被引用类型的<formal context parameters>。<formal context parameter list>必须对应被引用类型的<formal context parameter list>。

<type reference kind symbol> ::=
" "

<type reference kind symbol>置于<type reference heading>的上面或左边。

<data symbol> ::=

注1 — <data symbol>是一个不带任何可见框的长方形。这意味着不包含<type reference kind symbol>的<graphical type reference heading>实际上包含一个<data symbol>。

<data symbol> 对应一个<data type definition>或<interface definition>。

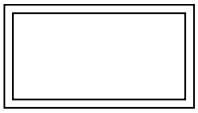
如果<graphical type reference heading>包含的是一个符号而不是<type reference kind symbol>，那么该符号必须置于<graphical type reference heading>的右上角。

<iconized type reference area> ::=
<system type symbol> ***contains*** <system type type reference heading>
| <block type symbol> ***contains*** <block type type reference heading>
| <process type symbol> ***contains*** <process type type reference heading>
| <composite state type symbol> ***contains*** <composite state type type reference heading>
| <procedure symbol> ***contains*** <procedure type reference heading>

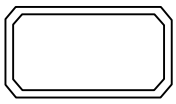
注2 — 没有<iconized type reference area>对应信号、接口、对象和值类型。

<system type symbol> ::=
<block type symbol>

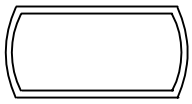
<block type symbol> ::=



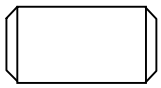
<process type symbol> ::=



<composite state type symbol> ::=



<procedure symbol> ::=



<gate property area> ::=

<gate definition> | <interface gate definition>

<attribute properties area> ::=

{ { <attribute property> <end> }* }***set***

<attribute properties area>中的第一个<attribute property>必须置于包含的<class symbol>的中间格中的最上面。每个后续的<attribute property>必须置于前一个<attribute property>的下面。

<behaviour properties area> ::=

{ { <behaviour property> <end> }* }***set***

<behaviour properties area>中的第一个<behaviour property>必须置于包含的<class symbol>的最下面格中的最上面。每个后续的<attribute property>必须置于前一个<attribute property>的下面。

<type reference properties> ::=

referenced <end>

<attribute property> ::=

<variable property>
| <field property>
| <signal parameter property>
| <interface variable property>

<attribute property>给出了变量或字段属性的部分规范，这些变量或字段在类型定义中进行定义，类型定义被类型引用所引用。<attribute property>的元素必须符合被引用类型定义中对应属性的要求。

<variable property> ::=

[<local> | <exported>] <variable name> [<sort>]

<local> ::=

<hyphen>

<exported> ::=

<plus sign>

<variable property>对应代理类型、过程或复合状态类型中的<variable definition>。<local>指明一个局部变量；<exported>指明一个输出变量。<variable name>和<sort>如果出现的话，它们必须与对应变量定义中的相同。

<field property> ::=

[<symbolic visibility>] <field name> [<sort>]

<symbolic visibility> ::=

<private>
| <public>
| <protected>

<private> ::=

<hyphen> | **private**

<public> ::=

<plus sign> | **public**

<protected> ::=

<number sign> | **protected**

<field property>对应数据类型中的<field>。<private> （<public>、<protected>）对应相应字段中的私有（公共、保护）<visibility>。<field name>和<sort>如果出现的话，它们必须与对应字段定义中的相同。

<signal parameter property> ::=
<sort>

<signal parameter property>对应信号定义中的信号参数。该类别必须对应相应信号定义的<signal definition item>中的<sort list>中的<sort>。<type reference properties>中的<signal parameter property>必须以被引用类型定义中对应属性相同的次序出现。

<interface variable property> ::=
<remote variable name> [<sort>]

<interface variable property>对应接口中的<interface variable definition>。<sort>必须与接口变量定义中的<sort>相同。

<behaviour property> ::=
{ [**operator**] <operation property> }
| { [**method**] <operation property> }
| { [**procedure**] <procedure property> }
| { [**signal**] <signal property> }
| { [**exception**] <exception property> }
| { [**timer**] <timer property> }
| { <interface use list> }

<behaviour property>给出了过程和操作属性的部分规范，这些过程和操作在类型引用（通过类型引用来引用）中定义，该规范必须符合相应类型定义中对应定义的要求。

<operation property> ::=
[<symbolic visibility>] <operation name>
<procedure signature>

<operation property>对应对象或值类型引用中的<operation definition>。<private> （<public>、<protected>）对应相应操作定义中的私有 （公共、保护）<visibility>。<procedure signature>中的<formal parameter>列表、<result>和<raises>如果出现的话，它们必须分别与相应操作定义中的<formal parameter>、<result>和<raises>相同。

<procedure property> ::=
[<local> | <exported>] <procedure name>
<procedure signature>

代理类型引用中的<procedure property>对应代理类型中的<procedure definition>。<local>指明一个局部过程；<exported>指明一个输出过程。<procedure signature>中的<formal parameter>列表、<result>和<raises>如果出现的话，它们必须分别与相应过程定义中的<procedure formal parameters>、<procedure result>和<raises>相同。

接口引用中的<procedure property>对应接口中的<interface procedure definition>。<local>不得出现在接口引用中。<procedure signature>如果出现的话，必须与相应接口过程定义中的相同。

<signal property> ::=
<signal name> [<sort list>]

代理类型引用中的<signal property>对应在代理类型中的某个输入中进行处理的信号。

接口引用中的信号属性对应<interface definition>中的某个<signal definition>中的<signal definition item>。<sort list>如果出现的话，必须与相应<signal definition item>中的相同。

<exception property> ::=
<exception name> [<sort list>]

类型引用中的<exception property>对应类型引用所引用之类型定义中的<exception definition item>。<sort list>如果出现的话，必须与相应<exception definition item>中的相同。

<timer property> ::=
<timer name> [<sort list>]

类型引用中的<timer property>对应该类型引用所引用之类型定义中的<timer definition item>。<sort list>如果出现的话，必须与相应<timer definition item>中的相同。

<interface use list>对应该类型引用所引用之接口定义的<interface use list>。每个<signal list item>必须对应被所引用接口定义的<interface use list>中的<signal list item>。

模型

用对应<referenced definition>替换每个引用。如果文本区域（如<agent text area>包含一个类型图的文本引用（即<agent type reference>、<composite state type reference>、<procedure reference>或<operation reference>），那么移去该引用，并在包含图（在包含文本区域的图中嵌套）的区域中插入所引用的图。如果文本区域包含类型定义的文本引用（即<procedure reference>、<operation reference>、<signal reference>、<data type reference>或<interface reference>），那么移去该引用，并在文本区域（在包含图引用的图中嵌套）中插入所引用的定义。

<name>之前不带<qualifier>的<type reference heading>是衍生语法，在其中用<qualifier>标识的实体是嵌套正文。

将类型引用（在其中，由<type reference heading>的<qualifier>标识的实体不同于嵌套的正文）移至通过限定词给出的正文中，并因此应用该正文的视见度规则。

同一正文中的多个类型引用如果指的是同一实体类，并有相同的限定词和相同的名称，那么这些类型引用等同于自该正文的一个类型引用，带有所有引用的所有<attribute property>和<behaviour property>元素。

如果类型引用的<type reference heading>的<qualifier>与嵌套正文相同，那么在缩减多个类型引用后，它将被所引用的类型替换，如第7节所述。

注3 — 类型引用模型（其中，用<type reference heading>的<qualifier>标识的实体与嵌套正文不同）意味着所引用的类型可以是直接在嵌套正文中的范围内的另一个不可见类型。

8.5 关联

关联表示两个实体类型之间的二元关系，没必要不同。关联旨在在包含类型引用的图或定义中，提供结构化的注释，来指出该关联所连接的类型的额外属性。本建议书没有定义这些属性的含义；也就是说，其含义可在其他建议书或标准或通用规范或通用谅解备忘录中定义。如果关联被删除，那么包含关联的SDL系统具有相同的含义和行为（如本建议书所定义）。

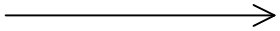
具体语法

<association area> ::=
<association symbol>
[***is associated with*** <association name>]
is connected to {<association end area> <association end area>} ***set***

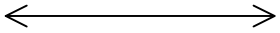
<association symbol> ::=
<association not bound symbol>
| <association end bound symbol>
| <association two ends bound symbol>
| <composition not bound symbol>
| <composition part end bound symbol>
| <composition composite end bound symbol>
| <composition two ends bound symbol>
| <aggregation not bound symbol>
| <aggregation part end bound symbol>
| <aggregation aggregate end bound symbol>
| <aggregation two ends bound symbol>

<association not bound symbol> ::=

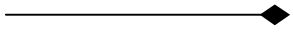
<association end bound symbol>::=



<association two ends bound symbol>::=

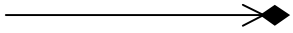


<composition not bound symbol> ::=

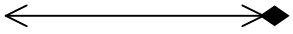


<composition part end bound symbol> ::=

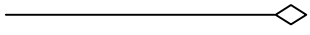
<composition composite end bound symbol> ::=



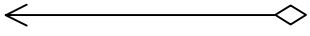
<composition two ends bound symbol> ::=



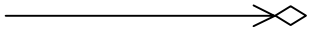
<aggregation not bound symbol> ::=



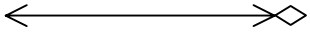
<aggregation part end bound symbol> ::=



<aggregation aggregate end bound symbol> ::=



<aggregation two ends bound symbol> ::=



<association end area> ::=
<linked type reference area> ***is associated with***
{ [<role name>] [<multiplicity>] [<ordering area>] [<symbolic visibility>] }***set***

<multiplicity> ::=
<range condition>

<ordering area> ::=
ordered

<linked type reference area> ::=
<agent type reference area>
| <data type reference area>
| <interface reference area>

<association symbol>允许链路代理类型、接口或数据类型。

如果一个<association end area>标识一个代理类型或一个接口，那么**protected**视见度不能用在<association area>的另一个<association end area>中。

如果两个不同的<association area>标识相同的类型，那么在相对本通用类型的<association end area>中，<role name>（如果给出的话）必须不同。

不得有包含复合的<association area>集，这样，类型可以直接或间接地通过复合链接回自身。

如果一个<composition not bound symbol>、<composition part end bound symbol>、<composition composite end bound symbol>或<composition two ends bound symbol>的复合端（带菱形的一端）连接至一个标识数据类型或接口的<linked type reference area>，那么另一端的<association end area>必须连接至分别标识数据类型或接口的<linked type reference area>。

<multiplicity>中<range condition>的基础类别必须是预定义的自然数类别。

语义

关联以某种方式连接两个实体类型，在本建议书中没有进一步的定义。

9 代理

代理定义用于定义一个（任意大）代理集。代理通过变量、过程、状态机（以显性或隐性的复合状态类型给出）和所含代理的集合来表征。

有两个种类的代理：功能块和过程。系统是最外层的功能块。一个功能块的状态机与其所含的代理并发地进行解释，而一个过程的状态机则与其所含的代理交替地进行解释。

抽象语法

Agent-definition :: *Agent-name*
Number-of-instances
Agent-type-identifier
Number-of-instances :: *Initial-number* [*Maximum-number*]
Initial-number = *Nat*
Maximum-number = *Nat*

具体语法

<agent diagram> ::=
{ <system diagram> | <block diagram> | <process diagram> }
[***is associated with*** <package use area>]

<agent instantiation> ::=
[<number of instances>]
<agent additional heading>

<agent additional heading> ::=
[<specialization>] [<agent formal parameters>]

<agent formal parameters> ::=
(<parameters of sort> {**,** <parameters of sort>}* **)**

<parameters of sort> ::=
<variable name> {**,** <variable name>}* <sort>

<number of instances> ::=
([<initial number>] [**,** [<maximum number>]] **)**

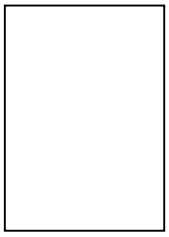
<initial number> ::=
<Natural simple expression>

<maximum number> ::=
<Natural simple expression>

<agent structure area> ::=
{ {<agent text area>}*
{<entity in agent diagram>}*
{ <interaction area> | <agent body area> } }***set***

<agent body area> ::=
{ [[<on exception association area>] <start area>]
{ <state area> | <exception handler area> | <in connector area> }* }***set***

<frame symbol> ::=



<package use area>必须置于<system diagram>的<frame symbol>、<block diagram>或<process diagram>的最上面。

<agent text area> ::=

<text symbol>
contains {
[<valid input signal set>]
{ <signal definition>
| <signal reference>
| <signal list definition>
| <variable definition>
| <remote procedure definition>
| <remote variable definition>
| <data definition>
| <data type reference>
| <timer definition>
| <interface reference>
| <macro definition>
| <exception definition>
| <procedure definition>
| <procedure reference>
| <select definition>
| <agent type reference>
| <agent reference> }* }

<entity in agent diagram>::=
<agent type diagram>
| <agent type reference area>
| <composite state area>
| <composite state type diagram>
| <composite state type reference area>
| <procedure diagram>
| <procedure reference area>
| <data type reference area>
| <signal reference area>
| <association area>

<interaction area> ::=
{{ <agent area>
| <create line area>
| <channel definition area>
| <state partition area> }+ }***set***

<agent area> ::=
<agent reference area>
| <agent diagram>
| <typebased agent definition>
| <inherited agent definition>

<create line area> ::=
<create line symbol>
is connected to {<create line endpoint area> <create line endpoint area>}

<create line endpoint area> ::=
<agent area> | <agent type area> | <state partition area>

<agent type area> ::=
<agent type reference area>
| <agent type diagram>

<create line symbol> ::=
<dependency symbol>

<valid input signal set> ::=
signalset [<signal list>] <end>

以下内容通常对代理是合法的。系统、功能块和过程的特殊属性在有关这些概念的单独子句中进行处理。

在*Number-of-instances*中包含的实例最初数和实例最大数源自＜number of instances＞。如果省略了＜initial number＞，那么＜initial number＞为1。如果省略了＜maximum number＞，那么＜maximum number＞为无穷大。

衍生中使用的＜number of instances＞如下所述：

a） 如果没有代理的＜agent reference＞，那么使用＜agent diagram＞或 ＜typebased agent definition＞中的＜number of instances＞。如果不包含＜number of instances＞，那么使用同时省略＜initial number＞和＜maximum number＞的＜number of instances＞。

b） 如果＜agent diagram＞中的＜number of instances＞和＜agent reference＞或＜agent reference area＞中的＜number of instances＞都省略，那么使用同时省略＜initial number＞ 和＜maximum number＞的＜number of instances＞。

c） 如果＜agent diagram＞中的＜number of instances＞或者＜agent reference＞或＜agent reference area＞中的＜number of instances＞有其中一个省略，那么使用出现的那个＜number of instances＞。

d） 如果既规定了＜agent diagram＞中的＜number of instances＞，又规定了＜agent reference＞或＜agent reference area＞中的＜number of instances＞，那么这两个＜number of instances＞在词汇上是等同的，并使用该＜number of instances＞。

实例的＜initial number＞必须小于或等于＜maximum number＞，并且＜maximum number＞必须大于0。

在＜agent instantiation＞中，如果有＜agent formal parameters＞那么一定会有＜number of instances＞，即使在＜initial number＞和＜maximum number＞同时被省略的情况下也如此。

当且仅当直接嵌套的＜agent structure area＞包含＜interaction area＞时，才允许＜agent text area＞包含＜agent reference＞。

在＜agent diagram＞中，＜gate on diagram＞必须置于图框的外面。

隐含代理类型的*抽象句法*（见*模型*）中的*Agent-definition*-**set**对应＜agent area＞。

隐含代理类型的*抽象句法*中的*Channel-definition*-**set**对应＜channel definition area＞。

＜create line symbol＞上的箭头指明代理或代理类型的＜agent area＞或＜agent type area＞，在其上执行创建动作。＜create line symbol＞是可选的，但如果使用了它，那么在＜create line symbol＞起始端的代理（或代理类型或状态机）中，＜create line symbol＞的箭头尾端必须有一个代理（或代理类型）创建请求。

注1 — 该规则可以在＜transition option area＞转化之前或之后独立应用。

＜interaction area＞的＜state partition area＞确定代理的状态机（复合状态），它可能会直接作为代理图给出，或以指向一个状态定义的引用给出。

＜start area＞只能在代理类型图中被省略。

如果有代理＜agent reference area＞，那么与该＜agent reference area＞相关的＜gate property area＞对应与＜agent diagram＞相关的＜gate on diagram＞。与＜agent reference area＞相关的＜gate property area＞中不能包含＜signal list item＞，＜signal list item＞不包含在与＜agent diagram＞相关的＜gate on diagram＞中。如果有代理＜agent reference area＞，那么对应规则适用。

连接到＜agent reference area＞的＜package dependency area＞必须符合所引用图的＜package use area＞的要求。

有关＜valid input signal set＞的使用和语法在第9节中定义。

语义

*Agent-definition*有一个名称，它可用于与**system**、**block**或**process**（根据代理种类而定）相结合的限定词中。

一个代理定义用于定义一个代理集。同一代理集的若干个实例可以同时存在，并被异步和并行地解释，或相互之间交叉解释，以及用系统中其他代理集的实例来解释。

*Number-of-instances*中的第一个值代表当系统或包含的实体创建时（初始实例）的代理集的实例个数，第二个值代表该代理集的并发实例的最大个数。

Agent-definition-**set**中的*Agent-definition*的行为依赖于包含的*Agent-definition*是否是一个功能块或过程，并因此分别为功能块和过程做出定义。

一个代理实例有一个通信扩展的有限状态机，该有限状态机以其显性或隐性状态机定义来定义。每当状态机处于一个状态、输入某给定信号的情况下，它都将执行某个动作序列，表示为一个转换。转换的完成会导致代理实例的状态机处于等待另一个状态，另一个状态不必与第一个状态不同。

当对代理进行解释时，将创建它包含的最初代理。这些最初代理的有限状态机、该代理的有限状态机及其环境之间的信号通信，只有在所有的最初代理都已创建的情况下才开始。创建代理所花的时间可能很可观也可能不可观。最初代理的形式参数没有相关的数据项（它们是“未定义的”）。

代理实例从包含代理创建时就存在，或它们可以通过正在解释的代理的创建请求动作进行创建；当对其开始动作进行解释时，其解释就开始了；可以通过执行停止动作是它们不再存在。

当一个代理的状态机解释一个停止时，如果该代理是一个并发的存储器，那么它将继续处理隐含的远程过程调用来间接访问全局变量。在所有包含的代理终止之前，这类代理的状态机仍保持在这种“停止状态”上，在这之后代理终结。处于停止状态时，除为每个全局变量引入的隐含设置和获取远程过程调用外，代理将不接受任何其他激励（如果有的话）。代理终结后，其pid不再合法。

如果一个代理没有显性或隐性状态机，那么一旦所有的最初包含代理创建，该代理就将进入一个停止状态。因此，带有包含最初实例而不包含状态机的代理一创建就不再存在。

代理实例接收的信号表示为输入信号，从代理实例传送过来的信号表示为输出信号。一个代理的<valid input signal set>定义了其状态机的合法输入信号集。

调用和服务远程调用过程以及访问远程变量，也对应信号的交换（分别见第10.5和10.6节）。

只有代理实例处于某种状态时，信号可以被代理实例的状态机使用。完整的合法输入信号集是下列的联合：

a） 通向代理状态机的所有信道或通道内的信号集；

b） 代理的<valid input signal set>；

c） 代理状态机的<valid input signal set>；

d） 在第10.5、10.6节中介绍的隐含输入信号；以及

e） 计时器信号。

准确地，只有一个输入端口与每个代理实例的有限状态机相关联。假如信号在连接至其状态机的（显性或隐性）信道上出现，那么送到存储器代理的信号将被传送到代理的输入端口中。只在<valid input signal set>中出现的信号不得用于外部通信。它们用于同一实例集内的实例间的通信。

一个代理的有限状态机或是在某个状态等待，或者是活动的，在执行某个转换。对每个状态，都存在一个保存信号集（见第11.7节）。当在某种状态下等待时，标识符不在保存信号集中的第一个输入信号会从队列中取出并供代理使用。一个转换也可以被初始化为一个独立于队列中出现的任何信号的自发转换。

输入端口可以保留任意个数的输入信号，这样，若干个输入信号可以排队，供代理实例的有限状态机使用。这组保留的信号根据其到达时间在队列中进行排序。如果两个或多个信号从不同路径“同时”到达，那么它们可以任意排序。

当代理创建时，其有限状态机给出一个空的输入端口，并创建代理的局部变量。

当一个存储器代理实例创建时，所包含代理集的最初代理被创建。如果存储器是通过<create body>创建的，那么所包含代理的**parent**（见下面模型）接收存储器的pid。形式参数是可变的，它们或者在系统创建时创建（但没有实际参数传给它们，因此它们是“未定义的”），或者当代理实例被动态创建时创建。

代理定义指的是代理同一范围内的接口定义（见第12.1.2节）。该接口隐含定义的pid类别用*Agent-name*表示，并在代理定义的同一范围单元内可见。

注*2* — 由于每个代理都有一个名称相同的隐含定义的接口，因此代理的名称必须与每个显性定义的接口名称不同，并且与同一范围内定义的每个代理类型（它们也有隐含的接口）名称也不同；否则，会出现名称冲突。

一个代理集的完整输出集与该代理集类型的完整输出集相同。

模型

*<agent diagram>*具有一个隐含的匿名代理类型，它定义了该代理的各属性。

带<agent body area>的代理是只有一个状态机而没有所包含代理的代理的一种缩写形式。该状态机通过用一个复合状态定义替换<agent body area>来获得。该复合状态定义与代理同名，其*State-transition-graph*用<agent body area>表示。

为特殊化的代理是定义一个隐含代理类型的一种缩写形式，是该类型的一个基于类型的代理。

在所有的代理实例中，都声明了代理pid类别（不是基于某种代理类型的代理）或代理类型pid类别（基于类型的代理）的四个匿名变量，在下面，分别称为**self**、**parent**、**offspring**和**sender**。它们为下列实例给出了结果：

a） 代理实例（self）；

b） 创建代理实例（parent）；

c） 由代理实例创建的最新代理实例（offspring）；

d） 最后一个输入信号已使用的代理实例（sender）（见第11.3节）。

这些匿名变量通过pid表达式访问，在第12.3.4.3节中有更多的解释。

对包含实例创建时创建的所有代理实例，**parent**被初始化为Null。

对所有最新创建的代理实例，**sender**和**offspring**被初始化为Null。

9.1 系统

系统是最外层的代理，并具有*Agent-kind* **SYSTEM**。它用<system diagram>定义。代理的语义适用于功能块，还加上本子句中提供的其他内容。

抽象语法

带*Agent-kind* **SYSTEM**的*Agent*不得包含在任何其他*Agent*中。它必须至少包含一个*Agent-definition*或一个显性或隐性的*State-machine-definition*。

所有在环境接口中或系统所包含代理之间（包括系统自身）使用的信号、信道、数据类型和共型都包含在系统的*Agent-definition*中。

实例的*Initial-number*为1，实例的*Maximum-number*为1。

注 — 不能规定<number of instances>。

具体语法

<system diagram> ::=

<frame symbol> ***contains*** {<system heading> <agent structure area> }
is connected to { {<gate on diagram>}* }***set***
[***is associated with*** <package use area>]

<system heading>::=

system <system name> <agent additional heading>

<system diagram>中的<agent additional heading>不能包含<agent formal parameters>。

<system diagram>中的<gate on diagram>不能包含<channel identifier>。

语义

带*Agent-kind* **SYSTEM**的*Agent-definition*是系统规范或描述的SDL表示。系统是最外层的功能块。这意味着系统内的代理是相互间以及与该系统可能的状态机间并发解释的功能块和过程。

一个系统通过系统界限从其环境中分离出来，它包含一个代理集。系统和环境之间、系统内代理之间的通信可通过使用信号、远程过程和远程变量发生。在系统内，这些通信手段在显性或隐性信道上进行传送。通道将所含的代理连接至另一个代理上或连接至系统边界上。

一个系统实例是系统类型的一个实例化，通过带*Agent-kind* **SYSTEM**的*Agent-definition*来标识。系统实例的解释通过一个抽象SDL机来执行，从而给出SDL概念的语义。解释一个系统实例是为了：

a） 初始化系统时间；

b） 解释所包含代理及其连接的信道；以及

c） 解释系统的可选状态机。

9.2 功能块

功能块是带*Agent-kind* **BLOCK**的代理。因此，代理的语义适用于功能块，还加上本子句中提供的其他内容。功能块通过<block diagram>定义。

功能块实例内包含的实例相互之间以及与包含功能块实例的状态机之间并发地和异步地进行解释。功能块内不同包含实例之间的所有通信显性或隐性地用信号交换（如远程调用）来异步执行。

具体语法

<block diagram> ::=

<frame symbol> ***contains*** {<block heading> <agent structure area> }
is connected to { {<gate on diagram> | <external channel identifiers>}* }***set***
[***is associated with*** <package use area>]

<block heading> ::=

block [<qualifier>] <block name> <agent instantiation>

<gate on diagram>标识与信道连接点相关的通道。在<agent structure area>是一个<interaction area>的情况下，<gate on diagram>置于紧靠<frame symbol>外内部通道的终点上。

<external channel identifiers>标识连接至<block diagram>内信道的外部信道。它置于<frame symbol>外面，紧靠<frame symbol>内部通道的终点上。

语义

功能块定义是一个代理定义，它为状态机（可能不带行为）以及零个或多个进程或功能块定义用于定义一个存储器。

一个功能块实例是功能块类型的一个实例化，用带*Agent-kind* **BLOCK**的*Agent-definition*标识。解释功能块实例是为了：

a） 解释所含代理及其连接的信道；

b） 解释功能块的状态机（如果有的话）。

在带有限状态机的功能块中，有限状态机作为功能块（及其所含的代理）的一部分来创建，并且它与功能块内的代理并发地进行解释。

带变量定义但没有状态机的功能块具有一个相关的隐性状态机，它与功能块内的代理并发地进行解释。

从功能块内所含代理到功能块变量的访问通过两个隐性定义的远程过程（设置和获取与变量相关的数据项）来实现。这些过程由功能块的状态机提供。

模型

带状态机和变量的功能块b建模方法如下：保持功能块b（不带变量），并将状态实体和变量转化成功能块b中的一个独立状态机（sm）。对b中的每个变量v，该状态机都有变量v和两个输出过程set_v（带有一个v类别的**in**参数）和get_v（带有一个v类别的返回类型）。自嵌套定义的v的每个指配都将被转化成set_v的一个远程调用。嵌套定义中表达式中v的每次事件都将被转化成get_v的一个远程调用。这些事件也适用于功能块b中定义的过程中的各事件，它们被转化成调用代理的本地过程。

只带变量和/或过程的功能块b其转化过程同上，其产生的状态机的图只有一个状态，来输入生成的集合，并获取过程。

对连接至状态机的信道进行转化，以便将其连接至sm。

该转化在类型和正文参数转化后进行。

9.3 进程

进程是一个带*Agent-kind* **PROCESS**的代理。因此，代理的语义适用于进程，还加上本子句中提供的其他内容。进程通过一个<process diagram>来定义。

进程用于将共享的数据引入到一个规范中，以使包含进程的变量能被对象使用，或使用对象。进程内的所有实例都可访问进程的局部变量。

尽管有了进程内的数据共享，为了实现安全通信，所有实例使用交互语义来解释。这意味着对进程内的任意两个实例，没有两个转换是并行解释的，并且一个实例内的转换解释不可被另一个实例中断。例如，但一个实例正在等待一个远程过程调用的返回时，它就处于某个状态；因此，可以对另一个实例进行解释。

抽象语法

带*Agent-kind* **PROCESS**的*Agent-definition*必须或者至少包含一个*Agent-definition*，或者有一个显性或隐性的*State-machine-definition*。

所有带*Agent-kind* **PROCESS**的*Agent-definition*的所含*Agent-definitions*都应该有*Agent-kind* **PROCESS**。

具体语法

<process diagram> ::=
<frame symbol> ***contains*** {<process heading> <agent structure area> }
is connected to { {<gate on diagram> | <external channel identifiers>}* }***set***
[***is associated with*** <package use area>]

<process heading> ::=
process [<qualifier>] <process name> <agent instantiation>

<gate on diagram>标识一个连接点指向信道的通道。在<agent structure area>是一个<interaction area>的情况下，<gate on diagram>置于<frame symbol>外紧靠内部信道的终点位置处。

<external channel identifiers>标识连接至<process diagram>中信道的外部信道。它置于<frame symbol>外，紧靠<frame symbol>处内部信道的终点位置处。

语义

一个进程定义是一个代理定义，它为状态机（可能不带行为）和零个或多个进程定义用于定义一个存储器。一个进程实例是一个用带*Agent-kind* **PROCESS**的*Agent-definition*标识的进程类型的实例化。

带所含进程实例集的进程的一个实例通过以下方式进行解释，即在所含进程实例集中的实例与其他实例之间交替地进行解释，以及与包含进程实例的状态机（如果有的话）之间交替地进行解释。交替解释意味着在某一时刻，在交互正文内只有一个实例可以解释转换，并且涉及的过程实例的转换解释一旦开始执行，则一直持续到达到某个（显性的或隐性的）状态为止，或直至该过程实例终结。状态可以是一个由转化引入的隐性状态（例如，因一个远程过程调用）。

带有变量定义和被包含进程但没有显性状态机的进程，有一个相关的隐性状态机，它与被包含进程交替地进行解释。

注 — 状态聚合也存在交替解释。不过，一个进程的各交替进程各有其自身的输入端口和其自身的**self**、**parent**、**offspring**和**sender**。在状态聚合情况下，只有一个输入端口和一组**self**、**parent**、**offspring** 和**sender**属于存储器代理。

9.4 代理和复合状态参考

具体语法

<agent reference> ::=
<block reference>
| <process reference>

<agent reference area> ::=
{ <system reference area>
| <block reference area>
| <process reference area> }
[***is connected to*** { <package dependency area>+ }***set***]

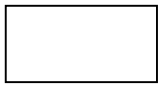
<system reference area> ::=
<block symbol> ***contains***
{ **system** <system name> }

<system reference area>只能作为<specification area>的一部分来使用。

<block reference> ::=
block <block name> [<number of instances>] **referenced** <end>

<block reference area> ::=
<block symbol> ***contains***
{ { <block name> [<number of instances>] } {<gate>*}***set*** }
is connected to { <gate property area>* }***set***

<block symbol> ::=



<gate>置于<block symbol>的边缘附近，并与指向信道的连接点相关。它们置于紧靠<block symbol>上的信道终点的位置处。

<process reference> ::=
process <process name> [<number of instances>] **referenced** <end>

<process reference area> ::=
<process symbol> ***contains***
{ { <process name> [<number of instances>] } {<gate>*}***set*** }
is connected to { {<gate property area>*}***set*** }

<process symbol> ::=

<gate>置于<process symbol>的边缘附近，并与指向信道的连接点相关。它们置于紧靠<process symbol>上的信道终点的位置处。

<composite state reference area> ::=
<state symbol> ***contains*** { <state name> { <gate>*}***set*** }

模型

每个引用都将由对应的<referenced definition>替换。如果文本区域（如<agent text area>）包含一个<agent reference>，那么移去该引用，并在包含文本区域的图内嵌套的、包含图的区域中插入引用图。

9.5 过程

过程通过过程定义来定义。过程通过标识过程定义的过程调用来调用。参数与过程调用相关。哪个变量将受过程解释影响由参数传递机制控制。过程调用可以是动作或表达式（仅为值返回过程）。

抽象语法

Procedure-definition	::	*Procedure-name* *Procedure-formal-parameter** [*Result*] [*Procedure-identifier*] *Data-type-definition-**set*** *Syntype-definition-**set*** *Variable-definition-**set*** *Composite-state-type-definition-**set*** *Procedure-definition-**set*** *Procedure-graph*
Procedure-name	=	*Name*
Procedure-formal-parameter	=	*In-parameter*
	|	*Inout-parameter*
	|	*Out-parameter*
In-parameter	::	*Parameter*
Inout-parameter	::	*Parameter*
Out-parameter	::	*Parameter*
Parameter	::	*Variable-name* *Sort-reference-identifier*
Result	::	*Sort-reference-identifier*
Procedure-graph	::	[*On-exception*] [*Procedure-start-node*] *State-node-**set*** *Free-action-**set*** *Exception-handler-node-**set***
Procedure-start-node	::	[*On-exception*] *Transition*
Procedure-identifier	=	*Identifier*

如果一个*Procedure-definition*包含*Result*，那么它对应一个值返回过程。

在一个SDL规范中，**所有潜在初始化的过程都必须有一个**Procedure-start-node。

具体语法

<procedure definition> ::=
<external procedure definition>
| {<package use clause>}*
<procedure heading>
[<end> <entity in procedure>+]
[<virtuality>] [<comment body>] <left curly bracket>
<statement list>
<right curly bracket>

<procedure definition>中在<left curly bracket> <statement list>之前的可选<virtuality>适用于过程的开始转换，在这种情况下，它是语句列表。

<procedure definition>中的<variable definition>不能包含**exported**<variable name>（见第12.3.1节）。

<procedure diagram> ::=
<frame symbol> ***contains*** {
<procedure heading>
{ <procedure text area>*
<procedure area>*
<procedure body area> }***set*** }
[***is associated with*** <package use area>]

<procedure heading> ::=
<procedure preamble>
procedure [<qualifier>] <procedure name>
[<formal context parameters>] [<virtuality constraint>]
[<specialization>]
[<procedure formal parameters>]
[<procedure result>] [<raises>]

<procedure preamble> ::=

<type preamble> [**exported** [**as** <remote procedure identifier>]]

<procedure formal parameters> ::=
(<formal variable parameters> {**,** <formal variable parameters> }* **)**

<formal variable parameters> ::=
<parameter kind> <parameters of sort>

<parameter kind> ::=
[**in/out** | **in** | **out**]

<procedure result> ::=
<result sign> [<variable name>] <sort>

<raises> ::=
raise <exception identifier> {**,** <exception identifier>}*

<procedure area> ::=
<procedure diagram>
| <procedure reference area>
| <composite state type diagram>
| <composite state type reference area>

<entity in procedure> ::=
<variable definition>
| <data definition>
| <data type reference>
| <procedure reference>
| <procedure definition>
| <exception definition>
| <select definition>
| <macro definition>

<procedure text area> ::=
<text symbol> ***contains***
{ <variable definition>
| <data definition>
| <data type reference>
| <procedure reference>
| <procedure definition>
| <exception definition>
| <select definition>
| <macro definition> }*

<procedure signature> ::=
[**(**<formal parameter> { **,** <formal parameter> }* **)**] [<result>] [<raises>]

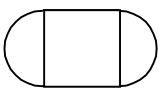
<external procedure definition> ::=
procedure <procedure name> <procedure signature> **external** <end>

不能在<type expression>、<formal context parameter>或<procedure constraint>中提到外部过程。

<procedure body area> ::=
[<on exception association area>] [<procedure start area>]
{<state area> | <exception handler area> | <in connector area> }*

<procedure start area> ::=
<procedure start symbol>
contains { [<virtuality>] }
[***is connected to*** <on exception association area>]
is followed by <transition area>

<procedure start symbol> ::=



<package use area>必须置于<frame symbol>的最上面。

<procedure body area>的<on exception association area>标识与整个图相关的异常句柄。起始端不得连接至任何符号。

一个输出过程不能有形式正文参数，其嵌套范围必须是一个代理类型或代理定义。

如果有**exported** 的话，**exported**将被过程的任何子类型继承。一个虚拟输出过程必须在所有重新定义中包含**exported**。包括虚拟过程的虚拟类型在第8.3.2节中描述。重新定义中的可选**as**子句必须表示与父类型相同的<remote procedure identifier>。如果在重新定义中省略**as**子句，那么指的是父类型的<remote procedure identifier>。

一个代理中的两个输出过程不能提到相同的<remote procedure identifier>。

在对应的句柄子句没有激活任何异常句柄时（即它未被处理），如果在过程中引起一个异常，那么<raises>必须提到该异常。如果在产生该异常的过程内存在一个潜在的控制流，那么异常被看做是未处理的，并且没有在该控制流中激活的异常句柄来处理异常。

如果在过程引用中给出了**exported**，那么被引用过程必须是一个输出过程；如果还给出了<remote procedure identifier>，那么该过程必须标识同一远程过程定义。

语义

过程是给项的汇集起个名称的手段，它通过一个单一的引用来表示该汇集。过程规则为汇集项的选择方式施加了约束，并限制了在该过程中定义的变量名称的范围。

根据第10.5节中的模型，<procedure preamble>中的**exported**意味着过程可以作为一个远程过程来调用。

过程变量是在过程内不能输出的局部变量。它在过程开始节点被解释时创建，当过程图的返回节点被解释时，它不再存在。

一个*Call-node*（以<procedure call area>或<procedure call area>表示，分别见第11.13和11.14节）、一个*Value-returning-call-node*（以<value returning procedure call>表示，见第12.3.5节），或一个*Operation-application*（以<operation application>表示，见第12.2.7节）的解释会引起过程实例的创建，该解释以下列方式开始：

a） 为每个*In-parameter*创建一个局部变量，其*Name*和*Sort*与*In-parameter*相同。如果给出了实际参数，那么该变量通过解释变量与由相应实际参数给出的表达式之间的指配，来与表达式的结果相关联。否则，变量得不到相关的数据项；即变为“未定义的”。

b） 为每个*Out-parameter*创建一个局部变量，其*Name*和*Sort*与*Out-parameter*相同。变量得不到数据项；即变为“未定义的”。

c） 为*Procedure-definition*中的每个*Variable-definition*创建一个局部变量。

d） 每个*Inout-parameter*表示一个以第11.13.3节中的实际参数表达式给出的变量。在解释*Procedure-graph*的整个过程中，当提到与该变量相关的数据项时，或当给该变量指配一个新的数据项时，使用所含的*Variable-name*。

e） 对*Procedure-start-node*中包含的*Transition*进行解释。

f） 在对*Procedure-graph*中包含的*Return-node*进行解释之前，为*Out-parameter*提供相应局部变量的数据项。

过程图节点的解释方式与代理等同节点的解释方式相同；也就是说，过程与嵌套代理具有相同的完全合法输入信号集，与直接或间接调用它的嵌套代理的实例具有相同的输入端口。

一个外部过程是一个过程，其<procedure body area>不包括在SDL描述中（见第13节）。

模型

不带显性<parameter kind>的形式参数有隐性的<parameter kind>**in**。

当<variable name>出现在<procedure result>中时，不带<expression>的过程图内的所有<return area>都将被一个包含<variable name>的<return area>所替换，作为<expression>。

带<variable name>的<procedure result>是带<variable name>的<variable definition>和<variables of sort>中的<sort>的衍生语法。如果存在某个涉及<variable name>的<variable definition>，那么不会再增加更多的<variable definition>。

包含<virtuality>的<procedure start area>或跟在<virtuality>后面的<procedure definition>中的<statement list>被称为虚拟过程开始。对虚拟过程开始在第8.3.3节中有更多的描述。

<procedure definition>（而不是<external procedure definition>）是<procedure diagram>的衍生语法，有相同的<procedure preamble>和一个带有相同<virtuality>的单一<start area>。<start area>的<transition area>由包含<procedure definition>的<statement list>的<task area>组成，后跟没有标签的<return area>。<procedure definition>的<entity in procedure>被插入到<procedure diagram>的<procedure text area>中。

该转换在<compound statement>转化后发生。

10 通信

10.1 信道

抽象语法

Channel-definition	::	*Channel-name* [***NODELAY***] *Channel-path-**set***
Channel-path	::	*Originating-gate* *Destination-gate* *Signal-identifier-**set***
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*
Gate-identifier	=	*Identifier*
Agent-identifier	=	*Identifier*
Channel-name	=	*Name*

Channel-path-**set**至少包含一个*Channel-path*，并不能超过两个。当有两条路径时，该信道是双向的，并且每个*Channel-path*的*Originating-gate*必须与另一个*Channel-path*的*Destination-gate*相同。

如果*Originating-gate*和*Destination-gate*是同一个代理，那么该信道必须是双向的（*Channel-path-**set***中必须只能有一个元素）。

*Originating-gate*或*Destination-gate*必须在信道定义的抽象句法中的同一个范围单元中进行定义。

NODELAY表示信道没有延迟。

允许信道互相连接至双向通道的两个方向上。

每个通道和信道在相同方向的信号列表中必须至少有一个共同元素。

具体语法

<channel definition area> ::=
<channel symbol>
is associated with
{ [<channel name>] { [<signal list area>] [<signal list area>] }***set*** }
is connected to {
{ <agent area> | <state partition area> | <gate on diagram> }
{ <agent area> | <state partition area> | <gate on diagram> }***set***

如果<channel symbol>连接至是一个<typebased agent definition>的<agent area>，那么在连接至代理符号的信道附近必须放置有<typebased agent definition>中的<gate>。该<gate>表示*Destination-gate*或*Originating-gate*，另一个通道由该信道的另一端决定。

当<channel symbol>连接至<state partition area>时，<state partition area>表示直接包含信道定义的代理的状态机。如果<state partition area>是一个<typebased state partition definition>，那么在连接至状态划分符号的信道附近必须放置有<typebased state partition definition>中的<gate>。该<gate>表示*Destination-gate*或*Originating-gate*，另一个通道由该信道的另一端决定。

对直接连接至<agent area>或<state partition area>的<channel symbol>末端，如果该代理或状态机在<external channel identifiers>中包含该信道的<channel identifier>，那么该信道被连接至由<external channel identifiers>引入的隐含通道上。否则，在没有<external channel identifiers>匹配的情况下，会有代理或状态上的一个隐含通道连接至<channel definition area>。该通道获得各自<channel definition area>中的<signal list>，作为其相应的通道约束。该信道连接至该通道。这一<gate>表示*Destination-gate*或*Originating-gate*，另一个通道由该信道的另一端决定。

对直接连接至<gate on diagram>的<channel symbol>末端，这表示*Destination-gate*或*Originating-gate*，另一个通道由该信道的另一端决定。

<channel symbol> ::=

<delaying channel symbol 1>
| <delaying channel symbol 2>
| <nondelaying channel symbol 1>
| <nondelaying channel symbol 2>

<delaying channel symbol 1> ::=

<delaying channel symbol 2> ::=

<nondelaying channel symbol 1> ::=

<nondelaying channel symbol 2> ::=

对<channel symbol>上的每个箭头，至多只能有一个<signal list area>。每个<signal list area>必须明确地、足够地靠近其中一个箭头。该箭头指明其所关联信号列表的信道路径方向。

<nondelaying channel symbol 1>和<nondelaying channel symbol 2>的箭头置于信道末端，并指明该信道没有延迟。

语义

*Channel-definition*表示信号（包括通过远程过程和远程变量表示的隐性信号，见第10.5和10.6节）的传输路径。一个信道可被视为两个代理间或代理与其环境间的一个或两个独立的双向信道路径。信道还可以将代理的状态机（复合状态）与环境以及与所含的代理相连。

*Channel-definition*中的每个*Channel-path*中的*Signal-identifier*-**set**包含可能在该*Channel-path*上进行传送的信号。

通过信道传送的信号被传送到目的端点。

信号在一个信道的目的端点上出现的次序与它们在其源处出现的次序相同。如果两个或多个信号同时提交给信道，那么其次序可以是任意的。

带延迟的信道可以延迟通过信道传送的信号。这意味着一个先入先出（FIFO）的延迟队列与信道中的每个方向相关。当信号提交给信道时，它被放入延迟队列中。经过一个不确定的且非固定的时间间隔后，队列中的第一个信号实例被释放，并传给连接至信道的其中一个端点上。

两个相同端点之间可能存在若干个信道。同一信号类型可以在不同信道上传送。

当某个信号实例被送到同一代理实例集的某个实例时，*Output-node*的解释或者指信号直接放到目的代理的输入端口中，或者指该信号经由连接该代理实例集至其自身的、没有延迟的信道进行发送。

信道上的远程过程或远程变量被称为来自输入器的输出以及到输出器的输入。

模型

如果<channel definition area>省略了<channel name>，那么该信道是隐含地、惟一地进行命名的。

两个端点同时都是一个<typebased agent definition>的通道的信道，代表从该集合中每个代理到集合中所有代理（包括起始代理）的单独信道。连接集合中某个代理到其自身的任何结果双向信道将被分为两个单向信道。

如果某代理或代理类型包含不被显性信道连接的显性或隐性通道，那么隐性信道按以下三种转化形式进行衍生，这些转化必须在第11.13.2节中基于类型的创建的转化已经进行之后才可以进行。

转化1：

在代理或代理类型内部的实例集之间以及实例集与代理状态机之间，插入信道；

转化2：

从代理或代理类型上的某通道到代理或代理类型内部的实例集上的通道以及从代理或代理类型上的某通道到代理状态机上的通道，插入信道。

转化3：

从代理或代理类型内部的实例集上的通道以及从代理状态机上的通道到代理或代理类型上的通道，插入信道。

这些转化将在下面详细描述。它们以给定的次序应用。

在转化中，一个信号列表元素（接口、信号、远程过程或远程变量）与另一个信号列表元素相匹配的条件是：

a） 二者指的都是同一接口、信号、远程过程或远程变量；或者

b） 第一个是指一个信号或远程过程或远程变量，第二个是指一个接口，并且该接口包括该信号或远程过程或远程变量；或者

c） 二者指的都是接口，并且第二个信号列表元素继承了第一个信号列表元素。

转化1： 在代理或代理类型内部的实体之间插入隐性信道

a） 如果与代理（或代理类型）中实例通道相关的输出信号列表的某个元素与同一代理（或代理类型，分别对应）中另一个实例通道相关的输入信号列表的某个元素相匹配；以及

b） 这些通道都没有与之相连的显性信道，

那么

a） 如果两个通道之间不存在显性信道，那么会创建一个从输出元素的通道到输入元素的通道的单向隐性信道。如果该信道在某过程（或过程类型）内，那么它是非延迟的，否则它是延迟的；以及

b） 元素被添加进隐性信道的信号列表中。

转化2： 从代理或代理类型上的通道插入隐性信道

a） 如果与代理（或代理类型）外部的通道相关的输入信号列表的某个元素与代理（或代理类型，分别对应）中某实例的通道相关的输入信号列表的某个元素相匹配；以及

b） 如果代理（或代理类型，分别对应）内部没有显性信道连接至代理（或代理类型，分别对应）外部的通道，并且没有显性信道连接至代理（或代理类型，分别对应）内部实例的通道，

那么

a） 如果两个通道之间不存在隐性信道，那么会创建一个从代理（或代理类型，分别对应）外部的通道到代理（或代理类型，分别对应）内部实例的通道的单向隐性信道。如果信道在某过程（或过程类型）之内，那么它是非延迟的，否则它是延迟的；以及

b） 元素被添加进隐性信道的信号列表中。

转化3： 从实例上的通道插入隐性信道

以下内容适用于从代理或代理类型内部实例集上的通道到代理或代理类型上的通道的隐性信道插入：

a） 如果与代理（或代理类型）外部通道相关的输出信号列表的某个元素与代理（或代理类型，分别对应）中某实例的通道相关的输出信号列表的某个元素相匹配；以及

b） 如果没有显性信道连接至代理（或代理类型，分别对应）外部的通道，并且没有显性信道连接至代理（或代理类型，分别对应）内部实例的通道，

那么

a） 如果在代理（或代理类型，分别对应）外部通道的方向上，两个通道之间不存在隐性信道，那么会创建一个从代理（或代理类型，分别对应）内部实例的通道到代理（或代理类型，分别对应）外部通道的单向隐性信道。如果信道在某过程（或过程类型）内，那么它是非延迟的，否则它是延迟的；以及

b） 元素被添加进隐性信道的信号列表中。

10.2 连接

具体语法

<external channel identifiers> ::=
<channel identifier> { , <channel identifier}*

<channel definition area>中的<channel symbol>附于连接至嵌套框符号的一个<external channel identifiers>上。

语义

<external channel identifiers>中的<channel identifier>必须表示连接至嵌套代理的信道。连接至嵌套代理的每个信道都必须至少在一个<external channel identifiers>中被提到。

<external channel identifiers>中用<channel identifier>标识的每个信道都必须在定义连接的同一代理中进行定义，并且必须以该代理的边界作为它的其中一个端点。周边代理中定义的、以其环境作为它的其中一个端点的每个信道都必须确切地在一个<external channel identifiers>中被提到。

模型

连接是缩写形式的构件，可转化为通道。

在一个给定范围单元中，信道与<external channel identifiers>之间的每个不同连接定义了一个范围单元上的隐性通道。<external channel identifiers>中的所有信道都被连接至其各自范围单元的通道上。隐性通道的通道约束自连接至通道的信道上衍生而来。

通道名称是一个惟一的、明确的衍生名称。在周边范围单元中，用<channel identifier>标识的<channel definition area>被连接至该隐性通道。在范围单元内部，通过<external channel identifiers>与外部信道相关联的信道被连接至该隐性通道。

当图直接包含在另一个图中时（即它不是被引用的），那么每个<external channel identifiers>都可以被省略，原因是，从该图外部连接至该图框架上同一点的外部信道将直接显示。

10.3 信号

抽象语法

Signal-definition :: *Signal-name*
*Sort-reference-identifier**
Signal-identifier = *Identifier*
Signal-name = *Name*

具体语法

<signal definition>::=
<type preamble>
signal <signal definition item> { , <signal definition item> }* <end>

<signal definition item> ::=
<signal name>
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
[<sort list>]

<sort list> ::=
(<sort> { , <sort>}* **)**

<formal context parameters>中的<formal context parameter>必须是一个<sort context parameter>。作为<specialization>一部分的<base type>必须是一个<signal identifier>。

抽象信号只能在规范和信号约束中使用。

语义

信号实例是代理间的信息流，是用信号定义来定义的信号类型的一个实例化。信号实例可以通过环境或者通过代理来发送，并总是被发送至代理或环境。信号实例在对*Output-node*进行解释时进行创建，当对*Input-node*进行解释时，它不再存在。

<virtuality>的语义在第8.3.2节中进行定义。

10.4 信号列表定义

<signal list identifier>可以用在<signal list>中，作为信号标识符、远程过程、远程变量、计时器信号和接口列表的缩写形式。

具体语法

<signal list definition> ::=
signallist <signal list name> <equals sign> <signal list> <end>

<signal list area> ::=
<signal list symbol> ***contains*** <signal list>

<signal list symbol> ::=


<signal list> ::=
<signal list item> { **,** <signal list item>}*

<signal list item> ::=
<signal identifier>
| **(** <signal list identifier> **)**
| <timer identifier>
| [**procedure**] <remote procedure identifier>
| [**interface**] <interface identifier>
| [**remote**] <remote variable identifier>

通过用其表示的<signal identifier>或<timer identifier>列表来替换列表中的所有<signal list identifier>、通过用它们每个所指的隐性信号来替换所有的<remote procedure identifier>和<remote variable identifier>（见第10.5和10.6节）来构建<signal list>，它对应抽象语法中的*Signal-identifier*-**set**。

<signal list item>是一个<identifier>，如果可能的话，根据视见度规则，它或者表示一个<signal identifier>或<timer identifier>或<interface identifier>，或者表示一个<remote procedure identifier>，或者表示一个<remote variable identifier>。为强制使<signal list item>表示<remote procedure identifier>、<interface identifier>或<remote variable identifier>，可分别使用关键字**procedure**、**interface**或**remote**。

<signal list>不得包含以<signal list definition>定义的<signal list identifier>，不管是直接地还是间接地（通过另一个<signal list identifier>）。

10.5 远程过程

客户代理可以调用在另一个代理中定义的过程，它通过服务器代理中过程的远程过程调用向服务器代理发出请求来实现。

具体语法

<remote procedure definition> ::=
remote procedure <remote procedure name>
<procedure signature> <end>

<remote procedure call area> ::=
<procedure call symbol> ***contains*** <remote procedure call body>
[***is connected to*** <on exception association area>]

<remote procedure call body> ::=
<remote procedure identifier> [<actual parameters>]
<communication constraints>

<communication constraints> ::=
{**to** <destination> | **timer** <timer identifier> | <via path>}*

<remote procedure definition>引入了输入和输出过程的名称和特征。

输出过程是一个带关键字**exported**的过程。

输入过程和输出过程之间的关联是通过同时指向同一个<remote procedure definition>来建立的。

在输出过程定义中，跟在**as**后面的<remote procedure identifier>必须表示一个与输出过程具有相同特征的<remote procedure definition>。在不带**as**子句的输出过程定义中，输出过程的名称是隐含的，暗指最近周边范围中的同名<remote procedure definition>。

<remote procedure call body>中提到的远程过程必须在某嵌套代理类型或代理集的完全输出集中（见第8.1.1.1节和第9节）。

如果<remote procedure call body>中的<destination>是一个非pid类别的<pid expression>（见第12.1.6节），那么<remote procedure identifier>必须代表用于定义pid类别的接口内所包含的远程过程。

当省略<destination>和<via path>时，<remote procedure call body>和<procedure call body>之间在语法上存在不确定性。在这种情况下，如果可能的话，根据视见度规则，所含的<identifier>表示一个<procedure identifier>，否则表示一个<remote procedure identifier>。

<communication constraints>的<timer identifier>不得有与<exception identifier>相同的<identifier>。

在<remote procedure call body>中，<communication constraints>列表与最后一个<remote procedure identifier>相关联。例如，在下列句子中：

call p **to call** q **timer** t **via** g

timer t与**gate** g都将适用于**call** q。

<communication constraints>中包含的<destination>和<timer identifier>都不能多于一个。

模型

通过请求代理的远程过程调用将导致请求代理等待，直至服务器代理完成对过程的解释。在它等待的同时，传送给请求代理的信号得以保存。根据信号接收的正常次序，服务器代理将在下一个状态中对被请求的过程进行解释，在其中不对过程保存进行解释。如果既没有规定状态的<save area>，也没有规定状态的<input area>，那么增加一个只由过程调用组成并回至同一状态的隐性转换。如果规定了状态的<input area>，那么增加一个后跟<transition area>的、由过程调用组成的隐性转换。如果规定了状态的<save area>，那么增加一个有关被请求过程的信号隐性保存。

如下所述的一个远程过程调用体

Proc(apar) **to** destination **timer** timerlist **via** viapath

通过隐性定义信号的交换来建模。如果从远程过程调用中省略**to**或**via**子句，那么在以下转化中也可以将它们省略。如果已经在至少一个通道或信道（连接至输入器或输出器）<signal list>（有关输入器的输出以及有关输出器的输入）中提到远程过程，那么信号是显性的。请求代理向服务器代理发送一个包含过程调用实际参数的信号，对应**out**参数的实际参数除外，并等待回答。在回答该信号中，服务器代理对相应的远程过程进行解释，用所有**in/out**参数和**out**参数的结果向请求代理回送一个信号，而后对转换进行解释。

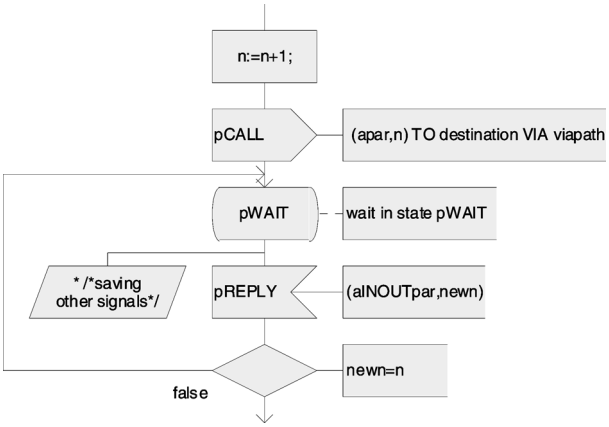
在<system diagram>中，对每个<remote procedure definition>有两个隐性<signal definition>。在这些<signal definition>中的<signal name>分别由pCALL和pREPLY来表示，其中p是惟一确定的。在同<remote procedure definition>的范围单元中对信号进行定义。pCALL和pREPLY的第一个参数都为预定义的整数类别。

在每个提到远程过程的信道上，用*pCALL*来替换远程过程。对每个这样的信号，在相反的方向上增加一个新的信号；该信道传送*pREPLY*信号。新的信道与最初信道具有相同的延迟属性。

a） 对每个输入的过程，定义两个隐性整数变量n和newn，并且n初始化为0。

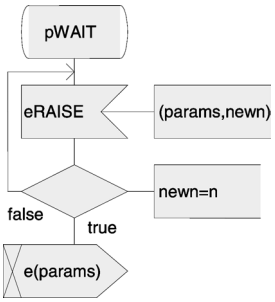
注1 — 引入参数n是为了认可并抛弃远程过程调用的回答信号，通过相关计时器的到期来抛弃它们。

<remote procedure call area>按如下步骤进行转化：



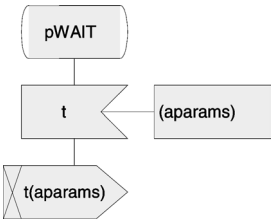
其中apar是实际参数列表，对应**out**参数的实际参数除外，aINOUTpar是实际**in/out**参数和**out**参数的经修改列表，如果对值返回远程调用进行转化，那么包括一个额外的参数。

对远程过程p的<raises>中所包含的每个异常以及所有预定义的异常e，定义一个信号eRAISE，它可以传输e的所有异常参数。在状态pWAIT中将插入下列内容：



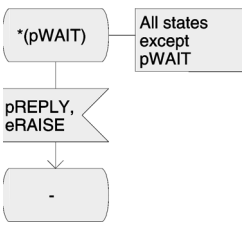
对于包括在<communication constraints>中的计时器t，在同计时器定义的范围中隐性地插入一个具有相同名称和参数的额外异常，不得存在一个显性定义的异常，它与同一范围单元（计时器在其中进行定义）中的计时器具有相同的名称。

另外，为计时器t（它包括在<communication constraints>中）插入下列内容：



其中aParams表示隐性定义的变量，参数类别包含在计时器定义中。

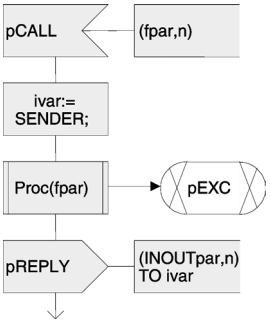
在pWAIT 除外的所有代理状态中插入：



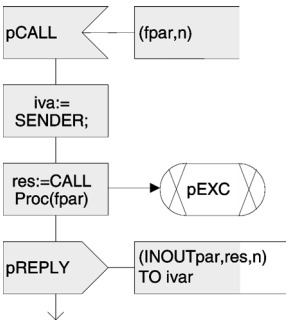
b） 在服务器代理中，为每个作为远程过程输入的显性或隐性<input area>定义一个隐性异常句柄pEXC和一个隐性整数变量。另外，在范围中为每个这样的<input area>定义一个ivar变量，在其中出现显性或隐性

远程过程输入。如果对值返回远程过程调用进行转化，那么定义一个与<procedure result>中<sort>具有相同类别的隐性变量res。

对带远程过程输入转换的所有<state area>，用下列<input area>替换远程过程输入，并导致对远程过程的转换：



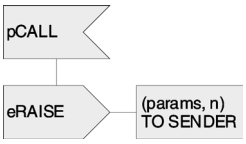
或，



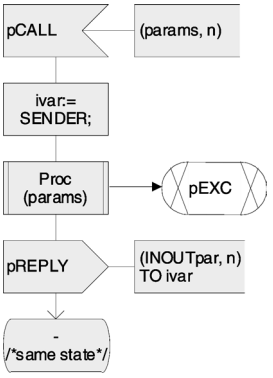
如果对值返回远程过程调用进行转化。

对所有带远程过程保存的<state area>，增加下列<save area>：

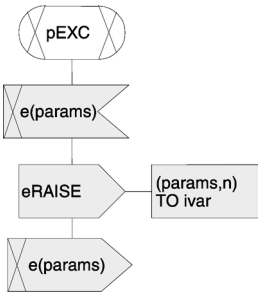
对所有带<remote procedure reject>的<state area>，增加下列<input area>，后跟远程过程拒绝转换：



对源自输入的、不包括隐性状态的所有其他<state area>，增加下列<input area>：



对包含在远程过程的<raises>中的每个异常e，以及对每个预定义的异常，插入下列内容：



如果异常句柄与远程过程输入相关，那么异常句柄变得与结果信号输入（未在上面模型中显示）相关。

注2 — 可能存在使用远程过程构件的死锁，尤其当没有＜destination＞给出时，或当＜destination＞不表示代理（通过规范来确保在接收pCALL信号之时它存在）的＜pid expression＞时。相关的计时器可以避免死锁。

10.6 远程变量

在SDL中，变量总为代理实例所拥有，并总为代理实例的局部变量。通常，变量只对拥有它的代理实例和所含的代理是可见的。如果在另一个代理中的代理实例需要访问与变量相关的数据项，那么需要与拥有变量的代理实例进行一次信号交换。

这可以通过以下缩写形式的符号来实现，即被称为输入的变量和输出的变量。缩写符号也可以用于将数据项输出给同一代理内的其他代理实例。

具体语法

＜remote variable definition＞ ::=
remote ＜remote variable name＞ {**,**＜remote variable name＞}* ＜sort＞
{**，** ＜remote variable name＞ {**，** ＜remote variable name＞}* ＜sort＞}*
＜end＞

＜import expression＞ ::=
import (＜remote variable identifier＞ ＜communication constraints＞ **)**

＜export body＞ ::=
(＜variable identifier＞ { **，** ＜variable identifier＞ }* **)**

＜remote variable definition＞用于引入有关输入变量和输出变量的名称和类别。

一个输出变量定义是一个带关键字**exported**的变量定义。

通过同时指向同一＜remote variable definition＞来建立输入变量与输出变量之间的关联。

输入变量规定为嵌套活动实体输出集的一部分。输出变量规定为嵌套活动实体完全输入集的一部分。

拥有变量（其数据项输出给其他代理实例）的代理实例称为变量的输出器。使用这些数据项的其他代理实例称为变量的输入器。变量称为输出变量。

输出变量定义中**as**后的＜remote variable identifier＞必须将具有相同类别的＜remote variable definition＞表示为输出变量定义。在没有**as**子句的情况下，在最近嵌套范围中的、具有相同名称和类别的远程变量定义表示为输出变量定义。

在＜import expression＞中提到的远程变量必须处于嵌套代理类型或代理集的完全输出集中（见第8.1.1.1节和9节）。

＜export body＞中的＜variable identifier＞必须表示用**exported**定义的变量。

如果＜import expression＞中的＜destination＞是一个非Pid类别（见第12.1.6节），那么＜remote variable identifier＞必须代表一个包含在接口（用于定义pid类别）中的远程变量。

模型

一个代理实例可以是同一远程变量的输入器和输出器。

a） 输出操作

输出变量在其＜variable definition＞中有关键字**exported**，并有一个隐性拷贝用在输入操作中。

输出操作是对＜export body＞的解释，输出器通过它发布输出变量的当前结果。输出操作将使输出变量的当前结果保存到其隐性拷贝中。

b） 输入操作

输入操作是对＜import expression＞的解释，输入器通过它访问输出变量的结果。结果保存在一个隐性变量中，隐性变量用＜import expression＞中的＜remote variable identifier＞来表示。通过＜import expression＞中的＜destination＞规定输出器包含的输出变量。如果没有规定＜destination＞，那么输入来

自输出同一远程变量的任意一个代理实例。输出器中的输出变量与输入器中的隐式变量之间的关联通过指向输出变量定义中和＜import expression＞中的同一远程变量来指定。

输入操作通过交换隐性定义信号来建模。输入器向输出器发送一个信号，并等待回答。在对该信号的回答中，输出器向输入器回送一个信号，带有输出变量的隐性拷贝中包含的结果。

如果一个缺省初始化附于输出变量，或者如果输出变量在其定义时进行了初始化，那么该隐性拷贝也被初始化，其结果与输出变量相同。

对系统定义中的每个＜remote variable definition＞，都有两个隐性＜signal definition＞。这些＜signal definition＞中的＜signal name＞分别用xQUERY和xREPLY表示，其中x是指＜remote variable definition＞的＜name＞。信号在与＜remote variable definition＞相同的范围单元中进行定义。信号xQUERY有一个预定义类别Integer的变元，xREPLY有变量和Integer类别的变元。输出变量的隐性拷贝用imcx来表示。

在每个提到远程变量的信道上，用xQUERY来替换远程变量。对每个这样的信道，在相反的方向上增加一个新的信道；该信道承载xREPLY信号。在一个信道的情况下，新的信道与最初的信道有相同的延迟属性。

为每个预定义的异常（表示为predefExc），定义一个额外的匿名信号（表示为predefExcRAISE）。

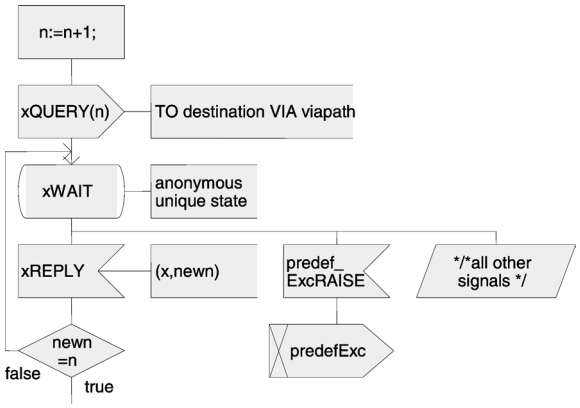
a） 输入器

为每个输入变量，定义两个隐性整数变量n和newn，并将n初始化为0。另外，对远程变量类别的隐性变量x在＜import expression＞的正文中进行定义。

＜import expression＞表达式

import (x **to** destination **via** via-path)

将被转化为以下内容，在初始表达式中，如果没有出现destination，那么省略**to**子句；如果没有出现destination，那么省略**via**子句。



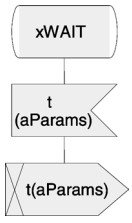
接着，原先包含＜import expression＞的＜import expression＞用x替换。

在所有其他状态下，保存xREPLY。

注1 — 返回语句终结依据第11.12.1节引入的隐性过程。

对＜communication constraints＞中包含的每个计时器t，在计时器定义相同的范围中，隐性地插入一个具有相同名称、相同参数的额外异常。在该情况下，在定时器定义的范围单元内不得存在具有相同名称的异常。

另外，为每个包括在＜communication constraints＞中的计时器t增加下列内容：

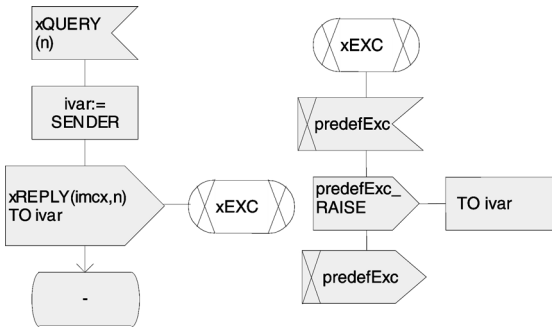


其中aParams代表隐性定义的变量，其参数类别包含在计时器定义中。

转化结果封装在一个隐性过程中，如第11.12.1节所述。附于入口的每个<on exception association area>都应附于隐性过程的一个调用上。

b） 输出器

对出口器的所有<state area>，源自入口的隐性声明除外，增加下列<input area>：



对每个这样的状态，ivar将被定义为Pid类别的变量，并将n定义为整数类型的变量。

<export statement>

export x;

被转化为如下所示：

imcx := x;

注2 — 可能存在使用入口构件的死锁，尤其当没有<destination>给出时，或当<destination>不表示代理（通过规范来确保在接收xQUERY信号之时它存在）的<pid expression>时。通过在<import expression>中规定一个设置好的计时器，可以避免此类死锁。

11 行为

11.1 开始

抽象语法

State-start-node :: [*On-exception*]
[*State-entry-point-name*]
Transition

具体语法

<start area> ::=
<start symbol> ***contains*** { [<virtuality>] [<state entry point name>]}
[***is connected to*** <on exception association area>]
is followed by <transition area>

<start symbol> ::=

如果在<start area>中给出了<state entry point name>，那么<start area>必须为<composite state area>的<start area>。

语义

解释*State-start-node*的*Transition*。

模型

包含<virtuality>的<start area>被称为虚拟开始。虚拟开始在第8.3.3节中做进一步描述。

包含在<composite state area>中的<start area>在第11.11节中定义。

11.2 状态

抽象语法

State-node :: *State-name*
[*On-exception*]
Save-signalset
*Input-node**-set***
*Spontaneous-transition**-set***
*Continuous-signal**-set***
*Connect-node**-set***
[*Composite-state-type-identifier*]

State-name = *Name*

在*State-transition-graph* 或 *Procedure-graph* 内的*State-node*s必须具有不同的*State-name*s。

对于每个 *State-node*，所有(在完整合法的输入信号集中的) *Signal-identifier*s 出现在*Save-signalset*或*Input-node*中。

在 *Input-node*-**set**中的*Signal-identifier*s必须是不同的。

带*Composite-state-type-identifier*的*State-node*代表复合状态。

具体语法

<state area> ::=
<state symbol> ***contains*** <state list>
[***is connected to*** <on exception association area>]
is associated with
{ <input association area>
| <priority input association area>
| <continuous signal association area>
| <spontaneous transition association area>
| <save association area>
| <connect association area> }*

<state symbol> ::=

<state list> ::=
{ <basic state name> | <composite state item> }
{ **,** { <basic state name> | <composite state item> } }*
| <asterisk state list>

<basic state name> ::=
<state name>

<asterisk state list> ::=
<asterisk> [**(** <state name> { **,** <state name>}* **)**]

<composite state item> ::=
<composite state name> [<actual parameters>]
| <typebased composite state>

<composite state name> ::=
<state name>

<input association area> ::=
<solid association symbol> ***is connected to*** <input area>

<save association area> ::=
<solid association symbol> ***is connected to*** <save area>

<spontaneous transition association area> ::=
<solid association symbol> ***is connected to*** <spontaneous transition area>

<connect association area> ::=
<solid association symbol> ***is connected to*** <connect area>

一个<state area>代表一个或多个*State-node*。

<basic state name>为以下状态的名称，它没有<composite state area>，并且不在<typebased composite state>中定义。<composite state name>为以下状态的名称，它有<composite state area>，或在<typebased composite state>中定义。

一个给定的状态至多可以有一个相关的异常句柄。

当<state list>包含一个<state name>时，<state name>代表一个*State-node*。对每个*State-node*，通过<save area>来表示*Save-signalset*，并保存任何隐含的信号。对每个*State-node*，通过<input area>来表示*Input-node*-**set**，并保存任何隐含的输入信号。对每个*State-node*，通过<spontaneous transition area>来表示*Spontaneous-transition*。

<state name>可以出现在实体的多个<state area>中。

<asterisk state list>中的<state name>必须不同，并且必须包含在嵌套实体或父类型实体中的其他<state list>中。

<composite state item>或<typebased composite state>应只能包含<actual parameters>，如果它在一个与<nextstate area>一致的<state area>中。在这种情况下，<state area>必须只能包含<composite state name>，以及可选地，<actual parameters>。

源自<state symbol>的<solid association symbol>可以拥有一条共同的初始路径。

<connect association area>只允许针对带<state list>的<state area>，它包含一个<composite state item>。

语义

一个状态代表一个基本状态或一个复合状态应用。

复合状态应用的语义在第11.11节中给出。

一个基本状态代表一个特殊的条件，在此条件下，代理的状态机可以使用一个信号实例。如果使用了一个信号实例，那么对相关的转换进行解释。转换还可以被解释为连续信号的结果或自发转换。

对每个状态，按以下步骤对*Save-signal*、*Input-node*、*Spontaneous-signal*和*Continuous-signal*进行解释。各步骤每重复一次，就将相关的信号集更新为数据端口上的信号；否则，每步的集合都相同。

a） 如果输入端口包含一个匹配当前状态优先级输入的信号，那么使用第一个这样的信号（见第11.4节）；否则

b） 按输入端口上的信号次序：

1） 如果有的话，以任意次序对对应当前信号的*Input-node*的*Provided-expression*进行解释；

2） 如果当前信号可用，那么使用该信号（见第11.6节）；否则

3） 选择输入端口上的下一个信号。

c） 如果没有发现可用的信号，那么以*Continuous-signal*的优先次序，如果有的话，以具有相同优先级的*Continuous-signal*（以任意次序考虑并且不考虑优先级）作为最低优先级：

1） 对包含在当前*Continuous-signal* 中的*Continuous-expression*进行解释；

2） 如果当前的连续信号可用，那么使用该信号（见第11.5节）；否则

3） 选择下一个连续信号。

d） 如果没有发现可用的信号，一旦输入端口上的信号不同于已考虑的信号集，或如果存在带*Provided-expression*（可以改变）或*Continuous-expression*（可以改变）的Input-node，那么应尽快重复这些步骤。只有当它包含一个*NOW-expression*、*Timer-active-expression*、*Any-expression*，或*Variable-access*一个变量时，变量在嵌套的进程中定义，通过在另一个代理实例或另一个状态划分中的指配来改变，Provided-*expression* 或 *Continuous-expression*才可以改变。

任何时候，在一个包含*Spontaneous-transition*的状态中，状态机都可以对*Spontaneous-transition*的*Provided-expression*进行解释，并随后，如果*Spontaneous-transition*可用，对其中一个*Spontaneous-transition*的*Transition*进行解释（见第11.9节），或如果不存在*Provided-expression*，对其中一个*Spontaneous-transition*的*Transition*进行解释。

模型

当<state area> 的 <state list> 包含一个以上的<state name> 项时，则为每个这样的<state name>创建一个此<state area>的备份。此后， <state area> 被这些备份取代。

当若干个<state area>都包含同一<state name>时，这些<state area>结合成一个具有该<state name>的<state area>。

带<asterisk state list>的一个<state area>被转化为一组<state area>，一个对应所议之实体的一个<state name>和<composite state name>，那些包含在<asterisk state list>中的<state name>和<composite state name>除外。

11.3 输入

抽象语法

Input-node :: [***PRIORITY***]
Signal-identifier
[*Variable-identifier*]*
[*Provided-expression*]
[*On-exception*]
Transition

Variable-identifier = *Identifier*

任选 *Variable-identifier*s的长度必须与由*Signal-identifier*表示的*Signal-definition*中的*Sort-reference-identifiers*地数量一样。

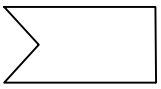
变量的类别应该与那些可以由信号携带的值的类别通过位置来对应。

具体语法

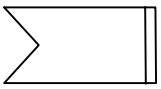
<input area> ::=
<input symbol> ***contains*** { [<virtuality>= <input list> }
[***is connected to*** <on exception association area>]
[***is associated with*** <solid association symbol> ***is connected to*** <enabling condition area>]
is followed by <transition area>

<input symbol> ::=
<plain input symbol>
| <internal input symbol>

<plain input symbol> ::=



<internal input symbol> ::=



<input list> ::=
<stimulus> { **,** <stimulus> }*
| <asterisk input list>

<stimulus> ::=
<signal list item>
[**(** [<variable>] { **,** [<variable>] }* **)** | <remote procedure reject>]

<remote procedure reject> ::=
raise <exception raise>

<asterisk input list> ::=
<asterisk>

当<input list> 包含一个<stimulus>时，则<input area>对应于一个*Input-node*。

包含在<input symbol>中的每个<signal identifier>或<timer identifier>给出一个*Input-node*的名称，它由该<input symbol>来表示。

注 — <plain input symbol>与<internal input symbol>在含义上没有区别。

一个<state area>至多可以包含一个<asterisk input list>。一个<state area>不得同时包含<asterisk input list>和<asterisk save list>。

只有当<signal list item>表示一个<remote procedure identifier>时，才可以规定一个<remote procedure reject>。在<remote procedure reject>中的<exception identifier>必须要在<remote procedure definition>中被提到。

一个<signal list item>不得表示一个<remote variable identifier>，如果它表示一个<remote procedure identifier>或<signal list identifier>，那么<stimulus>参数（包括圆括号）必须被省略。

当<input list>包含一个<stimulus>时，<input area>代表一个*Input-node*。在抽象句法中，计时器信号（<timer identifier>=也由*Signal-identifier*来表示。计时器信号和普通信号只有在适当的地方才有区别，在许多方面它们有类似的属性。计时器信号的确切属性在第11.15节中定义。

<stimulus>中最后<variable>后的逗号可以被省略。

在抽象句法中，<remote procedure identifier>也可以被表示为*Signal-identifier*。

语义

输入允许使用规定的输入信号实例。使用输入信号可使通过信号传送的信息对代理可用。为输入相关变量指配通过所用信号传送的数据项。

从左到右向变量指配数据项。如果对在信号中规定的类别没有输入相关的变量，那么抛弃对应的数据项。如果没有类别相关的数据项（类别在信号中规定），那么对应的变量变成“未定义的”。

向使用代理的发送者（见第9节模型）提供发起代理的pid，由信号实例承载。

从环境流向系统内某个代理实例的信号实例将总承载一个不同于系统中任何pid的pid。

模型

其<signal list item>为<signal list identifier>的<stimulus>是无参数之<stimulus>列表的衍生语法，并被插入到嵌套的<input list> 或 <priority input list>中。在该列表中，<stimulus>与信号列表的成员之间存在一对一的通信。

当<input area>的<stimulus>列表包含多个<stimulus>时，为每个这样的<stimulus>创建一份<input area>的拷贝。而后用这些拷贝替换<input area>。

当一个或多个<stimulus>的<variable>为<indexed variable>或<field variable>时，则用惟一的、新的、隐含声明的<variable identifier>来替换所有的<variable>。紧跟<input area>之后，插入一个<task area>，在它的<task body>中，针对每个<variable>，包含一个<assignment>，用于将对应之新变量的结果指配给<variable>。结果将按<variable>列表从左到右的次序进行指配。该<task area>变成为<transition area>中的第一个<action area>。

一个<asterisk input list>转化为一个<input area>列表，一个对应嵌套<agent diagram>的完全合法输入信号集的一个成员，由第10.5、10.6、11.4、11.5和11.6节中概念引入的隐含输入信号的<signal identifier>除外，包含在<state area>的其他<input list>和<save list>中的<signal identifier>也除外。

包含<virtuality>的<input area>被称为虚拟输入转换。对虚拟输入转换在第8.3.3节中有更多的描述。

11.4 优先级输入

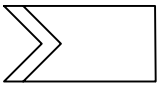
在一些情况下，可以很方便地来表示某个信号的接收优先于其他信号的接收。这可以通过优先级输入的方式来表示。

具体语法

<priority input association area> ::=
<solid association symbol> ***is connected to*** <priority input area>

<priority input area> ::=
<priority input symbol> ***contains*** { [<virtuality>] <priority input list> }
[***is connected to*** <on exception association area>]
is followed by <transition area>

<priority input symbol> ::=



<priority input list> ::=

<stimulus> {，<stimulus>}*

<priority input association area>表示一个带**PRIORITY**的*Input-node*。

语义

如果状态的*Input-node*有**PRIORITY**，那么信号是一个优先级信号，并将在任何其他信号被使用之前得到使用，条件是它有一个可用的转换。

模型

包含<virtuality>的<priority input area>被称为虚拟优先级输入。对虚拟优先级输入在第8.3.3节中有更多的描述。

11.5 连续信号

在对系统进行描述过程中，状况可以发生变化，当满足某个条件时，应对某个转换进行解释。一个连续的信号用于解释一个布尔表达式，当表达式返回预定义的布尔值“真”时，对相关的转换进行解释。

抽象语法

Continuous-signal	::	[*On-exception*] *Continuous-expression* [*Priority-name*] *Transition*
Continuous-expression	=	*Boolean-expression*
Priority-name	=	*Nat*

具体语法

<continuous signal association area> ::=

<solid association symbol> ***is connected to*** <continuous signal area>

<continuous signal area> ::=

<enabling condition symbol>
contains {
[<virtuality>] <continuous expression>
[[<end>] **priority** <priority name>] }
[***is connected to*** <on exception association area>]
is followed by <transition area>

<continuous expression> ::=

<Boolean expression>

<priority name> ::=

<Natural literal name>

语义

*Continuous-expression*解释为状态的一部分，其*Continuous-signal*与之相关（见第11.2节）。如果*Continuous-expression*返回预定义的布尔值“真”，那么连续信号可用。

有*Priority-name*最低值的连续信号拥有最高的优先级。

模型

包含<virtuality>的<continuous signal area>被称为虚拟连续信号。对虚拟连续转换在第8.3.3节中有更多的描述。

11.6 使能条件

一个使能的条件使对信号使用附加一个条件变得可能，超过其接收，或对自发转换施加一个条件。

抽象语法

Provided-expression = *Boolean-expression*

具体语法

<enabling condition area> ::=
<enabling condition symbol> ***contains*** <provided expression>

<enabling condition symbol> ::=

<provided expression> ::=
<Boolean expression>

当<provided expression>包含一个<imperative expression>时，*Provided-expression*为一个*Value-returning-call-node*，它只包含由以下*Model*隐含定义的过程的*Procedure-identifier*。否则，*Provided-expression*由<provided expression>来表示。

语义

*Input-node*的*Provided-expression*解释为该*Input-node*所附之状态的一部分（见第11.2节）。

如果*Input-node*的所有*Provided-expression*返回预定义的布尔值“真”，或者如果*Input-node*没有*Provided-expression*，那么输入端口中的信号可用。在代理处于该状态时，可以在任何时候对*Spontaneous-transition*的*Provided-expression*进行解释。

模型

当<provided expression>包含<imperative expression>时，隐含定义一个匿名过程。该过程返回一个布尔类型，并包含一个单个的<return area>，并以<provided expression>作为它的<return body>。

注 — 可以根据<import expression>模型，对<Boolean expression>做进一步转化。

11.7 保存

保存规定了一组信号标识符和远程过程标识符，其实例与所处状态下的代理不相关，保存附于其上，需要对它进行保存，以便今后处理。

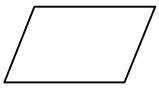
抽象语法

Save-signalset = *Signal-identifier-**set***

具体语法

<save area> ::=
<save symbol> ***contains*** { [<virtuality>] <save list> }

<save symbol> ::=



<save list> ::=
<signal list>
| <asterisk save list>

<asterisk save list> ::=
<asterisk>

<save list>表示*Signal-identifier*-**set**。

<state area>至多可以包含一个<asterisk save list>。<state area>不得同时包含<asterisk input list>和<asterisk save list>。

语义

Save-signalset中的信号是不可用的。

保存的信号按其到达的次序保留在输入端口中。

保存的效果仅对于保存所附的状态有效。在后继状态中，已经“保存的”信号实例当做为正常的信号实例。

模型

<asterisk save list>转化为一个<stimulus>列表，包含嵌套<agent diagram>的完全有效输入信号集，由第10.5、10.6、11.4、11.5和11.6节中概念引入的隐含输入信号的<signal identifier>除外，包含在<state area>的其他<input list>和<save list>中的<signal identifier>也除外。

包含<virtuality>的<save area> 或 <save area>被称为虚拟保存。对虚拟保存在第8.3.3节中有更多的描述。

11.8 隐性转换

具体语法

在<signal identifier>集中，可以省略包含在<agent diagram>完全合法输入信号集中的<signal identifier>，<signal identifier>包含在<state area>的<input list>、<priority input list>和<save list>中。

模型

对每个<state area>，存在一个隐含的<input area>，<input area>包含一个<transition area>，<transition area>只包含一个导致相同<state area>的<nextstate area>。

11.9 自发转换

自发转换规定了一个不带任何信号接收的状态转换。

抽象语法

Spontaneous-transition :: [*On-exception*]
[*Provided-expression*]
Transition

具体语法

<spontaneous transition area> ::=
<input symbol> ***contains*** { [<virtuality>] <spontaneous designator> }
[***is connected to*** <on exception association area>]
[***is associated with*** <solid association symbol> ***is connected to*** <enabling condition area>]
is followed by <transition area>

<spontaneous designator> ::=
none

语义

自发转换允许激活一个不带任何激励（提交给代理）的转换。激活一个自发转换与在代理的输入端口中是否存在信号实例无关。在由信号接收和自发转换激活的转换间不存在优先级。

激活一个自发转换后，代理的**sender**表达式返回**self**。

模型

包含<virtuality>的<spontaneous transition area>被称为虚拟自发转换。对虚拟自发转换在第8.3.3节中有更多的描述。

11.10 标签

抽象语法

Free-action :: *Connector-name*
Transition
Connector-name = *Name*

具体语法

<in connector area> ::=
<in connector symbol> ***contains*** <connector name>
is followed by <transition area>

<in connector symbol> ::=

术语“body”指的是一个状态机图形，可能在从<statement list>的转化之后，以及在从一个类型的转化之后。因此boby指的是<procedure definition>中的<statement list>以及<operation definition>、<agent body area>、<procedure body area>、<operation body area>，或<composite state body area> 和 <state aggregation body area>。

在boby中定义的所有<connector name>必须都是不同的。

标签<in connector area>是控制传送的入口点，控制来自与同一boby中相同<connector name>的对应连接。

只允许对同一boby中的各标签进行控制传送。允许从特殊化的boby到在父类型中定义的一个连接器之间有一个连接。

<in connector area>表示<flow line symbol>的连续性，来自同一<agent body area>或<procedure body area>中的、具有相同<connector name>的对应<out connector area>。

语义

*Free-action*定义*Join-node*的目标。

11.11 状态机和复合状态

复合状态是这样一种状态，它由顺次解释的子状态（带相关的转换）组成，或由以交叉模式解释的子状态聚合组成。子状态是一个状态，因此反过来一个子状态也可以是一个复合状态。

复合状态（子状态、转换、变量和过程）的属性通过<composite state area>或<composite state type diagram>以及通过状态机或复合状态中带<composite state name>的<state area>规范来定义。

抽象语法

Composite-state-formal-parameter	=	*Agent-formal-parameter*
State-entry-point-definition	=	*Name*
State-exit-point-definition	=	*Name*
Entry-procedure-definition	=	*Procedure-definition*
Exit-procedure-definition	=	*Procedure-definition*
Named-start-node	::	*State-entry-point-name* [*On-exception*] *Transition*
State-entry-point-name	=	*Name*

*Entry-procedure-definition*代表一个带名称入口的过程。*Exit-procedure-definition*代表一个带名称出口的过程。这些过程不得有参数，并只能含有一个转换。

具体语法

<composite state area> ::=

<composite state graph area> | <state aggregation area>

语义

在创建嵌套实体时创建复合状态，并在删除嵌套实体时删除复合状态。

分别在创建和删除复合状态时创建和删除局部变量。如果<variable definition>包含<constant expression>，那么在创建时为<variable definition>指配<constant expression>的结果。如果没有<constant expression>，那么<variable definition>的结果是未定义的。

*Composite-state-formal-parameter*为局部变量，它们在创建复合状态时创建。在进入复合状态之时，若出现在*Nextstate-node*中，那么通过对应的实际参数，将表达式的结果指配给变量。否则，变量的结果变成为未定义。

源自子状态的转换比源自任何包含状态的冲突转换优先级都要高。冲突转换指的是由相同输入、优先级输入、保存或连续信号触发的转换。

如果已经定义，那么当状态进入和退出时，分别隐性地调用*Entry-procedure-definition* 和 *Exit-procedure-definition*。对其中一个或两个过程不做强制定义。在调用起始转换之前，或如果因为解释一个带**HISTORY**的*Nextstate-node*的结果而重新进入状态，则调用进入过程。在解释*Composite-state-graph*的*Return-node*之后，并在解释直接附于*State-node*的转换之前（如果存在此类转换的话），调用退出过程。当在复合状态中引起一个异常时，不调用退出过程。

模型

<composite state area>有一个隐含的匿名复合状态类型，它定义了复合状态的各属性。

为一个特殊化的<composite state area>是一种缩写形式，用于定义一个隐性的复合状态类型以及该类型的一个基于类型的复合状态。

11.11.1 复合状态图

在复合状态图中，转换按序进行解释。

抽象语法

Composite-state-graph :: *State-transition-graph*
[*Entry-procedure-definition*]
[*Exit-procedure-definition*]
*Named-start-node-**set***

State-transition-graph :: [*On-exception*]
[*State-start-node*]
*State-node-**set***
*Free-action-**set***
*Exception-handler-node-**set***

在*SDL-specification*中，所有潜在实例化的代理都必须有一个*State-start-node*。在代理中必须确切地存在一个没有标签的*State-start-node*。

具体语法

<composite state graph area> ::=
<frame symbol> ***contains*** {
{ <composite state heading> <composite state structure area> }
is associated with {<state connection point>* } ***set***
is connected to { {<gate on diagram> | <external channel identifiers>}* }***set***
[***is associated with*** <package use area>]

<composite state heading> ::=
state [<qualifier>] <composite state name>
[<agent formal parameters>] [<specialization>]

<composite state structure area> ::=
{ <composite state text area>*
<entity in composite state area>*
{ <composite state body area> | <state aggregation body area> }}***set***

Composite-state-graph 代表 <composite state body area>。

只有当它直接包含在一个带<state aggregation type heading>的<state aggregation area>或<composite state type diagram>时，<composite state structure area>才应包含一个<state aggregation body area>；否则它包含一个<composite state body area>。

<composite state text area> ::=

<text symbol> ***contains***
{ <valid input signal set>
| <variable definition>
| <data definition>
| <data type reference>
| <procedure definition>
| <procedure reference>
| <exception definition>
| <select definition>
| <macro definition>}*

<entity in composite state area> ::=
<procedure area>
| <data type reference area>
| <composite state area>
| <composite state type diagram>
| <composite state type reference area>

<composite state body area> ::=
[<on exception association area>]
<start area>* { <state area> | <exception handler area> | <in connector area> }*

<composite state graph area>（或对应的复合状态类型定义）的<composite state text area>中至多应有一个<valid input signal set>。<valid input signal set>不得包含在<state aggregation area>（或对应的复合状态类型定义）的<composite state text area>中。

<package use area>必须置于<frame symbol>的最上面。

至多应有一个<start area>没有标签。每个额外的带标签的入口和出口点必须都通过对应的<state connection point>进行定义。每个额外的带标签的<start area>应包含有一个不同的<state entry point name>。

<composite state body area>中带有<state entry point name>（一个带标签返回）的<start area>指的只能是直接嵌套<composite state body area>的<composite state graph area>的<state entry point>。

如果<composite state body area>至少包含一个不同于星号状态的<state area>，那么必须出现<start area>。

如果<composite state area>被<procedure diagram>嵌套，那么<composite state text area>中的<variable definition>不得包含**exported**<variable name>。

<channel definition area>可以只连接至<composite state graph area>，其中<composite state graph area>为代表代理状态机的<state partition area>。

语义

如果*Composite-state-graph*至少包含一个*State-start-node*但不包含*State-node*，那么*Composite-state-graph*应被解释为转换的经封装的一部分。

*Composite-state-graph*没有标签的*State-start-node*被解释为复合状态的缺省入口点。当*Nextstate-node*没有*State-entry-point*时，对它进行解释。*Named-Start-nodes*被解释为复合状态额外的入口点。通过*Nextstate-node*的*State-entry-point*来定义对哪个命名的起始转换进行解释。

在复合状态中的*Action-return-node*被解释为复合状态的缺省出口点。对*Action-return-node*的解释将触发嵌套实体中不带名称的*Connect-node*。额外的*Named-return-node*应被解释为复合状态的额外的出口点。对*Named-return-node*的解释将触发嵌套实体（包含在一个带有相同名称的C*onnect-node*中）中的出口转换。

对状态图的各节点，以代理或过程图等同节点的方式进行解释。也就是说，状态图与嵌套代理具有相同的、完全合法的输入信号集，状态图与嵌套代理的实例具有相同的输入端口。

模型

注 — 规定一个只由转换（与星状态相关，不带<start area>并不带任何子状态）组成的<composite state area>是可能的。这些转换或者通过<dash nextstate>终结，或者通过<return area>终结。当代理或过程在复合状态中时，应用这

些转换。由＜dash nextstate＞终结的这样一个转换的nextstate是复合状态；不过，不调用复合状态的*Exit-procedure-definition*和*Entry-procedure-definition*。

如果＜composite state area＞不包含带＜state name＞的＜state area＞，但只包含带＜asterisk＞的＜state area＞，那么星状态转化为一个带＜state name＞的＜state area＞以及一个导致本＜state area＞的＜start area＞。

11.11.2 状态聚合

状态聚合是对复合状态的划分。它由多个复合状态组成，它对交互转换进行解释。在某个给定时间，状态聚合的每个划分部分处于该划分的其中一个状态，或（只对其中一个划分部分）处于转换，或已经完成并等待其他划分部分完成。每个转换都要完成。

抽象语法

State-aggregation-node	::	*State-partition-**set***
		[*Entry-procedure-definition*]
		[*Exit-procedure-definition*]
State-partition	::	*Name*
		Composite-state-type-identifier
		*Connection-definition-**set***
Connection-definition	::	*Entry-connection-definition* | *Exit-connection-definition*
Entry-connection-definition	::	*Outer-entry-point Inner-entry-point*
Outer-entry-point	::	*State-entry-point-name* | ***DEFAULT***
Inner-entry-point	::	*State-entry-point-name* | ***DEFAULT***
Exit-connection-definition	::	*Outer-exit-point Inner-exit-point*
Outer-exit-point	::	*State-exit-point-name* | ***DEFAULT***
Inner-exit-point	::	*State-exit-point-name* | ***DEFAULT***

Outer-entry-point 中的 *State-entry-point-name* 必须表示 *Composite-state-type-definition* 的 *State-entry-point-definition*，在其中出现*State-aggregation-node*。*Inner-entry-point*的*State-entry-point-name*必须表示*State-partition*中复合状态的*State-entry-point-definition*。同样，*State-exit-point*必须分别表示在内部和外部复合状态中的出口点。**DEFAULT**指明没有标签的入口点和出口点。

对每个*State-partition*，存储器状态的每个入口点都应准确地出现在一个*Connection-definition*中。对每个*State-partition*，*State-partition*的每个出口点都应准确地出现在一个*Connection-definition*中。

复合状态中*State-partition*的输入信号集必须脱离。*State-partition*的输入信号集被定义为复合状态内*Input-node*或*Save-signalset*中所出现的所有信号的联合，包括嵌套的状态以及在*Call-node*中提到的过程。

具体语法

＜state aggregation area＞ ::=
　　＜frame symbol＞ ***contains*** {
　　　　＜state aggregation heading＞
　　　　＜composite state structure area＞
　　is associated with {＜state connection point＞* } ***set***
　　is connected to { {＜gate on diagram＞ | ＜external channel identifiers＞}* }***set***
　　[***is associated with*** ＜package use area＞]

＜state aggregation heading＞ ::=
　　state aggregation [＜qualifier＞] ＜composite state name＞
　　[＜agent formal parameters＞][＜specialization＞]

＜state aggregation body area＞ ::=
　　{ { ＜state partition area＞ | ＜state partition connection area＞}* }***set***

＜state partition area＞ ::=
　　＜composite state reference area＞
　| ＜composite state area＞
　| ＜typebased state partition definition＞
　| ＜inherited state partition definition＞

<typebased state partition definition> ::=
<state symbol> ***contains*** { <typebased state partition heading> { <gate>*}***set*** }

<typebased state partition heading> ::=
<state name> <colon> <composite state type expression>

<inherited state partition definition> ::=
<dashed state symbol> ***contains*** <composite state identifier>

<dashed state symbol> ::=

<state partition connection area> ::=
<solid association symbol>
is connected to [<outer graphical point> <inner graphical point>]

<outer graphical point> ::=
{ <state entry points> | <state exit points> } ***is associated with*** <frame symbol>

<inner graphical point> ::=
{ <state entry points> | <state exit points> } ***is associated with*** <state partition area>

包含在<state symbol>中的<gate>置于符号边界附近，并与到信道的连接点相关。它们置于<state symbol>处的信道终点附近。

只有当<state partition area>代表一个代理或代理类型的状态机时，才允许在<typebased state partition definition>的<state symbol>或<state partition area>的<composite state reference area>中出现<gate>。

语义

如果*Composite-state-type-definition*包含*State aggregation-node*，那么以一种交叉方式在转换层面对每个*State-partition*的复合状态进行解释。在解释另一个转换之前，每个转换都必须完成。用状态划分创建一个复合状态意味着创建每个所含*State-partition*的及其连接。如果*State-partition*的*Composite-state-type-definition*有*Composite-state-formal-parameter*，那么当进入状态时，这些形式参数是未定义的。

以任何次序对划分的没有标签的*State-start-node*进行解释，作为复合状态的缺省入口点。当*Nextstate-node*没有*State-entry-point*时，对它们进行解释。*Named-start-node*被解释为复合状态的额外入口点。如果通过*Entry-connection-definition*的*Outer-entry-point*进入复合状态，那么对带相应*Inner-entry-point*的划分的起始转换进行解释。在状态聚合的进入过程完成后，以未确定的次序进入状态划分。

当（以任何次序）将每个划分解释为一个*Action-Return-node* 或 *Named-return-node*时，划分退出复合状态。*Exit-connection-definition*将自划分的退出点和复合状态的退出点关联起来。如果不同的划分通过不同的退出点退出复合状态，那么以非确定的方式选择复合状态的退出点。在所有状态划分都已经完成后，对状态聚合的退出过程进行解释。在所有其他划分完成之前，保存已完成其返回节点的、划分输入集中的信号。

以相同方式，对状态划分图的各节点进行解释，作为代理或过程图的等同节点，惟一的不同是它们具有脱离的输入信号集。状态划分共享同一作为嵌套代理的输入端口。

与复合状态应用（包含*State-aggregation-node*）关联的输入转换适用于所有状态划分的所有状态，并且它表示所有这些的一个缺省终结。如果这样一个转换以一个带**HISTORY**的*Nextstate-node*来终结，那么所有划分重新进入其相应的子状态。

模型

如果状态复合的入口点不与状态划分的任何入口点连接，那么增加一个与没有标签的入口的隐含连接。同样，如果状态划分的出口点不与状态复合的任何出口点连接，那么增加一个与没有标签的出口的隐含连接。

如果在代理（不由某个复合状态的任何状态划分来使用）的完全合法的输入集中存在信号，那么向复合状态增加一个额外的隐含状态划分。该隐含划分只有一个没有标签的起始转换，以及一个包含所有隐含转换的状态（包括那些有关出口过程和出口变量的转换）。当其他划分的其中一个退出时，向代理发送一个隐含的信号，它由隐含的划分来使用。在隐含划分使用过所有隐含的信号后，它通过一个*State-return-node*来退出。

11.11.3 状态连接点

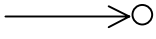
状态连接点在复合状态中进行定义，包括直接规定的复合状态和状态类型，并代表复合状态的入口和出口连接点。

具体语法

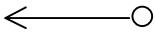
<state connection point> ::=
<state connection point symbol>
is associated with { <state entry points> | <state exit points> }
is connected to <frame symbol>

<state connection point symbol> ::=
<state connection point symbol 1> | <state connection point symbol 2>

<state connection point symbol 1> ::=



<state connection point symbol 2> ::=



<state entry points> ::=
<state entry point>
| **(** <state entry point> { **,** <state entry point> }* **)**

<state exit points> ::=
<state exit point>
| **(** <state exit point> { **,** <state exit point> }* **)**

<state entry point> ::=
<state entry point name>

<state exit point> ::=
<state exit point name>

对<state connection point symbol 1>，<state connection point>必须包含<state entry points>；否则<state connection point>必须包含<state exit points>。

在<state connection point symbol 1> 和 <state connection point symbol 2>中，圆的中心必须置于它所连接的<frame symbol>的边缘上。

语义

*State-entry-point-definition*在<composite state area>上定义一个入口点。*State-exit-point-definition*在<composite state area>上定义一个出口点。

每个复合状态都隐性地定义了两个匿名状态连接点。这些是缺省的入口和出口点，分别对应一个没有标签的 *State-start-node* 和 *Return-node*。

11.11.4 连接

抽象语法

Connect-node :: [*State-exit-point-name*]
[*On-exception*]
Transition
State-exit-point-name = *Name*

具体语法

<connect area> ::=
[<virtuality>] [<connect list>]
[***is connected to*** <on exception association area>]
is followed by <exit transition area>

<connect list> ::=
<state exit point list>
| <asterisk connect list>

<state exit point list> ::=
{ <state exit point> | **default** } { , { <state exit point> | **default**}*

<asterisk connect list> ::=
<asterisk> [**(** <state exit point list> **)**]

<exit transition area> ::=
<transition area>

至多带一个<state exit point>的<connect area>代表一个*Connect-node*。如果没有提供<connect list>，那么省略*State-exit-point-name*。

<connect list>必须只能指的是可见的<state exit point>。

语义

*Connect-node*代表复合状态中的一个出口点。如果在*Composite-state-graph*中将*Return-node*解释为*Connect-node*的一部分（*Return-node*用于描述*State-exit-point-name*集中的*State-exit-point-definition*），那么在该点进行解释。

不带*State-exit-point-name*的*Connect-node*对应复合状态中一个没有标签的*Return-node*。

模型

<state exit point list>中的**default**代表一个没有标签的<return area>。

当某个<connect area>的<connect list>包含多个<state exit point name>时，为每个这样的<state exit point name>创建一个<connect area>拷贝。而后用这些拷贝替换<connect area>。

将含有<asterisk connect list>的<connect list>转化为<state exit point>列表，每个对应一个在议的<composite state area>的<state exit point>（包括没有标签的<return area>），在<asterisk>后的圆括号中提到的那些除外。而后按上面所述对<state exit point>列表进行转化。

11.12 转换

11.12.1 转换体

抽象语法

Transition :: *Graph-node**
(*Terminator* | *Decision-node*)

Graph-node :: (*Task-node*
| *Output-node*
| *Create-request-node*
| *Call-node*
| *Compound-node*
| *Set-node*
| *Reset-node*) [*On-exception*]

Terminator :: (*Nextstate-node*
| *Stop-node*
| *Return-node*
| *Join-node*
| *Continue-node*
| *Break-node*
| *Raise-node*) [*On-exception*]

具体语法

<transition area> ::=
[<transition string area> ***is followed by***]
<terminator area>

<terminator area> ::=

<state area>
| <nextstate area>
| <decision area>
| <stop symbol>
| <merge area>
| <out connector area>
| <return area>
| <transition option area>
| <raise area>

<transition string area> ::=
<action area>
[***is followed by*** <transition string area>]

<action area> ::=
<task area>
| <output area>
| <create request area>
| <procedure call area>
| <remote procedure call area>

转换由一系列将由代理执行的动作组成。

<transition area>代表*Transition*，<transition string area>代表*Graph-node*列表。

<operation body area>中的<transition area>不得包含<state area>或<nextstate area>。

语义

转换执行一系列动作。在一次转换期间，可以处理代理的数据，并可以输出信号。转换将以进入一个状态的代理的状态机结束，以停止、返回或将控制转给另一个转换来结束。

可以同时将一个功能块某个进程中的一个转换解释为同一功能块或另一功能块另一个进程中的一个转换（假设它们都不被一个进程所包封）。一个进程中各进程的转换被认为是交叉的，也就是说，在某个时刻，只有一个所含的进程在对一个转换进行解释，直至它进入下一个状态（完成）。SDL系统解释的一个合法模型是，在所有动作层面，不同的进程是完全交叉的，动作不能转化为（通过本建议书有关模型一节中给出的规则）其他动作，并不得排除在外，原因是，它们处在一个转换中，该转换与另一个正在被解释的转换交叉（见第9.3节）。

在对一个动作进行解释之时，可以经过一段长短不定的时间。每次在对动作进行解释时，所用时间不同是合法的。每次在对动作进行解释时，所用时间相同或为零（也就是说，结果为**now**，见第12.3.4.1节，未发生变化）也是合法的。

模型

根据有关<import expression>（见第10.6节）和远程过程调用（见第10.5节）的转化规则，一个转换动作可以被转化为一个动作列表（可能包含隐性状态）。为保留与最初动作、终结符或决定相关的异常句柄，需要将该动作列表匿名地封装在一个新的、隐含定义的过程中，以一个带动作列表的<start area>作为它的<transition area>。

通过调用将旧的动作替换为该匿名过程。如果有一个异常句柄与最初的动作相关，那么异常句柄与有关该匿名定义的过程调用相关。

如果转化的构件出现在终结符区域或决定区域，那么通过有关该匿名过程的调用替换最初的终结符区域或决定区域，后跟新的终结符区域或决定区域。如果一个异常句柄与最初的终结符区域或决定区域相关，那么异常句柄与有关该匿名过程的调用相关，并与新的终结符区域或决定区域相关。

没有异常句柄与匿名过程体相关，或与本体的任何部分相关。

11.12.2 转换终结符

11.12.2.1 Nextstate

抽象语法

Nextstate-node :: *State-name*
[*Nextstate-parameters*]

Nextstate-parameters :: [*Expression*]*
[*State-entry-point-name*]
[***HISTORY***]

只有当*State-name*表示一个复合状态时，*Nextstate-parameters*才能出现。

在nextstate中规定的*State-name*必须是相同*State-transition-graph* 或 *Procedure-graph*中的状态名称，

具体语法

<nextstate area> ::=
<state symbol> ***contains*** <nextstate body>

<nextstate body>::=
<state name> [<actual parameters>] [**via** <state entry point name>]
| <dash nextstate>

<dash nextstate> ::=
<hyphen>
| <history dash nextstate>

<history dash nextstate> ::=
<history dash sign>

带**HISTORY**的*Nextstate-node*代表一个<history dash nextstate>。

如果转换由<history dash nextstate>终结，那么<state area>必须是一个<composite state area>。

如果给出了<state entry point name>，那么<nextstate area>必须指的是一个带状态入口点的复合状态。

如果给出了<actual parameters>，那么<nextstate area>必须指的是一个带<agent formal parameters>的复合状态。

包含在<start area>中的<transition area>不得直接或间接地导致一个<dash nextstate>。包含在<start area>或<handle area>中的<transition area>不得直接或间接地导致一个<history dash nextstate>。

<start area>内的或与整个body相关的<on exception association area>不得直接或间接地（通过<exception handler area>内的<on exception association area>）导致一个包含<dash nextstate>的<exception handler area>。

语义

nextstate表示转换的一个终结符。当终结转换时，它规定代理、过程或复合状态的状态。

复合状态的破折线nextstate意味着下一个状态是复合状态。

如果给出了*State-entry-point-name*，那么下一个状态是一个复合状态，并且用*State-start-node*继续进行解释，*State-start-node*在*Composite-state-graph*中具有相同的名称。

当对一个带**HISTORY**的*Nextstate-node*进行解释时，下一个状态是这样一个状态，在其中激活当前转换。如果解释重新进入一个复合状态，那么调用其入口过程。

模型

在<state area>的每个<nextstate area>中，用<state area>的<state name>替换<dash nextstate>。在转化<state area>转化和所有其他转化后应用该模型，那些有关结尾逗号、同义词、优先级输入、连续信号、使能条件、祈使动作的隐含任务以及远程变量或过程的转化除外。

本模型的其他部分描述如何确定异常句柄中的<dash nextstate>含义。

如果异常句柄与状态或异常句柄、附于状态或异常句柄的激励相关，或者如果异常句柄与激励之后的转换动作相关，那么异常句柄称为是可达的。所有从异常句柄（它从状态可达）可达的异常句柄也称为从状态是可达的。

注 — 可达性是可转换的。

对每个<state area>，应用下列规则：通过将每个异常句柄用一个新的名称拷贝至<exception handler area>，为状态区分所有可达的异常句柄。利用该新的名称对<on exception association area>进行修改。然后，移去从任何状态都不可达的异常句柄。

在该替换后，可以确切地、直接或间接地从某个＜state area＞到达一个包含＜dash nextstate＞的给定＜exception handler area＞。用该＜state area＞的＜state name＞替换每个这样＜exception handler area＞中的＜dash nextstate＞。

11.12.2.2 汇接

通过表明下一个待解释的＜action area＞含有相同的＜connector name＞，汇接改变体中的流程。

抽象语法

Join-node :: *Connector-name*

具体语法

＜merge area＞ ::=
＜merge symbol＞ ***is connected to*** ＜flow line symbol＞

＜merge symbol＞ ::=
＜flow line symbol＞

＜flow line symbol＞ ::=
——————

＜out connector area＞ ::=
＜out connector symbol＞ ***contains*** ＜connector name＞

＜out connector symbol＞ ::=
＜in connector symbol＞

对body区域（＜agent body area＞、＜composite state body area＞、＜exception handler body area＞、＜operation body area＞ 或 ＜procedure body area＞）中的每个＜out connector area＞，在该body区域中必须确切地存在一个＜in connector area＞，带有相同的＜connector name＞。

如果＜merge area＞包括在＜transition area＞中，那么它等同于在＜transition area＞中规定一个＜out connector area＞，它包含一个惟一的＜connector name＞，并附有一个＜in connector area＞，相对＜merge area＞中的＜flow line symbol＞，带有相同的＜connector name＞。

语义

当解释一个*Join-node*时，解释以*Free-action*继续，*Free-action*以*Connector-name*命名。

11.12.2.3 停止

抽象语法

Stop-node :: ( )

具体语法

＜stop symbol＞ ::=

＜stop symbol＞代表一个*Stop-node*。

语义

停止使解释它的代理执行一个停止。

这意味着抛弃输入端口中的保留信号，代理的状态机进入停止状态。当所有包含的代理都不再存在时，代理自身也将不再存在。

解释*Procedure-graph* 或 *State-transition-graph*中的*Stop-node*将造成代理（负责解释该*Procedure-graph*）停止。解释过程、操作、复合语句或复合状态将终止和停止向调用者传播，并当做是仿佛在过程调用、操作应用、调用复合语句或进入复合状态之处对*Stop-node*进行了解释。向外传播终结，直至到达包含的代理。

11.12.2.4 返回

抽象语法

Return-node = *Action-return-node*
| *Value-return-node*
| *Named-return-node*
Action-return-node :: ( )
Value-return-node :: *Expression*
Named-return-node :: *State-exit-point-name*

*Action-return-node*必须只能包含在不带*Result*或*Composite-state-graph*的*Procedure-definition*的*Procedure-Graph*中。*Value-return-node*必须只能包含在含有*Result*的*Procedure-definition*的*Procedure-Graph*中。*Named-return-node*必须只能包含在*Composite-state-graph*中。

*Value-return-node*的*Expression*必须在类别上兼容于嵌套*Procedure*的*Result*的类别。

具体语法

<return area> ::=
<return symbol>
[***is connected to*** <on exception association area>]
[***is associated with*** <return body>]

<return body> ::=
<expression>
| {**via** <state exit point>}

<return symbol> ::=

当且仅当嵌套的范围是一个运算符、方法或过程（有一个<procedure result>）时，才允许<return area>中的<expression>。

当且仅当嵌套的范围是一个复合状态（含有规定的<state exit point>）时，才允许<state exit point>。

如果嵌套的范围是一个带<procedure result>的运算符或方法或者是一个带<procedure result>而不带<variable name>的值返回过程时，不得省略<return area>中的<expression>。

注 — 如果在带<procedure result>的运算符或方法中或者在带命名之<procedure result>的值返回过程中省略了<expression>，那么第9.4节中的模型将增加过程结果变量作为<expression>。

语义

以下列方式对过程中的*Return-node*进行解释：

a） 由*Procedure-start-node*的解释所创建的所有变量将不再存在。

b） 完成对*Procedure-graph*的解释，过程实例不再存在。

c） 如果解释*Value-return-node*，那么以*Expression*同样的方式对*Expression*的结果进行解释，用结果类别将*Expression*指配给一个变量（见第12.3.3节），而不用与变量相关的结果或正在进行的范围检查；而后将对象或值结果返回给调用正文。

d） 之后，在该调用之后的节点处继续对调用正文进行解释。

复合状态中的*Return-node*将激活*Connect-node*。对*Named-return-node*，用相同的名称，在*Connect-node*处继续进行解释。对*Action-return-node*，无需名称，在*Connect-node*处继续进行解释。

11.12.2.5 引起

抽象语法

Raise-node :: *Exception-identifier*
[*Expression*]*

可选*Expression*列表的长度必须等同于异常定义中的*Sort-reference-identifier*数量，异常定义由*Exception-identifier*表示。

每个*Expression*都必须要有一个类别，它兼容于异常定义中对应（按位置）的*Sort-reference-identifier*。

具体语法

<raise area> ::=
<raise symbol> ***contains*** <raise body>

<raise symbol> ::=

<raise body> ::=
<exception raise>

<exception raise> ::=
<exception identifier> [<actual parameters>]

一个<raise body>代表一个*Raise-node*。

语义

解释*Raise-node*创建一个异常实例（见第11.6节，有关异常实例的解释）。通过异常实例传送的数据项是<raise body>实际参数的结果。如果省略了可选表达式列表中的表达式（也就是说，如果省略了<actual parameters>中的对应<expression>），那么没有数据项是用异常实例的对应位置进行传送的，也就是说，对应位置是“未定义的”。

如果在异常定义中规定共型，并且在<raise body>中规定表达式，那么在第12.1.9.5节中定义的范围检查适用于表达式。

模型

根据（例如，远程过程调用的）模型，<raise area>可以被转化成动作列表（可能包含隐性状态）加一个新的<raise area>。而后应用第11.12.1节中的转换终结符模型。

11.13 动作

11.13.1 任务

抽象语法

Task-node = *Assignment*
| *Assignment-attempt*
| *Informal-text*

具体语法

<task area> ::=
<task symbol> ***contains*** <task body>
[***is connected to*** <on exception association area>]
| <macro symbol> ***contains*** { <macro name> [<macro call body>] }

<task body> ::=
<statement list>
| <informal text>

<task symbol> ::=

<macro symbol> ::=

可以省略<task body>的<statement list>中的结尾<end>。

在其<statement list>中含有一个<statement>的<task area>代表任何该语句所代表的内容。含有任何其他<statement list>的<task area>代表一个*Compound-node*。*Connector-name*通过最近常见的匿名名称来表示。*Variable-definition-**set*** 通过<statement list>中所有<variable definition>的列表来表示。如果*Transition*通过<statement list>中的<statements>来表示，或者如果<statement list>未终结，那么通过<statement list>中的<statements>来表示，后跟带*Connector-name*的*Break-node*（见第11.14节）。

语义

Task-node 的解释就是*Assignment*、*Assignment-attempt* 的解释或者是*Informal-text*的解释。

对*Compound-node*的解释在第11.14.1节中给出。对*Assignment*和*Assignment-attempt*的解释在第12.3.3节中给出。

一个任务区域创建其自身的范围。

模型

如果<task body>的<statement list>为空，那么移去<task area>。任何导致这样一个空<task area>的语法项都将直接导致在<task area>后的项。

如果存在的话，由<macro symbol>定义的<task area>转化为由<task symbol>定义的<task area>，<task symbol>包含一个<macro call>，它具有相同的<macro name>和<macro call body>。

11.13.2 创建

抽象语法

Create-request-node :: [*Variable-identifier*]
Agent-identifier
[*Expression*]*

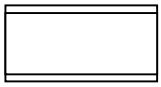
可选表达式列表的长度必须与*Agent-identifier*的*Agent-definition*中*Agent-formal-parameter*的数量相同。

按位置与*Agent-formal-parameter*对应的每个表达式必须拥有一个兼容于*Agent-definition*（由*Agent-identifier*定义）中*Agent-formal-parameter*之类别的类别。

具体语法

<create request area> ::=
<create request symbol> ***contains*** <create body>
[***is connected to*** <on exception association area>]

<create request symbol> ::=



<create body> ::=
{ <agent identifier> | <agent type identifier> | **this** } [<actual parameters>]

<actual parameters> ::=
(<actual parameter list> **)**

<actual parameter list> ::=
[<expression>] { **,** [<expression>]}*

可以省略<actual parameter list>中最后<expression>之后的逗号。

this应只在<agent type diagram>中和由<agent type diagram>嵌套的范围中进行规定。

一个<create request area>代表一个*Create-request-node*。

语义

创建动作将创建一个由*Agent-identifier*标识的集合的实例，在执行创建之代理的内部，或者在包含执行创建之代理的代理的内部。所建代理的**parent**（见第9节，模型）的pid与创建代理的**self**返回的pid相同。所建代理的**self**（见第9节，模型）和创建代理的子孙（见第9节，模型）拥有相同的、惟一的、新的pid。

当创建一个代理实例时，提供的是一个空的输入端口，创建变量，并按给定的次序对实际参数表达式进行解释，并指配（如第12.3.3节所定义）给对应的形式参数。如果所建代理包含代理结合，那么创建这些集合的最初实例。而后通过解释代理图中的起始节点启动代理，并在对由信号引起的转换进行解释之前，以某种次序对最初所含代理的起始节点进行解释。

而后对所建代理异步地和并发地进行解释，或依据包含代理（系统、功能块、进程）的种类与其他代理进行交互。

如果尝试创建比代理定义中的实例最多数量还多的代理实例，那么不创建新的实例，创建代理的子孙（见第9节，模型）其结果为Null，并且继续进行解释。

如果省略<actual parameters>中<expression>的，那么对应的形式参数没有相关的数据项；也就是说，它是“未定义的”。

如果<agent type identifier>用在<create body>中，那么对应的代理类型不可以被定义为<abstract>，或包含形式正文参数。

如果具有相同名称的实例集和代理类型都在一个范围单元中进行定义，并且在该范围单元中的创建语句也使用该名称，那么在该实例集中创建一个实例，并且不基于代理类型。注意，通过定义一个基于代理类型的实例集来创建代理类型的一个实例，而后创建在该集合中的一个实例，是可能的。

模型

声明**this**是隐性<process identifier>的衍生语法，<process identifier>用于标识代理的实例集，在其中对创建进行解释。

如果<agent type identifier>用在<create request area>中，那么下列模型适用：

a） 如果在包含实例（用于执行创建）的代理中存在一个所标明之代理类型的实例集，那么<agent type identifier>是衍生语法，用于表示该实例集。

b） 如果存在多个实例集，那么在解释之时确定在哪个集合中创建实例。在这种情况下，由一个使用**any**的非确定性决定来替换<create request area>，对每个实例集，后跟一个分支。在每个分支中，插入一个有关对应实例的创建请求。

c） 如果在包含代理中没有所标明之代理类型的实例集，那么：

i） 在包含代理中创建一个具有惟一名称的、有关给定类型的隐性实例；并且

ii）<create request area>中的<agent type identifier>是有关该隐性实例集的衍生语法。

11.13.3 过程调用

抽象语法

Call-node	::	*Procedure-identifier* [*Expression*]*
Value-returning-call-node	::	*Procedure-identifier* [*Expression*]*

可选*Expression*的列表长度必须等同于由*Procedure-identifier*表示的*Procedure-definition*中的*Procedure-formal-parameter*s数量。

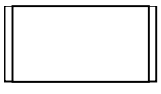
通过位置对应一个*In-parameter*的每个 *Expression*在类别上都必须兼容于*Procedure-formal-parameter*的类别。

通过位置对应一个*Inout-parameter*或*Out-parameter*的每个*Expression*都必须是一个带有相同*Sort-reference-identifier*的*Variable-identifier*，作为*Procedure-formal-parameter*。

具体语法

<procedure call area> ::=
<procedure call symbol> ***contains*** <procedure call body>
[***is connected to*** <on exception association area>]

<procedure call symbol> ::=



<procedure call body> ::=

[**this**] { <procedure identifier> | <procedure type expression> } [<actual parameters>]

不可以省略对应形式参数的、<actual parameters>中的<expression>，并且必须是一个<variable access>或<extended primary>。

在应用**this**的模型后，<procedure identifier>必须表示一个包含起始转换的过程。

如果使用**this**，那么<procedure identifier>必须表示一个嵌套的过程。

一个<procedure call area>代表一个*Call-node*。一个<value returning procedure call>（见第12.3.5节）代表一个*Value-returning-call-node*。

语义

对过程*Call-node*或*Value-returning-call-node*的解释用于按给定次序对实际参数表达式进行解释。如果参数解释没有引起异常，那么将解释传送给通过*Procedure-identifier*引用的过程定义，并且对该过程图进行解释（在第9.4节中含有说明）。

如果省略<actual parameters>中的<expression>，那么对应的实际参数没有相关的数据项；也就是说，它是“未定义的”。

如果过程*In-parameter* 或 *Inout-parameter*的*Call-node* 或 *Value-returning-call-node*变元类别为共型，那么在第12.1.9.5节中定义的范围检查适用于表达式结果。在解释时，如果范围检查的结果为预定义的布尔值“假”，那么引起预定义的异常OutOfRange（见D.3.16），而不解释更多的实际参数或过程定义。

如果未引起OutOfRange，那么当对所调用之过程的解释结束时，继续进行对含有*Call-node*之转换的解释。

如果未引起OutOfRange，那么当对所调用之过程的解释结束时，继续进行对含有*Value-returning-call-node*之转换的解释。所调用过程的结果通过*Value-returning-call-node*来返回。

*Value-returning-call-node*有一个类别，它为通过对过程的解释而获得的结果的类别。

如果值返回过程调用的结果类别是共型，那么在第12.1.9.5节中定义的范围检查适用于过程调用结果。在解释之时，如果范围检查的结果为预定义的布尔值“假”，那么引起预定义的异常OutOfRange（见D.3.16）。

模型

如果不在嵌套代理中定义<procedure identifier>，那么将过程调用转化成一个过程的局部、隐性创建共型的调用。

this意味着，当过程特殊化时，用特殊化过程的标识符来替换<procedure identifier>。

11.13.4 输出

抽象语法

Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*] *Direct-via*
Signal-destination	=	*Expression* | *Agent-identifier*
Direct-via	=	(*Channel-identifier* | *Gate-identifier*)***-set***
Channel-identifier	=	*Identifier*

可选表达式列表的长度必须与由*Signal-identifier*表示的*Signal-definition*中的*Sort-reference-identifier*数量相同。

每个表达式在类别上必须兼容于*Signal-definition*中对应的*Sort-identifier-reference*（按位置）。

对*Direct-via*中的每个*Channel-identifier*，必须存在零个或多个信道，这样，经由该路径的信号用*Signal-identifier*从代理是可达的，并且在自代理的方向上的*Channel-path*在其集中必须包括*Signal-identifier*。

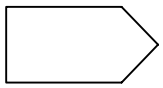
对*Direct-via*中的每个*Gate-identifier*，必须存在零个或多个信道，这样，经由该路径的通道用*Signal-identifier*从代理是可达的，并且通道的*Out-signal-identifier-**set***必须包括*Signal-identifier*。

具体语法

<output area> ::=
<output symbol> ***contains*** <output body>
[***is connected to*** <on exception association area>]

<output symbol> ::=
<plain output symbol>
| <internal output symbol>

<plain output symbol> ::=



<internal output symbol>::=

注1 — <plain output symbol>和<internal output symbol>在含义上没有区别。

<output body> ::=
<signal identifier> [<actual parameters>] {，<signal identifier> [<actual parameters>] }*
<communication constraints>

<destination> ::=
<pid expression0> | <agent identifier> | **this**

<via path> ::=
via { <channel identifier> | <gate identifier> }

<destination>中的<pid expression0>或<agent identifier>代表*Signal-destination*。在<destination>中的<pid expression0>与<agent identifier>之间存在语法上的不确定性。如果<destination>可解释为一个不与任何静态条件存在冲突的<pid expression0>，那么它可以被解释为<pid expression0>，否则被解释为<agent identifier>。<agent identifier>必须表示一个代理，它自起始的代理是可达的。

在代理类型的状态机的<output body>中提到的信号必须处于代理类型的完全合法输入信号集中，或处于自代理类型方向上的通道的<signal list>中。

在<output body>中的<communication constraints>（见第10.5节）不应包含**timer <**timer identifier>子句。它至多包含一个**to** <destination>子句和零个或多个**<**via path>。

<communication constraints>的每个<via path>代表一个*Direct-via*中的*Channel-identifier* 或 *Gate-identifier*。

this只能在<agent type diagram>中以及在由<agent type diagram>嵌套的范围中进行规定。

如果<destination>时一个带静态类别而不是Pid的<pid expression0>（见第12.1.6节），那么<signal identifier>必须代表一个由定义pid类别之接口定义或使用的信号。

<via path>中的<gate identifier>可以用于标识一个利用<interface gate definition>进行定义的通道。

语义

声明*Signal-destination*中的*Agent-identifier*用于指明将*Signal-destination*作为代理实例集（通过*Agent-identifier*指明）的任何现有实例。如果没有实例存在，那么抛弃信号。

如果在*Direct-via*中没有规定*Channel-identifier* 或 *Gate-identifier*，并且没有规定*Signal-destination*，那么任何存在一条通信路径的代理都可以接收信号。

如果存在一个既包含发送者又包含接收者的进程实例，那么通过信号实例传送的数据项为输出的实例参数结果。否则，通过信号实例传送的数据项为输出的实例参数结果的新建拷贝，并且不与输出的实例参数结果共享引用。当在实际参数结果中存在引用周期时，传送的数据项也将包含这些周期。每个传送的数据项都将等同于对应的输出的实际参数。

如果省略<actual parameters>中的<expression>，那么没有用信号实例的对应位置传送任何数据项；也就是说，对应位置是“未定义的”。

起始代理的pid也通过信号实例进行传送。

如果在信号定义中规定了共型，并且在输出中规定了表达式，那么在第12.1.9.5节中定义的范围检查适用于表达式。

如果<destination>是一个<pid expression0>，并且pid表达式的静态类别是Pid，那么为用*Signal-identifier*表示的信号执行有关pid表达式动态类别的兼容性检查（见第12.1.6节）。

而后将信号实例传递给一条能够传送它的通信路径。能够传送信号实例的通信路径集可以通过<via path>子句进行限制，以便包括至少一条在*Direct-via*中提到的路径。

如果是*Signal-destination*一个表达式，那么信号实例传递给用表达式表示的代理实例。如果该实例不存在或从起始代理不可达，那么抛弃信号实例。

如果*Signal-destination*是一个*Agent-identifier*，那么信号实例传递给用*Agent-identifier*表示的代理实例集的任意一个实例。如果不存在这样的实例，那么抛弃信号实例。

注2 — 如果在*Output-node* 中*Signal-destination*为Null，那么当对*Output-node* 进行解释时，将抛弃信号实例。

如果没有规定任何*Signal-destination*，那么分两步选择接收者。第一步，信号传送给一个代理实例集，通过能够传送信号实例的通信路径可以到达它。该信号实例集是任意选择的。第二步，当信号实例到达通信路径的末端时，将它传递给代理实例集的一个实例。该实例是任意选择的。如果没有实例可以选择，那么抛弃信号实例。

注意，在两个*Output-node*的*Direct-via*中规定相同的*Channel-identifier* 或 *Gate-identifier*不自动地表示，在对*Output-node*进行解释时，以相同的次序在输入端口中对信号进行排队。不过，如果通过同样的延迟信道来传送两个信号，或只通过没有延迟的信道来传送，那么保留次序。

模型

如果在一个<output body>中规定了若干对<signal identifier>和<actual parameters>，那么这是衍生语法，用于规定一个<output body>序列（分别在<output area>或<output statement>中），次序同初始<output body>中规定的次序，每个包含一对<signal identifier> 和 <actual parameters>。在每个<output body>中，重复**to** <destination>子句和<via path>。

在<destination>中声明**this**是有关隐性<agent identifier>的衍生语法，用于标识代理的实例集，在其中对输出进行解释。

11.13.5 决定

抽象语法

Decision-node	::	*Decision-question* [*On-exception*] *Decision-answer-**set*** [*Else-answer*]
Decision-question	=	*Expression*
	|	*Informal-text*
Decision-answer	::	(*Range-condition* | *Informal-text*) *Transition*
Else-answer	::	*Transition*

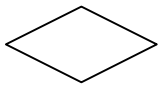
*Range-condition*的*Constant-expression*必须是一个兼容的类别。如果*Decision-question*是一个表达式，那么*Decision-answer*的*Range-condition*必须在类别上兼容于*Decision-question*的类别。

具体语法

<decision area> ::=

<decision symbol> ***contains*** <question>
[***is connected to*** <on exception association area>]
is followed by <decision body>

<decision symbol> ::=



<question> ::=

<expression> | <informal text> | **any**

<decision body> ::=

{ <answer part>+ [<else part>] } ***set***

<answer part> ::=

<flow line symbol> ***is associated with*** <graphical answer>
is followed by <transition area>

<graphical answer> ::=

[<answer>] | **(** [<answer>] **)**

<answer> ::=

<range condition> | <informal text>

<else part> ::=

<flow line symbol> ***is associated with*** **else**
is followed by <transition area>

<graphical answer>和**else**可以随相关的<flow line symbol>放置，或覆盖<flow line symbol>。

源自<decision symbol>的<flow line symbol>可以拥有一条公共的起始路径。

一个<decision area>代表一个*Decision-node*。

当且仅当<question>包括关键字**any**时，必须省略<graphical answer>的<answer>。在这种情况下，不可以出现<else part>。

语义

决定将解释传送给对外路径，其*Range-condition*包含由问题解释给出的结果。以任意次序，为每个*Decision-answer*完成一次决定，以确定*Decision-question*是否包含在每个*Decision-answer*中，直至确定包含*Decision-question*的*Range-condition*，或直至该决定要求对引起异常或选择*Informal-text*的操作应用做出解释。决定一系列可能的问题回答，每个回答用于规定有关该路径选择的待解释动作集。

一个问题可以是其他问题的补充。这通过规定*Else-answer*来实现，它指明当问题所在的表达式结果不被其他问题规定的结果所覆盖时的待执行的活动集。

无论何时当不规定*Else-answer*时，并且来自问题表达式的评估结果不与其中一个问题匹配，那么引起预定义的异常NoMatchingAnswer。

在<question>和<answer>中，<informal text> 与 <character string>之间存在不确定性。如果<question>和所有<answer>都是<character string>，那么所有这些都被解释为<informal text>。如果<question>或某个<answer>是一个<character string>，并且它与决定正文不匹配，那么<character string>表示<informal text>。

在确定决定的正文（即类别）时，无需考虑为<character string>的<answer>。

模型

只使用<decision area>中的**any**，是一种有关使用决定中<any expression>的缩写形式。假定<decision body>由N个<answer part>组成，那么<decision area>中的**any**是书写**any**(data_type_N)的一种缩写形式，其中data_type_N是一个匿名共型，定义如下：

```
syntype data_type_N =
   <<package Predefined>> Integer { constants 1:N; }
```

省略的<graphical answer>是书写文字1到N的缩写形式，作为N 个<graphical answer>中<range condition>的<constant>。

11.14 语句列表

语句列表可以用在<task area>、<procedure definition>或<operation definition>中，用于定义语句列表的局部变量，以及众多待解释动作的局部变量。语句列表的目的是为了允许对算法做出简明的文本描述，以便与图形SDL形式结合起来。根据下列模型，通过语句转化来确定语句列表的语义，从而可以从左到右、有效地对语句做出解释。

变量定义语句可以在<statement list>开始之处引入变量。与第12.3.1节中的<variable definition>相反，在这一正文中，变量的初始化不要求是一个<constant expression>。

具体语法

<statement list> ::=
<variable definitions> <statements>

<variable definitions> ::=
{ <variable definition statement> }*

<statements> ::=
<statement>*

<statement> ::=
<empty statement>
| <compound statement>
| <assignment statement>
| <output statement>
| <create statement>
| <set statement>
| <reset statement>
| <export statement>
| <call statement>
| <expression statement>
| <if statement>
| <decision statement>
| <loop statement>
| <terminating statement>
| <labelled statement>
| <exception statement>

<terminating statement> ::=
<return statement>
<stop statement>
| <break statement>
| <loop break statement>
| <loop continue statement>
| <raise statement>

<loop break statement>和<loop continue statement>只能出现在<loop statement>中。

<terminating statement>只能作为<statements>中的最后一个<statement>出现。如果<statement list>中的最后一个<statement>是一个<terminating statement>，那么<statement list>是终结的。

<variable definition statement> ::=
dcl <local variables of sort> { **,** <local variables of sort> }* <end>

<local variables of sort> ::=
<variable name> { **,** <variable name>}* <sort> [<is assigned sign> <expression>]

一个<statement list>代表一个*Graph-nodes*列表。

模型

如果<statement list>包含<variable definitions>，那么为每个<variable definition statement>执行下列内容。为<variable definition statement>中的每个<variable name>创建一个新的<variable name>。用对应的新建<variable name>替换在后随的<variable definition statement>中和在<statements>中的<variable name>的每个事件。

对每个＜variable definition statement＞，通过省略初始化＜expression＞（如果存在的话）来从＜variable definition statement＞形成一个＜variable definition＞，并在初始＜variable definition statement＞处作为一个＜variable definition statement＞进行插入。如果初始化＜expression＞存在，那么按其出现的次序，为在＜local variables of sort＞中提到的每个＜variable name＞构建一个＜assignment statement＞，其中为＜variable name＞提供＜expression＞的结果。按其出现的次序，在＜statements＞之前，将这些＜assignment statement＞插入。

注 — 如果＜statement list＞为空，那么用一个*Break-node*来表示它，如第9.4节和第11.14.1节中的具体句法所述。

11.14.1 混合语句

多个语句可以合并成一个语句。

抽象语法

Compound-node	::	*Connector-name* *Variable-definition-**set*** [*Exception-handler-node*] *Init-graph-node** *Transition* *Step-graph-node**
Init-graph-node	=	*Graph-node*
Step-graph-node	=	*Graph-node*
Continue-node	::	*Connector-name*
Break-node	::	*Connector-name*

具体语法

＜compound statement＞ ::=
[＜comment body＞] ＜left curly bracket＞ ＜statement list＞ ＜right curly bracket＞

＜compound statement＞代表一个*Compound-node*。用最新创建的匿名名称来表示*Connector-name*。用＜statement list＞中所有＜variable definition＞的列表来表示*Variable-definition-**set***。用＜statement list＞中＜statements＞的转化来表示转换，或者如果＜statement list＞不终结，那么用后跟一个带*Connector-name*的*Break-node*的、＜statement list＞中＜statements＞的转化来表示转换。

语义

＜compound statement＞创建其自身的范围。

对*Compound-node*的解释按下列步骤进行：

a） 为*Variable-definition-**set***中的每个*Variable-definition*创建局部变量。

b） 对*Init-graph-node*列表进行解释。

c） 对*Transition*进行解释。

d） 当对带*Connector-name*的*Continue-node*匹配*Connector-name*进行解释时，对*Step-graph-node*列表进行解释，并在步骤c）中继续做进一步解释。

e） 当对*Compound-node*的解释终结时，通过解释创建的所有变量将不再存在。在下列条件下，对*Compound-node*的解释终结：

i） 当对*Break-node*进行解释时；或

ii） 当对不同于*Compound-node*中*Connector-name*的、带*Connector-name*的*Continue-node*进行解释时；或

iii） 当对*Return-node*进行解释时；或

iv） 当创建异常实例时，它不在*Compound-node*的转换中进行处理。

f） 之后，按下列步骤继续进行解释：

i） 如果因解释带*Connector-name*的*Break-node*匹配*Connector-name*而终结对*Compound-node*的解释，那么在*Compound-node*之后的节点处继续进行解释；否则

ii） 如果因解释*Break-node*、*Continue-node*或*Return-node*而终结对*Compound-node*的解释，那么在*Compound-node*调用点处，分别以对*Break-node*、*Continue-node*或*Return-node*的解释，来继续进行解释；否则

iii） 如果因创建一个异常实例而终结对*Compound-node* 的解释，那么继续进行解释，如第11.16节所述。

11.14.2 作为语句的转换行动和终结符

在一个语句列表中，**task**关键字不得在指配语句之前，过程调用无需**call**关键字。类似<action area>中那些构件（见第11.12.1节）的各构件以及<terminator area>中的某些构件（见第11.12.1节），可以被用做<statement list>中的<statement>。

具体语法

<assignment statement> ::=
<assignment> <end>

<output statement> ::=
output <output body> <end>

<create statement> ::=
create <create body> <end>

<set statement> ::=
set <set body> <end>

<reset statement> ::=
reset <reset body> <end>

<export statement> ::=
export <export body> <end>

<return statement> ::=
return [<return body>] <end>

<stop statement> ::=
stop <end>

<raise statement> ::=
raise <raise body> <end>

<call statement> ::=
[**call**] { <procedure call body> | <remote procedure call body> } <end>

<assignment statement>代表一个*Assignment*或*Assignment-attempt*。

<output statement>代表一个*Output-node*，在第11.13.4节中有更多的描述。

<create statement>代表一个*Create-request-node*，在第11.13.2节中有更多的描述。

<set statement>代表一个*Set-node*，在第11.15节中有更多的描述。

<reset statement>代表一个*Reset-node*，在第11.15节中有更多的描述。

<return statement>只允许出现在一个<procedure definition>或一个<operation definition>内。

<return statement>代表一个*Return-node*，在第11.12.2.4节中有更多的描述。

<stop statement>代表一个*Stop-node*。

<raise statement>代表一个*Raise-node*，在第11.12.2.5节中有更多的描述。

如果<call statement>在语法上与具有相同名称的操作应用或变量不明确，那么不可以省略关键字**call**。

注 — 不通过正文来解决该不确定性。

<call statement>代表一个*Call-node*，在第11.13.3节中有更多的描述。

注 — <export statement>模型在第10.6节中给出。

11.14.3 作为语句的表达式

可以将为操作应用的表达式用做语句，在这种情况下，对<operation application>进行解释，并忽略结果。

具体语法

<expression statement> ::=
<operation application> <end>

模型

将<expression statement>转化为<call statement>，其中<procedure call body>构建自<operation identifier>和<operation application>的<actual parameters>。

11.14.4 如果语句

对<Boolean expression>进行解释，并且如果返回预定义的布尔值“真”，那么对<consequence statement>进行解释；否则，如果<alternative statement>出现，那么对它进行解释。

具体语法

<if statement> ::=
if (<Boolean expression> **)** <consequence statement>
[**else** <alternative statement>]

<consequence statement> ::=
<statement>

<alternative statement> ::=
<statement>

<alternative statement>与最靠前的<consequence statement>相关。

模型

<if statement>等同于下列<decision statement>：

decision <Boolean expression> {
(true **) :** <consequence statement>
(false **) :** <alternative statement>
}

如果<alternative statement>不出现，那么在其位置处插入一个<empty statement>。而后对<decision statement>进行转化，如第11.14.5节所述。

11.14.5 决定语句

决定语句是决定的简明形式。对<expression>进行评估，对其<range condition>包含表达式结果的<algorithm answer part>进行解释。重叠范围条件是不允许的。不像在<decision area>中（见第11.13.5节），表达式不必匹配其中的一个范围条件。如果不匹配，并且存在一个<alternative statement>，那么对<alternative statement>进行解释。如果不匹配，并且<alternative statement>不存在，那么在<decision statement>之后继续进行解释。

具体语法

<decision statement> ::=
decision (<question> **)** [<comment body>] <left curly bracket>
<decision statement body>
<right curly bracket>

<decision statement body> ::=
<algorithm answer part>+ [<algorithm else part>]

<algorithm answer part> ::=
(<answer> **)** <colon> <statement>

<algorithm else part> ::=
else <colon> <alternative statement>

一个<decision statement>代表一个*Decision-node*，其中<algorithm answer part>每个代表一个*Decision-answer*，<algorithm else part>代表*Else-answer*，如果存在，那么构建自对<statement>的转化。

11.14.6 循环语句

<loop statement>为<loop body statement>的限制或不限制迭代提供了经一般化的便利，循环变量可以是一个任意数。这些变量可以在<loop statement>中进行定义，并按<loop step>规定的方式步进。它们可以用于产生连续的结果，也可以用于积累结果。当<loop statement>终结时，可以在循环变量正文中对<finalization statement>进行解释。

重复地对<loop body statement>进行解释。通过任何<loop clause>的出现来对循环解释进行控制。<loop clause>可以用<loop variable indication>开始，它提供了一种便利的方式来声明和初始化局部循环变量。在<loop variable indication>中定义的任何变量的范围和生命周期为<loop statement>的范围和生命周期。如果初始化作为<loop variable definition>中的一个<expression>出现，那么在第一次解释循环体之前，只对表达式做一次评估。可选地，任何可见变量可以被定义为一个循环变量，并可以拥有一个指配给它的数据项。在每次迭代之前，对所有的<Boolean expression>元素进行评估。如果任何一个<Boolean expression>元素返回“假”，那么终结对<loop statement>的解释。必然地，如果没有出现<Boolean expression>，那么继续进行<loop statement>解释，直至<loop statement>不在本地退出。如果在该<loop clause>中出现一个<loop variable indication>，那么每个循环子句中的<loop step>进行计算，并在每次迭代结束时指配各自循环变量的结果。如果在该<loop clause>中未出现一个<loop variable indication>，或如果未出现一个<loop step>，那么不对循环变量执行指配语句。<loop variable indication>、<Boolean expression>和<loop step>是可选的。循环变量是可见的，但不得在<loop body statement>中进行指配。

当到达一个**break**时，循环体的解释也终结。到达一个**continue**语句将使循环解释立即跳至下一次迭代（还可参见第11.14.7节中的<break statement>）。

如果<loop statement>“正常”终结（也就是说，<Boolean expression>的评估结果为预定义的布尔值“假”），那么对<finalization statement>进行解释。

具体语法

<loop statement> ::=
loop ([<loop clause> { **;** <loop clause> }*] **)**
<loop body statement> [**then** <finalization statement>]

<loop body statement> ::=
<statement>

<finalization statement> ::=
<statement>

<loop clause> ::=
[<loop variable indication>]
, [<Boolean expression>]
<loop step>

<loop step> ::=
[**,** [{ <expression> | [**call**] <procedure call body> }]]

<loop variable indication> ::=
<loop variable definition>
| <variable identifier> [<is assigned sign> <expression>]

<loop variable definition> ::=
dcl <variable name> <sort> <is assigned sign> <expression>

<loop break statement> ::=
break <end>

<loop continue statement> ::=
continue <end>

如果将给具有相同名称的操作应用或变量带来不明确，那么在<loop step>中的关键字**call**不得省略。

<loop step>的<procedure call body>中的<procedure identifier>不可以指的是一个返回过程调用的值。

<finalization statement>与最靠前的<loop body statement>相关。

<loop statement>代表一个*Compound-node*。用最新创建的匿名名称来表示*Connector-name*，指的是标签。

用＜variable definition statement＞列表来表示*Variable-definition-**set***，＜variable definition statement＞构建自每个＜loop variable definition＞中提到的＜variable name＞和＜sort＞。

用＜statement list＞转化来表示*Init-graph-node*列表，按其出现次序，＜statement list＞构建自＜assignment statement＞，＜assignment statement＞形成自每个＜loop variable indication＞。

如果＜loop variable indication＞和＜expression＞都出现，那么＜assignment statement＞构建自＜variable name＞或＜variable identifier＞与＜loop step＞中的＜expression＞之间的每个＜loop clause＞。如果没有构建＜assignment statement＞，那么通过按序提取这些＜assignment statement＞元素，或＜expression＞，或＜loop step＞中的＜procedure call body＞，来构建＜statements＞。＜statements＞代表*Step-graph-node*列表。

用按下列方式构建的＜decision area＞来表示转换：利用＜expression＞中布尔类型的预定义运算符“and”，通过结合所有的＜Boolean expression＞项，获得＜question＞。单个的＜answer part＞包含＜expression＞“真”，作为＜answer＞，并有一个通过转化＜loop body statement＞而获得的＜transition area＞。通过转化＜finalization statement＞来获得＜else part＞的＜transition area＞，并且仅当＜finalization statement＞最初出现时，予以插入。

＜loop continue statement＞代表一个*Continue-node*。*Connector-name*用最内部的嵌套循环语句的标签来表示。

模型

＜loop clause＞内＜loop break statement＞的每个事件，或＜loop body statement＞，或包含在本＜loop statement＞内的另一个＜loop statement＞的＜finalization statement＞，所有都不出现在另一个内部＜loop statement＞中，用下列替换：

break *Label* ;

其中*Label*是所代表之*Compound-node*的最新创建的匿名名称。如果＜Boolean expression＞未出现在＜loop clause＞中，那么插入预定义的布尔值“真”，作为＜Boolean expression＞。

而后用如此修改过的＜loop statement＞替换＜loop statement＞，后跟一个带＜connector name＞ *Break*的＜labelled statement＞。

11.14.7 中断和标签语句

＜break statement＞是一种限制性更强的＜merge area＞形式。

＜break statement＞使解释立即传送给带匹配＜connector name＞的语句之后的那个语句。

具体语法

＜break statement＞ ::=
break ＜connector name＞ ＜end＞

＜labelled statement＞ ::=
＜connector name＞ **:** ＜statement＞

＜break statement＞必须包含在一个用给定的＜connector name＞打上了标签的语句中。

＜break statement＞代表一个带*Connector-name* 的*Break-node* ， *Connector-name*由＜connector name＞表示。

＜labelled statement＞代表一个*Compound-node*。*Transition*由＜statement＞转化结果表示。

11.14.8 空语句

一个语句可以为空，通过使用一个单个的分号来表示。＜empty statement＞没有作用。

具体语法

＜empty statement＞ ::=
＜end＞

模型

＜empty statement＞的转化结果是空文本。

11.14.9 异常语句

在异常句柄内，可以对语句进行封装。

具体语法

<exception statement> ::=
try <try statement> <handle statement>+

<try statement> ::=
<statement>

<handle statement> ::=
handle (<exception stimulus list> **)** <statement>

<handle statement>与最近的、之前的<try statement>相关。<try statement>不得是一个<break statement>。

在*Model*中构建的异常句柄代表*Compound-node*的可选*Exception-handler-node*，*Compound-node*由<compound statement>表示，它自<try statement>得到（见第11.14.1节）。

语义

<exception statement>用一个异常句柄创建其自身的范围。

模型

如果<try statement>不是一个<compound statement>，那么<try statement>首先转化为一个在其<statement list>中只包含<try statement>的<compound statement>。

而后对（已转化的）<try statement>和所有的<handle statement>进行转化。为每个<handle statement>，构建一个<handle area>，其中<transition area>构建自<statement>的转化结果，<exception stimulus list>取自<handle statement>。

构建的<handle area>聚集成一个<exception handler body area>，最后自该<exception handler body area>形成一个<exception handler area>，并给出一个匿名名称。

11.15 计时器

抽象语法

Timer-definition	::	*Timer-name* *Sort-reference-identifier**
Timer-name	=	*Name*
Set-node	::	*Time-expression* *Timer-identifier* *Expression**
Reset-node	::	*Timer-identifier* *Expression**
Timer-identifier	=	*Identifier*
Time-expression	=	*Expression*

Set-node 和 *Reset-node*中的*Expression*列表类别必须按位置对应紧跟*Timer-name*之后的*Sort-reference-identifier*列表，*Timer-name*由*Timer-identifier*标识。

具体语法

<timer definition> ::=
timer
<timer definition item> { **,** <timer definition item>}* <end>

<timer definition item> ::=
<timer name> [<sort list>] [<timer default initialization>]

<timer default initialization> ::=
<is assigned sign> <Duration constant expression>

<reset body> ::=
(<reset clause> { **,** <reset clause> }* **)**

<reset clause> ::=
<timer identifier> [**(** <expression list> **)**]

<set body> ::=
<set clause> { **,** <set clause> }*

<set clause> ::=

([<Time expression> **,**]<timer identifier> [**(**<expression list>**)]**)

如果<timer identifier>表示一个在其定义中有<timer default initialization>的计时器，那么<set clause>可以省略<Time expression>。

注 — 有关计时器具体句法至抽象句法的映射在第11.13.1节中描述。

语义

一个计时器实例是一个对象，它可以是活动的或不活动的。只有当适用于所有对应表达式的等同表达式（见第12.2.7节）都获得预定义的布尔值“真”时（也就是说，如果两个表达式列表具有相同的结果），后跟一个表达式列表的计时器标识符的两个事件指的才是同一个计时器实例。

当设置一个不活动的计时器时，一个时间值与计时器发生关联。假设在系统达到给时间值之前，没有重新设置或另外设置该计时器，那么在代理的输入端口中输入一个带有相同名称的信号，作为计时器。如果计时器设置的时间值小于等于**now**，那么采取相同的动作。在使用计时器信号后，**sender**表达式获得与**self**表达式相同的结果。如果表达式列表在设置计时器时给出，那么这些表达式的结果将以相同的次序包含在计时器信号中。从设置之时刻到使用计时器信号之时刻，计时器是活动的。

如果在计时器定义中规定的类别是一个共型，那么在第12.1.9.5节中定义的、适用于设置或重新设置中相应表达式的范围检查必须为预定义的布尔值“真”；否则，引起预定义的异常OutOfRange。

当重新设置不活动的计时器时，它保持不活动。

当重新设置活动的计时器时，失去与时间值的关联；如果在输入端口存在一个对应的保留计时器信号，那么它被移走，并且计时器变成不活动。

当设置活动的计时器时，这相当于重新设置计时器，后面紧跟设置计时器。在该重新设置和设置之间，计时器保持活动。

在第一次设置计时器实例之前，它是不活动的。

按给定的次序对*Set-node* 或 *Reset-node*中的*Expression*进行评估。

模型

不带<Time expression>的<set clause>是<set clause>的衍生语法，其中<Time expression>为：

now + <Duration constant expression>

其中<Duration constant expression>源自计时器定义中的<timer default initialization>。

一个<task area>可以包含若干个<reset clause>或<set clause>。这是用于规定一个<task area>序列的衍生语法，每个对应一个<reset clause> 或 <set clause>，这样，就保留了在<task area>中为其规定的最初次序。在对所含表达式中的缩写形式进行扩充之前，对该缩写形式进行扩充。

11.16 异常

异常实例将控制传送给异常句柄。

抽象语法

Exception-definition	::	*Exception-name*
		*Sort-reference-identifier**
Exception-name	=	*Name*
Exception-identifier	=	*Identifier*

具体语法

<exception definition> ::=

exception <exception definition item> { **,** <exception definition item> }* <end>

<exception definition item> ::=

<exception name> [<sort list>]

语义

一个异常实例表示在解释系统的同时发生了一个异常状况（典型地是一个错误状况）。通过基本系统隐性地创建一个异常实例，或通过*Raise-node*来显性地创建，如果*Handle-node* 或 *Else-handle-node*获取到了它，那么异常实例不再存在。

异常实例的创建将中断代理、操作或过程内的正常控制流。如果在一个被调用的过程、操作或复合语句中创建异常实例，并且不在该处被获取，那么过程、操作或复合语句分别终结，并且异常实例向外传播（动态地）给调用者，并被当做仿佛它是在过程调用、操作应用或复合语句调用处创建的。该规则也适用于远程过程调用；并且在这种情况下，除在被调用代理实例内进行传播外，异常实例还传播回调用进程实例。

许多异常类型是在预定义的包中进行预定义的。这些异常类型是可以通过基础系统隐性地进行创建的异常类型。它也允许特殊使用者显性地创建这些异常类型的实例。

如果在一个代理实例内创建异常实例，并且不在此处获取，那么系统的更多行为是未定义的。

11.16.1 异常句柄

抽象语法

Exception-handler-node :: *Exception-handler-name*
[*On-exception*]
*Handle-node-**set***
[*Else-handle-node*]

Exception-handler-name = *Name*

在某个给定*State-transition-graph*或*Procedure-graph*内的所有Exception-handler-node都必须具有不同的Exception-handler-name。

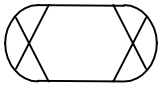
注 — *Exception-handler-name*可以与*State-name*有相同的名称。不过，它们是不同的。

*Handler-node-**set***中的*Exception-identifier*必须不同。

具体语法

<exception handler area> ::=
<exception handler symbol> ***contains*** <exception handler list>
[***is connected to*** <on exception association area>]
is associated with <exception handler body area>

<exception handler symbol> ::=



<exception handler body area> ::=
<handle association area>*

<handle association area> ::=
<solid association symbol> ***is connected to*** <handle area>

<exception handler list> ::=
<exception handler name> { **,** <exception handler name> }*
| <asterisk exception handler list>

<asterisk exception handler list> ::=
<asterisk> [**(** <exception handler name> { **,** <exception handler name>}* **)**]

<exception handler area>代表一个或多个*Exception-handler-node*。源自<exception handler symbol>的<solid association symbol>可以拥有一条共同的起始路径。如果<exception handler area>与<on exception area>一致，那么它必须包含<state name>（不是<asterisk state list>）。

当<exception handler list>包含一个<exception handler name>时，<exception handler name>代表一个*Exception-handler-node*。对每个*Exception-handler-node*，通过在其<exception stimulus list>中含有<exception identifier>的<handle area>来表示*Handle-node-**set***。对每个*Exception-handler-node*，通过一个显性或隐性的<handle area>来表示*Else-handle-node*，其中的<exception stimulus list>是一个<asterisk exception stimulus list>。

<asterisk exception handler list>中的各<exception handler name>必须是不同的，并必须包含在嵌套体或父类型体中的其他<exception handler list>中。

一个<exception handler area>至多包含一个<asterisk exception stimulus list>（见第11.16.3节）。

一个<exception handler area>至多有一个相关的<exception handler area>。

语义

一个异常句柄代表一个特殊的条件，在该条件下，代理、操作或过程可以处理它已创建的异常实例。处理一个异常实例的结果是产生一个转换。进程或过程的状态不会发生改变。

如果*Exception-handler-node*没有*Handle-node*，它以相同的*Exception-identifier*作为异常实例，那么通过*Else-handle-node*来获取异常实例。如果没有*Else-handle-node*，那么在该异常句柄中不对异常实例进行处理。

模型

当某个<exception handler area>的<exception handler list>包含多个<exception handler name>时，为每个这样的<exception handler name>创建一个<exception handler area>的拷贝。而后用这些拷贝来替换<exception handler area>。

将带<asterisk exception handler list>的<exception handler area>转化为<exception handler area>列表，每个对应一个在异体的<exception handler name>，包含在<asterisk exception handler list>中的那些<exception handler name>除外。

11.16.2 On-Exception

抽象语法

On-exception :: *Exception-handler-name*

在*On-exception*中规定的*Exception-handler-name* 必须为同一*State-transition-graph* 或 *Procedure-graph*内的某个<exception handler area> 的名称。

具体语法

<on exception association area> ::=
<solid on exception association symbol> ***is connected to***
{ <on exception area> | <exception handler area> }

<solid on exception association symbol> ::=

<on exception area> ::=
<exception handler symbol> ***contains*** <exception handler name>

<exception handler name>可以出现在一个体的多个<exception handler area>中。

<solid on exception association symbol>可以由若干个水平和垂直的线段组成。箭头必须附于<on exception area>或<exception handler area>之上。

语义

*On-exception*指明如果代理或过程创建一个异常实例，那么应该进入哪个异常句柄代理、操作或过程。通过一个<on exception association area>或<handle statement>，异常句柄与另一个实体相关。无论何时能对异常实例创建做出反应，都认为异常句柄是活动的。

在同一时间可以有若干异常句柄是活动的。对每个代理、过程或操作实例，有若干异常范围可以包含有一个活动的异常句柄。按递增位置次序，异常范围为：

a） 实例的整个图；

b） 复合状态（如果正在解释一个复合状态的话）；

c） 复合状态的图（如果有的话）；

d） 当前状态；

e） 有关当前状态中激励的转换，或起始转换；

f） 当前异常状态；

g） 有关当前句柄子句的转换；以及

h） 当前动作。

由于复合状态的嵌套，任何时候都可以有多个有关复合状态或复合状态图的异常句柄是活动的。

当创建一个异常实例时，按递减位置次序，对活动的异常句柄进行访问。当访问一个异常状态时，异常句柄为当前不活动的异常范围。如果没有一个有关某个异常范围的异常句柄是活动的，或者如果异常状态不处理异常，那么访问下一个异常范围。

在解释<constant expression>期间，没有一个异常句柄是活动的。

一个异常句柄可以与整个代理/过程/操作图、起始转换、状态、异常句柄、带有其相关转换的状态触发器（如输入或处理）、转换动作（大多数种类）或转换终结符（一些种类）相关。下列文本描述每种异常句柄为活动和不活动的情况。

a） 整个代理/过程/操作图

在解释代理图、操作或进程实例之初，激活异常句柄；当代理、操作或进程实例处于停止条件或不再存在时，异常句柄是不活动的。

b） 起始转换

当在代理、操作或过程中开始解释起始转换时，激活异常句柄；当代理或过程解释一个nextstate节点时，或者当处于停止条件或不再存在时，异常句柄是不活动的。

c） 复合状态

当进入复合状态时，激活异常句柄；对包括任何*Connect-node*或附于状态的转换的复合状态，它是活动的。当解释进入另一个状态时，它是不活动的。

d） 复合状态图

在调用复合状态的入口过程之前，异常句柄是活动的。在复合状态的出口过程完成后，异常句柄是不活动的。

e） 状态

无论何时当代理或过程进入给定状态时，异常句柄是活动的。当代理或过程解释一个nextstate节点时，或者当进入一个停止条件或不再存在时，异常句柄是不活动的。

f） 异常句柄

无论何时当代理或过程进入给定异常句柄时，异常句柄是活动的。当代理或过程进入一个nextstate节点时，或者当进入一个停止条件或不再存在时，异常句柄是不活动的。

g） 输入

无论何时当在代理或过程中开始解释给定Input-node时，有关激励的异常句柄是活动的。当代理或过程进入一个nextstate节点时，或者当进入一个停止条件或不再存在时，异常句柄是不活动的。

h） 处理

无论何时当在代理、操作或过程中开始解释*Handle-node*的*Transition*时，有关当前*Handle-node*的异常句柄是活动的。当在代理、操作或过程中进入一个nextstate节点时，异常句柄是不活动的。

i） 决定

无论何时当在代理、操作或过程中开始解释给定决定时，异常句柄是活动的。当代理或过程进入决定分支的转换时（也就是说，异常句柄覆盖决定的表达式，并且无论表达式是否匹配决定分支的任何范围），异常句柄是不活动的。

j） 转换动作（决定除外）

无论何时当在代理、操作或过程中开始解释给定动作时，异常句柄是活动的。当动作的代理或过程解释完成时，异常句柄是不活动的。

k） 转换终结符（带表达式）

无论何时当代理、操作或过程进入给定终结符时，异常句柄是活动的。当终结符解释完成时，异常句柄是不活动的。

当处理异常和创建异常实例时，任何异常句柄都是不活动的。动作和终结符的异常句柄也覆盖源自<transition area>模型的动作，例如<import expression>。

注 — 以上规则意味着，在某些情况下，可以在同一时间使若干异常句柄变得不活动。例如，如果在同一时间，状态的一个异常句柄和相关输入转换的一个异常句柄是活动的，那么当输入转换进入一个nextstate 节点时，两个异常句柄都将变得不活动。隐性状态或激励由语法正文的异常句柄覆盖；也就是说，将<on exception association area>拷贝进模型中。

模型

当若干个<exception handler area>含有同一<exception handler name>时，将这些<exception handler area>连接成一个具有该<exception handler name>的<exception handler area>。

在特殊化中，与异常句柄的关联被认为是图或转换的一部分。如果重新定义一个虚拟转换，那么新的转换将替换最初转换的<on exception association area>。如果在特殊化中继承图或状态，那么也继承任何相关的异常句柄。

11.16.3 句柄

抽象语法

Handle-node :: *Exception-identifier*
[*Variable-identifier*]*
[*On-exception*]
Transition

Else-handle-node :: [*On-exception*]
Transition

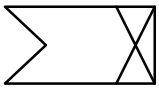
*Handle-node*中可选*Variable-identifier*的列表长度必须等同于以*Exception-identifier*表示的*Exception-definition*中的*Sort-reference-identifier*数量。

变量的类别必须按位置对应数据项的类别，数据项可以通过异常承载。

具体语法

<handle area> ::=
<handle symbol> ***contains*** { [<virtuality>] <exception stimulus list> }
[***is connected to*** <on exception association area>]
is followed by <transition area>

<handle symbol> ::=



<exception stimulus list> ::=
<exception stimulus> { **,** <exception stimulus> }*
| <asterisk exception stimulus list>

<exception stimulus> ::=
<exception identifier> [**(** [<variable>] { **,** [<variable>] }* **)**]

<asterisk exception stimulus list> ::=
<asterisk>

至<handle area>中<transition area>的路径必须源自<handle symbol>。

其<exception stimulus list>包含<exception stimulus>的<handle area>对应*Handle-node*。包含在<handle symbol>中的每个<exception identifier>给出该<handle symbol>代表的一个*Handle-node*的名称。一个带<asterisk exception stimulus list>的<handle area>代表一个*Else-handle-node*。

当＜exception stimulus list＞包含＜exception stimulus＞时，代表*Handle-node*。一个带＜asterisk exception stimulus list＞的＜handle area＞代表一个*Else-handle-node*。

可以省略＜exception stimulus＞中最后一个＜variable＞之后的逗号。

语义

*Handle-node*使用特定异常类型的一个实例。异常实例的使用使得通过异常实例传送的信息可以供代理或过程使用。在*Handle-node*中提到的变量是通过所用异常实例传送的数据项。

数据项从左到右指配给各变量。如果在异常中未提到有关某个给定参数位置的变量，那么抛弃该位置上的数据项。如果没有数据项与某个给定参数位置相关，那么对应的变量变成为“未定义的”。

为**sender**表达式提供等同于**self**表达式的结果。

注 — **state**表达式不改变异常句柄的名称。

模型

当某个＜handle area＞的＜exception stimulus list＞包含多个＜exception stimulus＞时，为每个这样的＜exception stimulus＞创建一个＜handle area＞的拷贝。而后用这些拷贝来替换＜handle area＞。

当某个＜exception stimulus＞的一个或多个＜variable＞为＜indexed variable＞或＜field variable＞时，用惟一的、新的、隐含声明的＜variable identifier＞来替换所有的＜variable＞。就在＜handle area＞的＜transition area＞之前，插入一个＜task area＞，在它的＜task body＞中对每个＜variable＞含有一个＜assignment＞，用于将对应的新变量的结果指配给＜variable＞。对结果按＜variable＞列表从左到右的次序进行指配。该＜task area＞变成为＜transition area＞中的第一个＜action area＞。

包含＜virtuality＞的＜handle area＞被称为虚拟句柄转换。对虚拟句柄转换在第8.3.3节中有更多的描述。

12 数据

SDL中的数据概念在本节中定义。它包括数据术语、用于定义新数据类型和预定义数据的各概念。

SDL中的数据原则上与数据类型有关。一个数据类型定义了一组元素或数据项，作为类别，一组操作可用于这些数据项。类别和操作定义数据类型的属性。这些属性通过数据类型定义来定义。

数据类型包括一个集合（它是数据类型的类别）以及一个或多个操作。作为一个例子，考虑布尔预定义数据类型。布尔数据类型的布尔类别由元素“真”和“假”组成。布尔数据类型的操作有“=”（等于）、“/=”（不等于）、“not”、“and”、“or”、“xor”、“=>”（意味着）。作为另一个例子，考虑自然数预定义数据类型。它有自然数类别，包括元素0、1、2等等，操作有“=”、“/=”、“＋”、“－”、“*”、“/”、“rem”、“＜”、“＞”、“<=”、“>=”和幂。

SDL提供若干预定义数据类型，其行为和语法是常见的。预定义数据类型在附件D中描述。

变量可以通过指配与类别的某个元素相关联。当访问变量时，返回相关的数据项。

一个数据类型的类别元素可以是值、作为值的引用的对象，或作为代理的引用的pid。数据类型的类别可以用以下方式进行定义：

a） 明确地列举出类别的元素。

b） 生成类别S_1，S_2，… S_n的笛卡尔积；类别等同于由所有tuple组成的集合，集合可以通过以下方式生成：从类别S_1中提出第一个元素，从类别S_2中提取第二个元素，……，最后从类别S_n中提取最后一个元素。

c） Pid类别通过定义一个接口来定义（见第12.1.2节）。

d） 预定义了若干类别，并形成附件D中所述之预定义数据类别的基础。预定义的类别Any和Pid在第12.1.5和12.1.6节中描述。

如果类别元素为对象，那么类别是一个对象类别。如果类别元素是pid，那么类别是一个pid类别。值类别的元素为值。

定义自类别元素和到类别元素的操作。例如，应用自整数类别元素和到整数类别元素的加（“＋”）操作是合法的，而对布尔类别元素实施加操作是不合法的。

每个数据项都明确地属于某个类别。也即类别没有共同的数据项。

对大多数类别，有文字形式可以来表示类别元素：例如，对整数，使用“2”而不使用“1＋1”。可以有多个文字来表示同一数据项；例如，可以使用“12”和“012”来表示同一整数数据项。可以对多个类别使用相同的文字表示；例如，“A”既是一个字符，也是一个长度为1的字符串。某些类别可以没有文字形式来表示类别元素；例如，类别也可以形成为其他类别的笛卡尔积。在这种情况下，这些类别的元素通过操作来表示，它从部件类别元素来构造数据项。

一个表达式表示一个数据项。如果表达式不包含变量或祈使表达式，例如，如果它是某个给定类别的一个文字，那么表达式的每个事件将总表示同一数据项。这些“被动的”表达式对应语言的一个功能性用法。

包含变量或祈使表达式的表达式可以被解释为在解释SDL系统期间具有不同的结果，这依据与变量相关的数据项。数据的主动使用包括对变量的指配、变量的使用、变量的初始化。主动表达式和被动表达式之间的差别在于被动表达式的结果与其何时被解释无关，而主动表达式依据当前值、对象、与变量或当前系统状态相关的pid，可以有不同的结果。

12.1 数据定义

数据定义用于定义数据类型。定义数据的基本机制是数据类型定义（见第12.1.1节）和接口（见第12.1.2节）。特殊化（见第12.1.3节）允许定义一个基于另一个数据类型的数据烈性，作为其父类型。数据类型类别定义以及操作意味着可以通过数据类型构造器给出类别（见第12.1.7节）。可以定义额外的操作，如第12.1.4节所述。第12.1.8节显示了如何定义一个数据类型的操作行为。

由于预定义数据在预定义包中定义并隐含地使用（见第7.2节和D.3），因此预定义类别（例如，布尔类别和自然数类别）及其操作可以在整个系统中自由地使用。等同性（见第12.2.5节）、条件表达式（见第12.2.6节）和共型（见第12.1.9.4节）的语义依赖于布尔数据类型（见D.3.1）的定义。名称类（见第12.1.9.1节）的语义也依赖于字符和字符串（见D.3.2和D.3.4）的定义

抽象语法

Data-type-definition	=	*Value-data-type-definition*
	|	*Object-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort*
		Data-type-identifier
		*Literal-signature-**set***
		*Static-operation-signature-**set***
		*Dynamic-operation-signature-**set***
		*Data-type-definition-**set***
		*Syntype-definition-**set***
		*Exception-definition-**set***
Object-data-type-definition	::	*Sort*
		Data-type-identifier
		*Literal-signature-**set***
		*Static-operation-signature-**set***
		*Dynamic-operation-signature-**set***
		*Data-type-definition-**set***
		*Syntype-definition-**set***
		*Exception-definition-**set***
Interface-definition	::	*Sort*
		*Data-type-identifier**
		*Data-type-definition-**set***
		*Syntype-definition-**set***
		*Exception-definition-**set***
Data-type-identifier	=	*Identifier*
Sort-reference-identifier	=	*Sort-identifier*

| | *Syntype-identifier*
| | *Expanded-sort*
| | *Reference-sort*

Sort-identifier = *Identifier*

Expanded-sort = *Sort-identifier*

Reference-sort = *Sort-identifier*

Sort = *Name*

*Data-type-definition*引入了一个类别，它在抽象句法的嵌套范围单元中是可见的。另外，它还可以引入一组文字和操作。

具体语法

<data definition> ::=
<data type definition>
| <interface definition>
| <syntype definition>
| <synonym definition>

如果一个数据定义是<data type definition>、<interface definition>或<syntype definition>，那么它代表一个*Data-type-definition*。

<sort> ::=
<basic sort> [**(** <range condition> **)**]
| <anchored sort>
| <expanded sort>
| <reference sort>
| <pid sort>
| <inline data type definition>
| <inline syntype definition>

<inline data type definition> ::=
{ **value** | **object** } [<data type specialization>]
[[<comment body>] <left curly bracket> <data type definition body>
<right curly bracket>]

<inline syntype definition> ::=
syntype <basic sort>
[[<comment body>] <left curly bracket>
{ <default initialization> [[<end>] <constraint>] | <constraint> } <end>
<right curly bracket>]

<basic sort> ::=
<datatype type expression>
| <syntype>

<anchored sort> ::=
this [<basic sort>]

<expanded sort> ::=
value { <basic sort> | <anchored sort> }

<reference sort> ::=
object { <basic sort> | <anchored sort> }

<pid sort> ::=
<sort identifier>

带<basic sort>的<anchored sort>只允许处于<basic sort>定义内。

<anchored sort>只有当出现在<data type definition>内时才是合法的具体句法。<anchored sort>中的<basic sort>必须命名由<data type definition>引入的<sort>。

语义

数据定义用于数据类型定义或接口或表达式同义词定义，在第12.1、12.1.9.4或12.1.9.6节中做进一步定义。

每个<data type definition>引入一个类别，名称同<data type name>（见第12.1.1节）。每个<interface definition>引入一个类别，名称同<interface name>（见第12.1.2节）。

注1 — 为避免繁琐的文本，使用以下约定：当不可能引起混淆时，常用短语“类别S”来代替“通过数据类别S定义的类别”或“通过接口定义的类别”。

<sort identifier>通过数据类型定义命名引入的<sort>。

一个类别是元素的一个集合：值、对象（即值的引用）或pid（即代理的引用）。两个不同的类别没有共同的元素。<value data type definition>引入一个值集合类别。<object data type definition>引入一个对象集合类别。<interface definition>引入一个pid类别。

如果<sort>是一个<expanded sort>，那么用该<sort>定义的变量、同义词、字段、参数、返回、信号、计时器和异常将与类别的值相关，而不与这些值的引用相关，即使类别已作为对象集合定义。*Expanded-sort-identifier*通过<expanded sort>来表示。

如果<sort>是一个<reference sort>，那么用该<sort>定义的变量、同义词、字段、参数、返回、信号、计时器和异常将与类别的值引用相关，而不与类别的值相关，即使类别已作为值结合定义。*Referenced-sort-identifier*通过<reference sort>来表示。

<anchored sort>的含义在第12.1.3节中给出。

<pid sort>中的<sort identifier>必须引用一个pid类别。

模型

带<basic sort>的<expanded sort>代表一个值类别，它由<basic sort>替代。

带<basic sort>的<reference sort>代表一个对象类别，它由<basic sort>替代。

注2 — 结果是，如果类别已定义为一个值的集合，那么关键字**value**没有作用，如果类别已定义为一个对象的集合，那么关键字**object**没有作用。

不带<basic sort>的<anchored sort>是一种缩写形式，用于规定<basic sort>，它在正文中引用数据类型定义或共型定义的名称，<anchored sort>在正文中出现。

如果<sort>是带<range condition>的<basic sort>，那么它源自隐含的<syntype definition>的<syntype>的具体句法，匿名。如果<basic sort>已利用文字数据类型构造器进行构造，那么该匿名的<syntype definition>通过其元素进行定义，元素受限于<range condition>；否则<range condition>是<size constraint>的一部分。

<inline data type definition>源自隐含的<data type definition>的<basic sort>的具体句法，匿名。该匿名的<data type definition>源自<inline data type definition>，通过在<inline data type definition>中的**value** 或 **object**之后插入**type**和匿名生成。每个<inline data type definition>定义一个不同的隐含的<data type definition>。

<inline syntype definition>源自隐含的<syntype definition>的<basic sort>的具体句法，匿名。该匿名的<syntype definition>源自<inline syntype definition>，通过在<inline syntype definition>中的共型之后插入匿名和<equals sign>生成。

12.1.1 数据类型定义

数据类型定义有一个体，通常包含一个数据类型构造器，并指明数据类型是否是一个**value**或**object**数据类型。

数据类型构造器定义如何构造数据集（结构化值、文字值和选择值）。如果数据类型定义是一个**value**类型，那么这些值是类别的元素。如果数据类型定义是一个**object**类型，那么这些值是由类别元素引用的内容。

具体语法

```
<data type definition> ::=
                        {<package use clause>}*
                        <type preamble> <data type heading> [<data type specialization>]
                        {           <end>
                        |           [ <comment body> ] <left curly bracket> <data type definition body>
                                    <right curly bracket> }

<data type definition body> ::=
                        {<entity in data type>}* [<data type constructor>] <operations>
                        [<default initialization> <end> ]

<data type heading> ::=
```

{ **value** | **object** } **type** <data type name>
[<formal context parameters>] [<virtuality constraint>]

<entity in data type> ::=
<data type definition>
| <syntype definition>
| <synonym definition>
| <exception definition>

<operations> ::=
<operation signatures>
<operation definitions>

<value data type definition>在<data type heading>中包含关键字**value**。<object data type definition>在<data type heading>包含关键字**object**。

<formal context parameters>的<formal context parameter>必须是一个<sort context parameter>或者是一个<synonym context parameter>。

对每个<operation signatures>的<operation signature>，在<operations>的<operation definitions>中只能有一个相应的定义（<operation definition>或<external operation definition>）。

语义

<data type definition>由描述类别元素的<data type constructor>（见第12.1.6节）和在构造类别之时产生的操作组成，定义一组操作的<operations>可以应用于类别的各元素（见第12.1.4节）。数据类型也可以通过特殊化基于父类型（见第12.1.3节）。

12.1.2 接口定义

接口在包、代理或代理类型中定义。接口定义了一个pid类别，其元素为代理的引用。

接口可以定义信号、远程过程、远程变量和议程。在接口中定义的实体定义正文是接口的范围单元，并且在接口可见之处，定义的实体是可见的。接口还可以指通过<interface use list>在接口外面定义的信号、远程过程或远程变量。

接口用在信号列表中，用来表示信号、远程过程调用和接口定义的远程变量包括在信号列表中。

具体语法

<interface definition> ::=
{<package use clause>}*
[<virtuality>] <interface heading>
[<interface specialization>] <end>
| {<package use clause>}*
[<virtuality>] <interface heading>
[<interface specialization>] [<comment body>] <left curly bracket>
<entity in interface>* [<interface use list>]
<right curly bracket>

<interface heading> ::=
interface <interface name>
[<formal context parameters>] [<virtuality constraint>]

<entity in interface> ::=
<signal definition>
| <interface variable definition>
| <interface procedure definition>
| <exception definition>

<interface use list> ::=
use <signal list> <end>

<interface variable definition> ::=
dcl <remote variable name> { **,** <remote variable name>}* <sort> <end>

<interface procedure definition> ::=

procedure <remote procedure name> <procedure signature> <end>

<formal context parameters>应只包含<signal context parameter>、<remote procedure context parameter>、<remote variable context parameter>、<sort context parameter>或<exception context parameter>。

在接口（<entity in interface>）中定义的实体定义正文是接口的范围单元，并且在接口可见之处，定义的实体是可见的。

模型

<virtuality>的语义在第8.3.2节中定义。

接口内容是所有信号、远程过程和远程变量的集合，它们通过特殊化（即继承或正文参数化）在接口的<entity in interface>中定义，或在<interface use list>中引用，或包括在接口中。

在<signal list>中包括<interface identifier>指的是形成<interface definition>一部分的所有信号标识符、远程过程标识符和远程变量标识符都包括在<signal list>中。

接口通过代理和代理类型定义以及代理状态机和代理类型定义来隐含地定义。代理或代理类型隐含定义的接口具有相同的名称，并在与定义它的代理或代理类型相同的范围单元中进行定义。状态机隐含定义的接口与包含的代理或代理类型具有相同的名称，但在与定义它的状态机相同的范围单元中进行定义，即在代理或代理类型内部。

通过代理或代理类型定义的接口在其<interface specialization>中包含在输入信号列表中给出的所有接口，它们与代理或代理类型的显性或隐性通道相关，这样，通道通过隐性或显性的信道与代理或代理类型的状态机的通道相连。接口还在其<interface use list>中包含在输入信号列表中给出的所有信号、远程变量和远程过程，它们与代理或代理类型的显性或隐性通道相关，这样，通道通过隐性或显性的信道与代理或代理类型的状态机的通道相连。另外，继承另一个代理类型的代理类型接口还在其<interface specialization>中包含隐性接口，它们通过继承的代理类型定义。

注1 — 由于每个代理和代理类型具有一个名称相同的、隐含定义的接口，因此任何显性定义的接口必须具有一个不同于任何代理和代理类型的名称，它们在同一范围内定义；否则，名称上会存在冲突。

通过代理或代理类型状态机定义的接口在其<interface specialization>中包含通过代理或代理类型本身定义的接口。另外，接口在其<interface specialization>中包含在输入信号列表中给出的所有接口，它们与状态机的显性或隐性通道相关，这样，通道不通过隐性或显性信道与代理或代理类型的显性或隐性通道相连。如果包含的实体是一个代理类型，它继承了另一个代理类型，那么接口还将在其<interface specialization>中包含继承代理类型的状态机的隐性接口。

通过基于类型的代理定义的接口在其<interface specialization>中包含通过其类型定义的接口。

注2 — 为避免繁琐的文本，使用以下约定：当不可能引起混淆时，常用短语“代理的pid类别”来代替“通过接口定义的pid类别（接口通过代理A隐含地定义）”。

12.1.3 数据类型的特殊化

特殊化允许基于另一个（父）类型定义一个数据类型。通过特殊化定义的类别被认为是通过基类型定义的类别的子类别。通过基类型定义的类别是通过特殊化定义的类别的父类别。

具体语法

<data type specialization> ::=

inherits <data type type expression> [<renaming>] [**adding**]

<interface specialization> ::=

inherits <interface type expression> { **,** <interface type expression> }* [**adding**]

<renaming> ::=

(<rename list> **)**

<rename list> ::=
<rename pair> { , <rename pair> }*
<rename pair> ::=
<operation name> <equals sign> <base type operation name>
| <literal name> <equals sign> <base type literal name>

*Data-type-definition*中的*Data-type-identifier*通过<data type definition>表示，其中含有<data type specialization>（或<interface specialization>），用于标识数据类型，它通过<data type specialization>中的<data type type expression>来表示（见第8.1.2节）。

*Interface-definition*可以包含一个*Data-type-identifier*s列表。通过*Interface-definition*表示的接口是所有通过*Data-type-identifier*s表示的接口的一个特殊化。

特殊化接口定义的结果内容（<interface specialization>）包括父类型的内容，后跟特殊化定义的内容。这意味着特殊化接口定义的信号、远程过程和远程变量集是特殊化定义中给出的信号、远程过程和远程变量以及父类型的信号、远程过程和远程变量的组合。定义的结果集合必须遵守第6.3节中给出的规则。

<data type constructor>必须与<data type constructor>属于同一种类，<data type constructor>在类别的<data type definition>中使用，类别由<data type specialization>中的<data type type expression>引用。也就是说，如果用在（直接或间接）父类型中的<data type constructor>是一个<literal list>（<structure definition>、<choice definition>），那么<data type constructor>必须也是一个<literal list>（<structure definition>、<choice definition>）。

可通过重新命名来修改衍生数据类型中继承的文字、运算符和方法的名称。

在<rename list>中的所有<literal name>和所有<base type literal name>都必须是不同的。

在<rename list>中的所有<operation name>和所有<base type operation name>都必须是不同的。

在<rename list>中规定的<base type operation name>必须是一个带<operation name>的操作，<operation name>在数据类型定义中定义，数据类型定义用于定义<data type type expression>的<base type>。

语义

类别兼容性用于确定何时可以以及何时不可以用一个类别来代替另一个类别。这个关系用于指配（见第12.3.3节）、用于参数传递（见第12.2.7和9.4节）、用于在继承期间重新声明参数类型（见第12.1.2节）以及用于实际的正文参数（见第8.1.2节）。

设T和V为两个类别。当且仅当满足下列条件时，V才兼容于T：

a） V和T为同一类别；

b） V是直接兼容于T的类别；

c） T已通过一个对象类型或一个接口进行定义，并且，对某类别U，V兼容于U，并且U是兼容于T的类别。

注1 — 类别兼容性只对通过对象类型或接口定义的类别才是可传递的，对通过值类型定义的类别并不可传递。

设T和V为类别。当且仅当满足下列条件时，V才是直接兼容于T的类别：

a） V通过<basic sort>表示，并且T是一个对象类别，T是V的父类别；

b） V通过**this** T形式的<anchored sort>表示；

c） V通过**object** T形式的<reference sort>表示；

d） T通过**object** V形式的<reference sort>表示；

e） V通过**value** T形式的<expanded sort>表示；

f） T通过**value** V形式的<expanded sort>表示；或者

g） V通过<pid sort>表示（见第12.1.2节），并且T是V的父类别。

模型

使用第8.3节中的特殊化模型，变量如下所述。

通过使用<data type definition>以及<data type specialization>，特殊化数据类型基于另一个（基）数据类型。通过特殊化定义的类别与通过基类型定义的类别脱节。

如果通过基类型定义的类别具有定义的文字，那么文字名称是继承的，类别文字名称由特殊化类型定义，除非已对该文字进行文字重命名。如果基类型文字名称作为第二个名称出现在<rename pair>中，那么对文字进行重命名，在这种情况下，文字成对地命名为第一个名称。

如果基类型具有定义的运算符或方法，那么操作名称是继承的，类别运算符名称或方法依据第8.3.1节中所述的限制条件进行定义，除非运算符或方法已声明为是专用的（见第12.1.9.3节）或已对该运算符或方法进行操作重命名。如果继承的操作名称作为第二个名称出现在<rename pair>中，那么对运算符或方法进行操作重命名，在这种情况下，运算符或方法成对地命名为第一个名称。

当<sort type expression>的<base type>的若干个运算符或方法具有相同的名称时，作为<rename pair>中的<base type operation name>，那么对所有这些运算符或方法进行重命名。

在特殊化类型中的<anchored sort>的每个事件中，用子类别来替换<basic sort>。

变元类别和继承运算符或方法的结果与基类型对应运算符或方法的结果相同，除非在每个<argument>含有一个在继承运算符或方法中的<anchored sort>，用子类别替换<basic sort>。对继承的虚拟方法，如果它尚不存在，那么将<argument virtuality>加入到含有<anchored sort>的<argument>中。

注2 — 根据第8.3节中所述的特殊化模型，运算符只能在其特征包含至少一个<anchored sort>或已进行重命名的情况下才能继承。

12.1.4 操作

抽象语法

Dynamic-operation-signature	=	*Operation-signature*
Static-operation-signature	=	*Operation-signature*
Operation-signature	::	*Operation-name* *Formal-argument** [*Result*] *Identifier*
Operation-name	=	*Name*
Formal-argument	=	*Virtual-argument*
	|	*Nonvirtual-argument*
Virtual-argument	::	*Argument*
Nonvirtual-argument	::	*Argument*
Argument	=	*Sort-reference-identifier*

运算符特征中的*Identifier*对对应操作的匿名过程是一个匿名标识符。

类别兼容性的概念扩展至*Operation-signature*s。当满足下列条件时，*Operation-signature* S1是一个兼容于*Operation-signature* S2的类别：

a） S1和S2拥有相同数量的形式变元；以及

b） 对S1的每个*Virtual-argument* A，通过其*Sort-reference-identifier*标识的类别都兼容于通过S2中相应变元的*Sort-reference-identifier* 标识的类别；

c） 对S1的每个*Nonvirtual-argument* A，通过其*Sort-reference-identifier*标识的类别都等同于通过S2中相应变元的 *Sort-reference-identifier* 标识的类别。

具体语法

<operation signatures> ::=
[<operator list>] [<method list>]

<operator list> ::=
operators <operation signature> { <end> <operation signature> }* <end>

<method list> ::=
methods <operation signature> { <end> <operation signature> }* <end>

<operation signature> ::=
<operation preamble>
{ <operation name> | <name class operation> }
[<arguments>] [<result>] [<raises>]

<operation preamble> ::=
[<virtuality> [<visibility>] | <visibility> [<virtuality>]]

<operation name> ::=
<operation name>
| <quoted operation name>
<arguments> ::=
(<argument> { **,** <argument> }* **)**
<argument> ::=
[<argument virtuality>] <formal parameter>
<formal parameter> ::=
<parameter kind> <sort>
<argument virtuality> ::=
virtual
<result> ::=
<result sign> <sort>

在*Operation-signature*中，*Formal-argument*中的每个*Sort-reference-identifier*通过一个变元<sort>来表示，*Result*通过结果<sort>来表示。<argument>中含有<argument virtuality>的<sort>代表一个*Virtual-argumen*，否则<argument>的<sort>代表一个*Nonvirtual-argument*。

在抽象句法中的定义范围单元内，*Operation-name*是惟一的，尽管相应的<operation name>可以不惟一。惟一的*Operation-name*源自：

a） <operation name>；加上

b） （可能为空）变元类别标识符列表；加上

c） 结果类别标识符；加上

d） 数据类型定义的类别标识符，<operation name>在其中进行定义。

<quoted operation name>允许具有特殊语法形式的运算符和方法名称。引入特殊语法以便，例如算术操作和布尔操作，能够具有其通常的语法形式。也就是说，使用者可以写成“（1＋1）=2”，而不必使用，例如equal(add(1,1),2)。

如果<operation signature>包含在一个<operator list>中，那么<operation signature>代表一个*Static-operation-signature*，并且<operation signature>不必包含<virtuality>或<argument virtuality>。

如果<operation signature>包含在<method list>中，并且<virtuality>未出现，那么<operation signature>代表一个*Static-operation-signature*，并且没有一个<argument>可包含<argument virtuality>。

如果<operation signature>包含在<method list>中，并且<virtuality>出现，那么<operation signature>代表一个*Dynamic-operation-signature*。在这种情况下，*Dynamic-operation-signature*s集由代表<operation signature>的*Dynamic-operation-signature*和带*Operation-name*（源自同一<operation name>，考虑到了重命名=的父类型中的匹配方法特征集中的任何元素组成，这样，*Operation-signature*在类别上兼容于父类型中的*Operation-signature*（如果有的话）。

该集合必须以下列方式进行闭合：对*Operation-signature*s集合中的任何两个*Operation-signature*s S_i 和 S_j，*Operation-signature* S必须惟一，这样

a） S在类别上兼容于S_i 和 S_j；以及

b） 对任何类别上兼容于S_i 和 S_j的*Operation-signature* S_k，S_k也在类别上兼容于S；

它也在*Dynamic-operation-signature*s集合中。

该条件确保*Dynamic-operation-signature*s集合形成一个框架，并确保在解释操作应用时能发现惟一最佳匹配的*Operation-signature*（见第12.2.7节）。如果*Dynamic-operation-signature*s集合不满足该条件，那么<sdl specification>无效。

注 — 类型规范可以要求将额外的*Operation-signature*s加入到<method list>中，以便满足该条件。

只有当<operation signature>出现在<method list>中时，才可以省略<operation signature>中的<result>。

只有当<virtuality>含有关键字**virtual** 或 **redefined**时，<argument virtuality>才合法。

语义

中缀或单体运算符或方法的引用形式是运算符或方法的合法名称。

一个运算符或方法有一个结果类别，它是通过结果确定的类别。

模型

如果<operation signature>包含在<method list>中，那么它是衍生语法，并按下列方式进行转化：如果出现<virtuality>、<parameter kind> **in/out**并且类别的<sort identifier>由嵌套的<data type definition>定义，那么从关键字**virtual**构建<argument>。如果不存在<arguments>，那么从构建的<argument>生成<arguments>，并插入<operation signature>。如果存在<arguments>，那么构建的<argument>被加至<arguments>中<argument>最初列表的开始处。

如果<argument>的<sort>是<anchored sort>，那么<argument>隐含地包含<argument virtuality>。如果<operation signature>包含<virtuality>中的关键字**redefined**，对基类型的匹配<operation signature>中的每个<argument>，如果该<argument>（隐性地或显性地）包含<argument virtuality>，那么<operation signature>中对应的<argument>也隐含地包含<argument virtuality>。

不带显性<parameter kind>的<argument>有隐性的<parameter kind> **in**。

12.1.5 Any

每个值或对象类型是抽象对象类型Any（直接的或间接的）子类型。当变量声明为Any类别时，属于任何值或对象类别的数据项可以被指配给该变量。

具体语法

数据类型Any可以通过预定义的**package**来限定。

语义

Any通过下列<data type definition>来隐含地定义，其中Boolean是预定义的布尔类别：

```
abstract object type Any
operators
    equal          ( this Any，this Any    ) -> Boolean;
    clone          ( this Any              ) -> this Any;
methods
    virtual is_equal   ( this Any          ) -> Boolean;
    virtual copy       ( this Any          ) -> this Any;
endobject type Any;
```

注1 — 由于所有的数据类型构造器隐性地重新定义数据类型Any的虚拟方法，因此这些方法不能显性地在<data type definition>中重新定义。

另外，每个<object data type definition>引入一个名为S的类别意味着*Operation-signature*s相当于在<operator list>中的下列<operation signature>中包括显性定义：

```
Null            -> this S;
Make            -> this S;
```

由Any定义的运算符和方法对任何值或对象数据类型都可用。

每个<object data type definition>增加一个惟一的数据项，表示一个尚未与某个值相关的引用。运算符Null返回该数据项。任何从由Null返回的对象获得相关值的尝试都将引起预定义异常InvalidReference（见D.3.1.6节）。

由<object data type definition>引入的运算符Make创建一个新的、未初始化的Make运算符的<result sort>的元素。每个<object data type definition>提供一个适当的有关*Make*运算符的定义。

运算符*equal*比较两个值是否相等（当由值类型定义时），或比较两个由对象引用的值是否相等（当由对象类型定义时）。设X和Y是其实际参数的*Expression*s的结果，则：

a） 如果X或Y为Null，那么结果是：如果二者都为Null，那么预定义布尔值为“真”；如果只有一个为Null，那么预定义布尔值为“假”；否则

b） 如果Y的动态类别在类别上不兼容于X的动态类别，那么结果是：预定义布尔值为“假”；否则

c） 结果通过解释*x.is_equal*(*y*)获得，其中x和y分别代表X和Y。

运算符*clone*创建一个新的数据项，它属于其实际参数的类别，并通过拷贝至该数据项来初始化最近创建的数据项，给出最初的实际参数。应用*clone*后，最近创建的数据项等于实际参数。设Y是其实际参数*Expression*的结果，那么运算符*clone*按下列步骤进行定义：

a） 如果Y为Null，那么结果为Null；否则

b） 如果X的类别是一个对象类别，设X是为数据类型（它定义Y的类别）解释*Make*运算符的结果。通过解释*x.copy*(*y*)来获得结果，其中x和y分别代表X和Y；否则

c） 如果X的类别是一个值类别，设X是X类别的一个任意元素。通过解释*x.copy*(*y*)来获得结果，其中x和y分别代表X和Y。

方法*is_equal*用于比较**this**和实际参数，如果有的话，逐个部件地进行。通常，对返回预定义布尔值“真”的*is_equal*方法，**this**和实际参数都不得为Null，并且实际参数的类别必须在类别上兼容于**this**的类别。

数据类型定义可以重新定义*is_equal*，以便考虑到其对应类别的语义之间存在的差别。类型构造器将按下列步骤隐性地重新定义*is_equal*。设X和Y为其实际参数*Expression*s的结果，则：

a） 如果X的类别已经由一个＜literal list＞构建，如果X和Y为相同值，那么结果为预定义布尔值“真”；

b） 如果X的类别已经由一个＜structure definition＞构建，如果对X的每个部件，该部件等于Y的对应部件（通过用这些作为操作数的部件解释*Equality-expression*来决定，省略X和Y的那些部件，它们已经被定义为是可选的，并且不出现），那么结果为预定义布尔值“真”。

方法*copy*拷贝实际参数至**this**，如果有的话，逐个部件地进行。每个数据类型构造器增加一个方法，用于重新定义*copy*。通常，**this**和实际参数都不得为Null，并且实际参数的类别必须在类别上兼容于**this**的类别。*copy*的每个重新定义都必须满足后条件，它在*copy*方法应用之后，**this**与实际参数*is_equal*。

数据类型定义可以重新定义*copy*，以便考虑到其对应类别的语义之间存在的差别。类型构造器将按下列步骤自动地重新定义*copy*。设X和Y为其实际参数*Expression*s的结果，则：

a） 如果X的类别已经由一个＜literal list＞构建，那么Y拷贝至X；

b） 如果X的类别已经由一个＜structure definition＞构建，对X的每个部件，Y的对应部件通过解释*xc.Modify*(*yc*)来拷贝到X的那个部件上 — 其中*xc*代表X的部件，*Modify*为该部件的字段修改方法，*yc*代表Y的对应部件，省略X和Y的那些部件，它们已经被定义为是可选的，并且不出现。

注2 — 对*Modify*方法的解释涉及将实际参数指配给形式参数，并且因此递归地调用拷贝方法（见第12.3.3节）。

模型

如果＜data type definition＞不包含＜data type specialization＞，那么这是带＜data type specialization＞的＜data type definition＞的缩写符号。

inherits Any;

12.1.6 Pid和pid类别

每个接口（直接地或间接地）都是接口Pid的一个子类型。当变量声明为Pid类别时，属于任何pid类别的数据项都可以被指配给该变量。

具体语法

数据类型Pid可以通过预定义的**package**限定。

语义

Pid类别包含一个单个的数据项，它由文字Null表示。Null表示一个不与任何代理相关的引用。

通过不带＜interface specialization＞的＜interface definition＞表示的*Interface-definition*只包含一个表示接口Pid的*Data-type-identifier*。

通过接口（由代理定义隐含地定义）引入的pid类别的元素通过解释*Create-request-node*与代理的引用相关（见第11.13.2节）。

每个接口增加一个兼容性检查操作，它给出一个信号，确定是否存在以下情况：

a） 信号是否已在接口中定义或使用；或

b） 兼容性检查是否满足pid类别要求（它由在其<interface specialization>中包含的接口定义）。

如果这未能满足，那么将引起预定义的异常InvalidReference（见D.3.16）。兼容性检查的定义类似于远程变量（见第10.6节）和远程过程（见第10.5节）。

注 — 可以对一个pid类别进行多态指配（见第12.3.3节）。

12.1.7 数据类型构造器

数据类型构造器规定数据类型类别的内容，或者通过枚举组成类别的各元素，或者通过收集所有的数据项，它们可以通过从给定类别的元素构造一个tuple来获得。

具体语法

<data type constructor> ::=
<literal list>
| <structure definition>
| <choice definition>

12.1.7.1 文字

文字数据类型构造器通过枚举类别的元素（可能无限多）来规定数据类型类别的内容。文字数据类型构造器隐含地定义允许对类别元素进行比较的操作。文字类别元素被称为文字。

抽象语法

Literal-signature :: *Literal-name*
Result
Literal-name = *Name*

具体语法

<literal list> ::=
[<visibility>] **literals** <literal signature> { **,** <literal signature> }* <end>
<literal signature> ::=
<literal name>
| <name class literal>
| <named number>
<literal name> ::=
<literal name>
| <string name>
<named number> ::=
<literal name> <equals sign> <Natural simple expression>

在*Literal-signature*中，*Result*是通过定义<literal signature>的<data type definition>引入的类别。

在抽象句法中，定义的范围单元内的*Literal-name*是惟一的，即使对应的<literal name>可能并不惟一。惟一的*Literal-name*源自：

a） <literal name>；加上

b） 数据类型定义的类别标识符，在其中定义<literal name>。

注 — 一个<string name>是词汇单元<character string>、<bit string>和<hex string>之一。

出现在<named number>中的<Natural simple expression>的每个结果，在<literal list>中的所有<literal signature>中必须是惟一的。

语义

通过枚举集合的所有元素，一个<literal list>定义一个类别。类别中的每个元素由一个*Literal-signature*来表示。

由<character string>形成的文字用于预定义的字符串数据类别（见D.3.4）和字符（见D.3.2）。它们与<regular expression>之间还有一种特殊的关系（见第12.1.9.1节）。由<bit string> 和 <hex string>形成的文字还用于预定义的整数数据类别（见D.3.5）。还可以对这些文字进行定义，以便用于其他用途。

<literal list>重新定义（直接地或间接地）自Any继承的操作，如第12.1.5节所述。

<literal list>中的<visibility>的含义在第12.1.9.3节中描述。

模型

<literal list>中的<literal name>是<named number>的衍生语法，<named number>包含<literal name>，并包含<Natural simple expression>，<Natural simple expression>表示最不可能出现在<literal list>的任何其他<literal signature>中的非负自然数值。<named number>对<literal name>的替换从左到右逐个进行。

文字列表是运算符定义的衍生语法，它在类别中建立一个元素次序，类别由<literal list>定义：

a） 运算符就建立的次序对两个数据项进行比较；

b） 运算符返回次序中的第一个、最后一个、下一个或前一个数据项；以及

c） 运算符给出每个数据项在次序中的位置。

通过<literal list>引入一个名为S的类别的<data type definition>表示的是一组*Static-operation-signature*s，它相当于下列<operator list>中的显性定义：

```
"<"     ( this S，this S )    -> Boolean;
">"     ( this S，this S )    -> Boolean;
"<="    ( this S，this S )    -> Boolean;
">="    ( this S，this S )    -> Boolean;
first                         -> this S;
last                          -> this S;
succ    ( this S )            -> this S;
pred    ( this S )            -> this S;
num     ( this S )            -> Natural;
```

其中Boolean为预定义的布尔类别，Natural为预定义的自然数类别。

<data type definition>中的<literal signature>以<Natural simple expression>的升序形式来命名。例如：

```
literals C = 3, A, B;
```

表示A<B和B<C。

比较运算符“<”（“>”、“<=”、“>=”）表示在两个文字的<Natural simple expression>之间进行标准的小于（大于、小于等于、大于等于）比较。运算符首先返回次序中的第一个数据项（具有最低<Natural simple expression>的文字）。运算符最后返回次序中的最后一个数据项（具有最高<Natural simple expression>的文字）。运算符pred返回前一个数据项（如果存在的话），否则返回最后一个数据项。运算符succ返回次序中的后一个数据项（如果存在的话），否则返回第一个数据项。运算符num返回对应文字<Natural simple expression>的自然数值。

如果<literal signature>是一个<regular expression>，那么它是枚举<literal name>集合（可能无穷多个）的缩写形式，如第12.1.9.1节所述。

12.1.7.2 结构数据类型

结构数据类型构造器通过生成给定类别集的笛卡尔积来规定类别的内容。结构类别的元素被称为结构。结构数据类型构造器隐含地定义操作，操作用于从部件类别元素构建结构、访问结构部件元素的投影操作以及更新结构部件元素的操作。

具体语法

```
<structure definition> ::=
                        [<visibility>] struct [<field list>] <end>
<field list> ::=
                        <field> { <end> <field> }*
<field> ::=
                        <fields of sort>
                    |   <fields of sort> optional
                    |   <fields of sort> <field default initialization>
<fields of sort> ::=
                        [<visibility>] <field name> { , <field name> }* <field sort>
```

<field default initialization> ::=
default <constant expression>

<field sort> ::=
<sort>

结构类别的每个<field name>都必须不同于同一<structure definition>的每个其他<field name>。

语义

一个<structure definition>定义一个结构类别，其所有元素为tuple，它们可以从数据项来构建，数据项属于在<field list>中给定的类别。结构类别的一个元素可以拥有像<field list>中的<field>那样多的部件元素，虽然如果对应的<field>已经用关键字**optional**进行声明，或尚未进行初始化，那么一个字段不可以与一个数据项相关。

<structure definition>重新定义（直接地或间接地）从Any继承的操作，如第12.1.5节所述。

<fields of sort>和<structure definition>中的<visibility>的含义在第12.1.9.3节中描述。

模型

<fields of sort>中包含带<field name>列表的<field>的<field list>是衍生的具体句法，其中该<field>由<field>列表替换，通过<end>进行分隔，这样，通过拷贝最初的<field>产生该列表中的每个<field>，并反过来对列表中的每个<field name>，用<field name>替换<field name>的列表。

结构定义是下列定义的衍生语法：

a） 运算符Make，用于创建结构；

b） 方法，用于修改结构，并访问结构的部件数据项；以及

c） 方法，用于测试结构中可选的部件数据项是否存在。

Make运算符的<arguments>包含<field sort>的列表，按其出现次序出现在字段列表中。Make运算符的结果<sort>是结构类别的类别标识符。Make运算符创建一个新的结构，并将每个字段与对应的形式参数的结果相关联。如果实际参数在应用Make运算符过程中被省略，那么对应的字段将不得到值；也就是说，它变成“未定义的”。

引入名为S的类别的<structure definition>指的是一组*Dynamic-operation-signatures*，等同于在下列<method list>中的显性定义，对其<field list>中的每个<field>：

virtual *field-modify-operation-name* (<field sort>) -> S;
virtual *field-extract-operation-name* -> <field sort>;
field-presence-operation-name -> Boolean;

其中Boolean为预定义的布尔类别，<field sort>为字段类别。

修改字段之隐含方法的名称*field-modify-operation-name*由字段名称与“Modify”连接而成。修改字段之隐含方法将字段与其变元*Expression*相关联起来。当<field sort>是一个<anchored sort>时，只有当变元*Expression*在类别上与该字段的<field sort>兼容时，该关联才会发生。否则，引起预定义的异常UndefinedField（见D.3.16）。

访问字段之隐含方法的名称*field-extract-operation-name*由字段名称与“Extract”连接而成。访问字段的方法将返回与该字段相关的数据项。如果在解释期间，结构的一个字段是“未定义的”，那么使用该方法访问结构的该字段将导致引起预定义的异常UndefinedField。

测试一个字段数据项是否存在之隐含方法的名称*field-presence-operation-name*由字段名称与“Present”连接而成。如果该字段是“未定义的”，那么测试一个字段数据项是否存在的隐含方法将预定义的布尔值“假”，否则返回预定义的布尔值“真”。只有在该<field>含有关键字**optional**时，才定义一个测试字段数据项是否存在的方法。

如果<field>用<field default initialization>进行定义，那么作为可选项，它是该<field>定义的衍生语法。当创建该类别的一个结构并且没有实际的变元提供给缺省字段时，在结构创建之后，通过相关的<constant expression>，立即增加一个字段的修改。

12.1.7.3 选择数据类型

选择数据类型构造器是一种缩写形式，用于定义一个带所有可选部件的结构类型，并确保每个结构数据项将总能确切地拥有一个部件数据项。选择数据类型因而模拟一个为部件类别元素直和的类别。

具体语法

<choice definition> ::=
[<visibility>] **choice** [<choice list>] <end>

<choice list> ::=
<choice of sort> { <end> <choice of sort> }*

<choice of sort> ::=
[<visibility>] <field name> <field sort>

选择类别的每个<field name>必须区别于同一<choice definition>的每个其他 <field name>。

语义

<choice definition>重新定义（直接地或间接地）自Any继承的各操作，如第12.1.5节所述。

<choice of sort> 和 <choice definition>中的<visibility>的含义在第12.1.9.3节中阐述。

模型

含有<choice definition>的数据类型定义是衍生语法，按以下步骤进行转化：设*Choice-name*为初始数据类型定义的<data type name>，则：

a） 增加一个带匿名名称*anon*的<value data type definition>和作为类型构造器的<structure definition>。在<value data type definition>中，对每个<choice of sort>，构造的<field>含有带关键字**optional**的、相当的<fields of sort>。

b） 用<literal list>增加一个带匿名名称*anonPresent*的<value data type definition>，<literal list>含有<choice list>中所有的<field name>，作为<literal name>。文字的次序同<field name>的规定次序。

c） 构建一个带匿名名称*anonChoice*的<data type definition>，如下所述：

object type *anonChoice*
struct
protected Present *anonPresent*;
protected Choice *anon*;
endobject type *anonChoice*;

如果初始的数据类型定义已定义一个对象类别。否则，<data type definition>为一个<value data type definition>。

d） 按如下所述构建<data type definition>：

object type *Choice-name* inherits *anonChoice* (*anonMake* = Make,
anonPresentModify = *PresentModify*,
anonPresentExtract = *PresentExtract*,
anonChoiceModify = *ChoiceModify*,
anonChoiceExtract = *ChoiceExtract*)
adding
operations
endobject type *Choice-name*;

如果初始的数据类型定义已定义一个对象类型，并且此处*operations*为<operations>，如下所定义。否则，<data type definition>为一个<value data type definition>。<renaming>重新命名提到的操作，操作从*anonChoice* 继承至匿名名称。

e） 对每个<choice of sort>，向操作的<operator list>增加一个<operation signature>，操作代表一个用于创建数据项的隐含运算符：

field-name (*field-sort*) -> *Choice-name*;

其中*field-name* 为<field name>，*field-sort* 为<choice of sort>中的<field sort>。用于创建数据项的隐含运算符创建一个称为*anonMake*的新结构，用最新创建的结构初始化字段选择，用<field name>来初始化，并向字段Present指配对应<field name>的文字。

f） 对每个<choice of sort>，向操作的<method list>增加一个<operation signature>，操作代表用于修改和访问数据项的各隐含方法：

virtual *field-modify* (*field-sort*) -> *Choice-name*;
virtual *field-extract* -> *field-sort;*
field-present -> Boolean;

其中*field-extract* 为以*anon隐含*的方法的名称，用于访问对应的字段，*field-modify*为以*anon*隐含的方法的名称，用于修改该字段，*field- present*为以*anon*隐含的方法的名称，用于测试一个字段数据项是否存在。对*field-extract* 和*field-present* 的调用提交给Choice。对*field-modify*的调用指配一个最新创建的、用<field name>初始化的结构给Choice，并将对应<field name>的文字指配给Present。

g） 向操作的<operator list>增加一个<operation signature>，操作代表一个用于获得当前出现在Choice中的数据项类别的隐含运算符：

PresentExtract (*Choice-name*) -> *anonPresent*;

PresentExtract返回与Present字段相关的值。

12.1.8 操作行为

<data type definition>允许为数据类型增加操作。操作行为可以用一种类似于值返回过程调用的方式进行定义。不过，数据类型的操作不得访问或修改代理输入队列的全局状态，它们在其中被调用。它们因此只能包含一个转变。

具体语法

```
<operation definitions> ::=
                        {       <operation definition>
                        |       <operation reference>
                        |       <external operation definition> }*
<operation definition> ::=
                        {<package use clause>}*
                        <operation heading>
                                [ <end> <entity in operation>+ ]
                                [ <comment body> ] <left curly bracket>
                                        <statement list>
                                 <right curly bracket>
<operation heading> ::=
                        { operator | method } <operation preamble> [<qualifier>] <operation name>
                                [<formal operation parameters>]
                                [<operation result>] [<raises>]
<operation identifier> ::=
                        [<qualifier>] <operation name>
<formal operation parameters> ::=
                        ( <operation parameters> {，<operation parameters> }* )
<operation parameters> ::=
                    [<argument virtuality>] <parameter kind> <variable name> {，<variable name>}* <sort>
<entity in operation> ::=
                        <data definition>
                    |   <variable definition>
                    |   <exception definition>
                    |   <select definition>
                    |   <macro definition>
<operation result> ::=
                        <result sign> [<variable name>] <sort>
<external operation definition> ::=
                        { operator | method } <operation signature> external <end>
```

如果在具有相同名称的同一类别中没有其他的<external operation definition>，并且存在一个<operation signature>，那么可以省略<external operation definition>中的<arguments>和<result>。在这种情况下，<arguments>和<result>源自<operation signature>。

对每个<operation signature>，至多只能提供一个对应的<operation definition>。

<operation definition>中的<statement>既不可以包含<imperative expression>，也不可以包含<identifier>，它们分别在嵌套的<operation definition>或<operation diagram>外面定义，除了对<synonym identifier>、<operation identifier>、<literal identifier>和<sort>。

如果当相应的on-exception子句没有激活异常句柄（即它没有被处理）时在操作中能够激起一个异常，那么<raises>必须提到该异常。如果在产生异常的操作内存在一个潜在的控制流，那么认为该异常不在操作内处理，并且在该控制流中没有任何激活的异常句柄来处理异常。

<variable name>列表被认为绑定得比<formal operation parameters>中的<operation parameters>列表更紧。

<operation diagram> ::=
<frame symbol> ***contains*** {
<operation heading>
{ <operation text area>* <operation body area> }***set*** }
[***is associated with*** <package use area>]

<operation body area> ::=
[<on exception association area>] <procedure start area>
{ <in connector area> | <exception handler area> }*

<operation text area> ::=
<text symbol> ***contains***
{ <data definition>
| <variable definition>
| <macro definition>
| <exception definition>
| <select definition> }*

<package use area>必须置于<frame symbol>的最上面。

<operation diagram>中的<start area>不得包含<virtuality>。

对每个<operation signature>，至多只能提供一个对应的<operation diagram>。

<operation definition>中的<operation body area>和<statement>既不可以包含<imperative expression>，也不可以包含<identifier>，它们分别在嵌套的<operation definition>或<operation diagram>外面定义，除了<synonym identifier>、<operation identifier>、<literal identifier>和<sort>。

语义

运算符是由结果标识之类别元素的构造器。它必须总返回一个值，或者一个最新构建的对象。相反地，方法可以返回一个现有的对象。

运算符不得修改以下自实际参数的引用可及的或实际参数自身可及的对象。如果在产生修改的运算符内部存在一个潜在的控制流，那么认为在运算符中对对象进行了修改。

一个操作定义是一个范围单元，它定义其自身的数据和变量，它们可以在<operation body area>内部进行处理。

如果<operation heading>以关键字**operator**开始，那么<operation definition>定义运算符的行为。如果<operation heading>以关键字**method**开始，那么<operation definition>定义方法的行为。

在<formal operation parameters>中引入的变量为运算符或方法的局部变量，可以在<operation body area>中进行修改。

<external operation definition>为运算符或方法，其行为不包括在SDL描述中（见第13节）。

模型

对每个没有相应<operation signature>的<operation definition>或<operation diagram>，构建一个<operation signature>。

<operation definition> 或 <operation diagram>分别转化为<procedure definition> 或 <procedure diagram>，有源自<formal operation parameters>的<procedure formal parameters>，有源自<operation result>的<result>。在<operation diagram>情况下，<procedure body area>源自<operation body area>。

对应作为结果的<procedure definition>或<procedure diagram>的*Procedure-definition*与由<operation signature>表示的*Operation-signature*相关。

如果<operation definition>或<operation diagram>定义一个方法，那么在转化至<procedure definition> 或<procedure diagram>期间，用做类别的变元<sort>将一个带<parameter kind> **in/out** 的初始参数插入到<formal operation parameters>中，类别由<data type definition>定义，<data type definition>组成范围单元，<operation definition>出现在该范围单元中。有关该插入参数的、<formal operation parameters>中的<variable name>是一个最新形成的匿名名称。

注 — 不可能为<literal signature>规定一个<operation definition> 。

如果任何<operation definition> 或 <operation diagram>包含非正式的文本，那么涉及相应运算符或方法应用的表达式的解释不由SDL正式定义，但可以通过解释器从非正式的文本做出确定。如果规定了非正式的文本，那么在SDL中未提供完整的正式规范。

12.1.9 额外的数据定义构件

本节介绍在数据中可能用到的更多的构件。

12.1.9.1 名称类

名称类是一种缩写形式，用于书写一个（可能为无穷大）文字名称或运算符名称的集合，通过正规的表达式进行定义。

<name class literal>是一种可选的方法，用于规定<literal name>。<name class operation>是一种可选的方法，用于规定空操作的<operation name>。

具体语法

```
<name class literal> ::=
                        nameclass <regular expression>
<name class operation> ::=
                        <operation name> in nameclass <regular expression>
<regular expression> ::=
                        <partial regular expression> { [or ] <partial regular expression> }*
<partial regular expression> ::=
                        <regular element> [ <Natural literal name> | <plus sign> | <asterisk> ]
<regular element> ::=
                        ( <regular expression> )
                    |   <character string>
                    |   <regular interval>
<regular interval> ::=
                        <character string> { <colon> | <range sign> } <character string>
```

由<regular expression>生成的名称必须满足有关名称或<character string>、<hex string>或<bit string>的词汇规则（见第6.1节）。

除去引导和结尾处的<apostrophe>后，<regular interval>中的两个<character string>长度都必须为1。

<name class operation>只能用在<operation signatare>中。包含<name class operation>的<operation signature>必须只能出现在<operator list>中，并不得含有<arguments>。

当<name class operation>的等价名称集中所含的一个名称作为<operation name>出现在<operation application>中时，它不得有<actual parameters>。

名称类的名称等价集定义为满足<regular expression>所规定之语法要求的名称集。<data type definition>中所含的<regular expression>等价名称集不得重叠。

模型

<name class literal>等同于抽象句法中的该名称集。当在<operation signature>中使用<name class operation>时，通过用名称等价集中的每个名称替换<operation signature>中的<name class operation>，来创建一个<operation signature>的集合。

为不带**or**的<partial regular expression>列表的<regular expression>规定，名称可以自字符生成，字符由第一个<partial regular expression>定义，后跟由第二个<partial regular expression>定义的字符。

当在两个<partial regular expression>之间规定一个**or**时，那么名称自这些<partial regular expression>中的第一个或第二个生成。**or**是比简单序列更紧密的绑定。

如果<regular element>后跟<Natural literal name>，那么<partial regular expression>等同于<regular element>，<regular element>重复<Natural literal name>所规定的次数。

如果<regular element>后跟“*”，那么<partial regular expression>等同于<regular element>，<regular element>重复0次或多次。

如果<regular element>后跟<plus sign>，那么<partial regular expression>等同于<regular element>，<regular element>重复1次或多次。

为相等<regular expression>的<regular element>定义由<regular expression>定义的字符序列。

为<character string>的<regular element>定义在字符串中给出的字符序列（省略引用）。

为<regular interval>的<regular element>定义所有由<regular interval>规定的字符为可选的字符序列。根据字符类别的定义，由<regular interval>定义的字符为所有大于或等于第一个字符并小于或等于第二个字符的字符（见D.2）。

根据字符类别的排序，由<name class literal>生成的名称按字母次序进行定义。认为字符是大小写敏感的，并且认为字的一个真实前缀小于整个字。

注 — 附件D中有例子。

12.1.9.2 名称类映射

名称类映射是一种缩写形式，用于定义遍及<name class operation>中所有名称的操作定义数量（可能无穷大）。名称类映射允许为通过<name class operation>定义的运算符和方法定义行为。当在<operation definitions>或 <operation diagram>中使用<operation name>时（<operation name>出现在嵌套的<data type definition>的<operation signature>内的<name class operation>中），发生名称类映射。

名称类映射中的拼法项指的是包含名称拼法的字符串。该机制允许使用字符串操作来定义名称类映射。

具体语法

<spelling term> ::=

spelling (<operation name>)

如果发生名称类映射，那么仅在<operation definition> 或 <operation diagram>中，<spelling term>是合法的具体句法。

模型

名称类映射是<operation definition>集或<operation diagram>集的一种缩写形式。通过用相应<name class operation>的名称等价集中的每个名称替换<operation definition>中<operation name>的每个事件，来从<operation definition>生成<operation definition>集。

即使不允许将<string name>作为具体句法中的<operation name>，也可以认为衍生的<operation definition>和<operation diagram>是合法的。

将衍生的<operation definition>加入到同一<data type definition>中的<operation definitions>上（如果有的话）。

如果<operation definition>或<operation diagram>包含一个或多个<spelling term>，那么用字符串文字替换每个<spelling term>（见D.3.4）。

如果在上述转化期间，<spelling term>中的<operation name>已由<operation name>替换，那么<spelling term>是源自<operation name>的字符串的一种缩写形式。字符串包含<operation name>的拼法。

如果在上述转化期间，<spelling term>中的<operation name>已由<string name>替换，那么<spelling term>是源自<string name>的字符串的一种缩写形式。字符串包含<string name>的拼法。

12.1.9.3 有限的视见度

具体语法

<visibility> ::=

public | **protected** | **private**

在含有<data type specialization>的<data type definition>中，<visibility>不得位于<literal list>、<structure definition>或<choice definition>之前。<visibility>不得在<operation signature>中使用，<operation signature>重新定义一个集成的操作特征。

语义

<visibility> 控制文字名称或操作名称的视见度。

当<literal list>之前为<visibility>时，该<visibility>适用于所有的<literal signature>。当<structure definition> 或<choice definition>之前为<visibility>时，该<visibility>适用于所有的隐含<operation signatures>。

当<fields of sort> 或 <choice of sort>之前为<visibility>时，该<visibility>适用于所有的隐含<operation signatures>。

如果在<visibility>中<literal signature> 或 <operation signature>含有关键字**private**，那么源自该<operation signature>的*Operation-name*只在包含<operation signature>的<data type definition>的范围内可见。当特殊化包含此类<operation signature>的<data type definition>时，<operation signature>中的<operation name>被隐性地重命名为一个匿名名称。当<operation signature>和对应的操作定义通过特殊化继承时，<operation definitions> 或 <operation diagram>（对应该<operation signature>）内该<operation name>的每个事件都被重命名为同一匿名名称。

注1 — 结果是，由本<operation signature>定义的运算符或方法只能用在数据类型定义内的操作应用中，数据类型定义最先定义本<operation signature>，但子类型不在其中。

如果<literal signature> 或 <operation signature>在<visibility>中含有关键字**protected**，那么源自本<operation signature>的 *Operation-name*只在含有<operation signature>的<data type definition>的范围内可见。

注2 — 由于继承的运算符和方法被拷贝到了子类型体中，因此由本<operation signature>定义的运算符或方法可以在任何<data type definition>的范围内进行访问，<data type definition>是最先定义本<operation signature>的<data type definition>的一个子类型。

注3 — 如果<literal signature>或<operation signature>不包含<visibility>，那么在<sort name>可见的任何地方（<sort name>在嵌套的<data type definition>中定义），源自本<operation signature>的*Operation-name* 都可见。

模型

如果<literal signature> 或 <operation signature>在<visibility>中含有关键字**public**，那么它是有关无保护特征的衍生语法。

12.1.9.4 共型

一个共型规定一个类别元素的子集。用做类别的共型与共型引用的类别具有相同的语义，除非检查结果是：数据项属于类别元素的特定子集。

抽象语法

Syntype-identifier = *Identifier*

Syntype-definition :: *Syntype-name*
Parent-sort-identifier
Range-condition
Syntype-name = *Name*
Parent-sort-identifier = *Sort-identifier*

具体语法

<syntype> ::=
<syntype identifier>

<syntype definition> ::=
{<package use clause>}*
syntype <syntype name> <equals sign> <parent sort identifier>
[<comment body>] <left curly bracket>
[{ <default initialization> [[<end>] <constraint>] | <constraint> } <end>]
<right curly bracket>
| {<package use clause>}*
<type preamble> <data type heading> [<data type specialization>]
[<comment body>] <left curly bracket>
<data type definition body> <constraint> <end>
<right curly bracket>

<parent sort identifier> ::=
<sort>

<syntype>是<sort>的二选一项。

带关键字**value type** 或 **object type** 的<syntype definition>是下面定义之语法的衍生语法。

具体句法中带关键**syntype** 的<syntype definition>对应抽象句法中的*Syntype-definition*。

当<syntype identifier>用做<arguments>中的<sort>时，当定义一个操作时，相应*Formal-argument*s的类别为共型的*Parent-sort-identifier* 。

当<syntype identifier>作为操作的结果使用时，*Result*的类别为共型的*Parent-sort-identifier*。

当<syntype identifier>作为名称的标识符使用时，*Qualifier*为共型的*Parent-sort-identifier*。

如果使用关键字**syntype**并省略<constraint>，那么共型的<syntype identifier>在抽象句法中，表示为*Parent-sort-identifier*。

如果<constraint>可被解释为属于<default initialization>或<syntype definition>，那么应认为它是<default initialization>的一部分。

语义

共型定义用于定义一个共型，它引用一个类别标识符和一个约束。规定一个共型标识符等同于规定共型的父类别标识符，下列情况除外：

a） 对以共型声明之变量的指配（见第12.3.3节）；

b） 信号的输出，如果为信号规定的类别之一为共型（见第10.3和11.13.4节）；

c） 当为参数变量中的过程规定的类别之一为共型时，调用一个过程（见第9.4和11.13.3节）；

d） 当为代理参数规定的类别之一为共型时，创建一个代理（见第9.3和10.3节）；

e） 信号和其中之一变量的输入，它与拥有共型类别的输入相关（见第11.3节）；

f） 调用操作应用，它拥有一个定义为变元类别或结果类别的共型（见第12.2.7节）；

g） 设置或重新设置有关计时器的条款或主动表达式，计时器定义中的一个类别为共型（见第11.15和12.3.4.4节）；

h） 远程变量或远程过程定义，如果用于衍生或隐含信号的一个类别为共型（见第10.5和10.6节）；

i） 与实际正文参数匹配的、在<procedure signature>中带in/out或out参数的过程形式正文参数，其中对应的形式参数或<procedure signature>中的in/out或out参数为共型；

j） <any expression>，其中结果将位于范围内（见第12.3.4.5节）；

k） 引起一个异常，如果为异常规定的类别之一为共型（见第11.12.2.5节）。

当依据<syntype identifier>规定共型时，两个共型不得相互定义。

一个共型有一个类别，它是在共型定义中给出的父类别标识符标识的类别。

共型有一个*Range-condition*，用于约束类别。如果使用范围条件，那么类别被约束为由共型定义常量规定的数据项的集合。如果使用大小约束，那么类别被约束为包含由大小约束给出的数据项。

模型

通过纳入一个<constraint>，可以将带关键字**value type** 或 **object type**的<syntype definition>与<data type definition>区别开来。这样一个<syntype definition>是一种缩写形式，用于引入一个带匿名名称的<data type definition>，后跟一个带关键字**syntype**的<syntype definition>，它基于该匿名地命名的类别和纳入的<constraint>。

12.1.9.5 约束

抽象语法

Range-condition :: *Condition-item-**set***
Condition-item = *Open-range* | *Closed-range*
Open-range :: *Operation-identifier*
Constant-expression
Closed-range :: *Open-range*
Open-range

具体语法

<constraint> ::=
constants (<range condition> **)**
| <size constraint>

<range condition> ::=
<range> { **,** <range> }*

<range> ::=
<closed range>
| <open range>

<closed range> ::=
<constant> { <colon> | <range sign> } <constant>

<open range> ::=
<constant>
| { <equals sign>
| <not equals sign>
| <less than sign>
| <greater than sign>
| <less than or equals sign>
| <greater than or equals sign> } <constant>

<size constraint> ::=
size (<range condition> **)**

<constant> ::=
<constant expression>

符号“<”必须只能用在<range condition>的具体句法中，如果该符号已用一个<operation signature>进行定义：

"<" (P，P) -> <<**package** Predefined>>Boolean;

其中P为共型的类别，对符号（分别为“<=”、“>”、“>=”）也类似。这些符号代表*Operation-identifier*。

如果用一个<operation signature>来定义符号“<=”，那么必须只能使用<closed range>：

"<=" (P，P) -> <<**package** Predefined>>Boolean;

其中P为共型的类别。

<range condition>中的<constant expression>必须与共型拥有相同的类别。

＜size constraint＞必须只能用在＜range condition＞的具体句法中，如果该符号长度已用一个＜operation signature＞进行定义：

Length (P) -> <<**package** Predefined>>Natural;

其中P为共型的类别。

语义

一个约束定义一个范围检查。当共型对共型类别有额外的语义时，使用范围检查（见第12.3.1、12.1.9.4节，以及对共型有不同语义的情况 — 见第12.1.9.4语义一节中a）至k）项中所引用的子节）。范围检查也用于确定一个决定的解释（见第11.13.5节）。

范围检查是由范围条件或大小约束形成之操作的应用。对共型范围检查，该操作的应用必须相当于预定义的布尔值“真”；否则，引起预定义的异常OutOfRange（见D.3.16）。范围检查按下列步骤获得：

a） ＜range condition＞中的每个＜open range＞ 或 ＜closed range＞在*Condition-item*中有一个相应的*Open-range*（预定义的布尔**or**）或*Closed-range*（预定义的布尔**and**）。

b） 形式＜constant＞的＜open range＞相当于形式= ＜constant＞的＜open range＞。

c） 对一个给定的表达式，A，则：

1） 形式= ＜constant＞、/= ＜constant＞、< ＜constant＞，＜less than or equals sign＞ ＜constant＞，> ＜constant＞和＜greater than or equals sign＞ ＜constant＞的＜open range＞在形式A= ＜constant＞、A/= ＜constant＞、A< ＜constant＞，A＜less than or equals sign＞ ＜constant＞，A> ＜constant＞和A＜greater than or equals sign＞ ＜constant＞的范围检查中分别有子表达式；

2） 形式*first* ＜constant＞ : *second* ＜constant＞ 的＜closed range＞在形式*first* ＜constant＞ ＜less than or equals sign＞ A **and** A ＜less than or equals sign＞ *second* ＜constant＞的范围检查中有子表达式，其中**and**对应预定义的布尔**and**；

3） ＜size constraint＞在形式Length(A) = ＜range condition＞的范围检查中有子表达式。

d） 对遍及*Condition-item*-**set**中所有数据项的分布操作，存在一个预定义的布尔**or**操作。范围检查是这样一个表达式，它由源自＜range condition＞的所有子项的预定义布尔**or**形成。

如果共型不用＜constraint＞来规定，那么范围检查为预定义布尔值“真”。

12.1.9.6 同义词定义

同义词为常数表达式提供了一个名称，它代表类别的其中一个数据项。

具体语法

＜synonym definition＞ ::=
synonym ＜synonym definition item＞ { **,** ＜synonym definition item＞ }*＜end＞

＜synonym definition item＞ ::=
＜internal synonym definition item＞
| ＜external synonym definition item＞

＜internal synonym definition item＞ ::=
＜synonym name＞ [＜sort＞] ＜equals sign＞ ＜constant expression＞

＜external synonym definition item＞ ::=
＜synonym name＞ ＜predefined sort＞ = **external**

具体句法中的＜constant expression＞表示抽象句法中的一个*Constant-expression*，如第12.2.1节定义。

如果规定一个＜sort＞，那么＜constant expression＞的结果有一个＜sort＞的静态类别。它必须有可能使＜constant expression＞拥有该类别。

如果不能惟一地确定＜constant expression＞的类别，那么必须在＜synonym definition＞中规定一个类别。

＜constant expression＞不必指向由＜synonym definition＞定义的同义词，不论是直接地还是间接地（通过另一个同义词）。

一个＜external synonym definition item＞定义一个＜synonym＞，其结果不在一个规范中定义（见第13节）。

语义

同义词代表的结果由正文确定，同义词定义在该正文中出现。

如果在同义词的正文中不能惟一地确定常数表达式的类别，那么通过<sort>给出类别。

一个同义词有一个结果，它是同义词定义中常数表达式的结果。

一个同义词有一个类别，它是同义词定义中常数表达式的类别。

12.2 数据的被动使用

以下各节定义如何在表达式中解释类别、文字、运算符、方法和同义词。

12.2.1 表达式

抽象语法

Expression = *Constant-expression*
| *Active-expression*

Constant-expression = *Literal*
| *Conditional-expression*
| *Equality-expression*
| *Operation-application*
| *Range-check-expression*

Active-expression = *Variable-access*
| *Conditional-expression*
| *Operation-application*
| *Equality-expression*
| *Imperative-expression*
| *Range-check-expression*
| *Value-returning-call-node*
| *State-expression*

具体语法

为简化描述起见，在*Constant-expression*和*Active-expression*的具体句法之间不做区别。

```
<expression> ::=
        <expression0>
    |   <range check expression>
<expression0>
        <operand>
    |   <create expression>
    |   <value returning procedure call>
<operand> ::=
        <operand0>
    |   <operand> <implies sign> <operand0>
<operand0> ::=
        <operand1>
    |   <operand0> { or | xor } <operand1>
<operand1> ::=
        <operand2>
    |   <operand1> and <operand2>
<operand2> ::=
        <operand3>
    |   <operand2> { <greater than sign>
                   | <greater than or equals sign>
                   | <less than sign>
                   | <less than or equals sign>
                   | in } <operand3>
    |   <equality expression>
```

<operand3> ::=
        <operand4>
    | <operand3> { <plus sign> | <hyphen> | <concatenation sign> } <operand4>
<operand4> ::=
        <operand5>
    | <operand4> { <asterisk> | <solidus> | **mod** | **rem** } <operand5>
<operand5> ::=
        [<hyphen> | **not**] <primary>
<primary> ::=
        <operation application>
    | <literal>
    | **(** <expression> **)**
    | <conditional expression>
    | <spelling term>
    | <extended primary>
    | <active primary>
    | <synonym>
<active primary> ::=
        <variable access>
    | <imperative expression>
<expression list> ::=
        <expression> { **,** <expression> }*
<simple expression> ::=
        <constant expression>
<constant expression> ::=
        <constant expression0>

<expression0>不包含任何<active primary>，<create expression>或<value returning procedure call>是一个<constant expression0>。<constant expression0>代表抽象句法中的*Constant-expression*。每个<constant expression>在系统初始化期间被解释一次，并保留解释的结果。解释期间，无论何时需要<constant expression>的值，都使用该计算值的一个完整拷贝。

不是<constant expression>的一个<expression>代表一个*Active-expression*。

如果<expression>包含<extended primary>，那么在考虑与抽象句法的关系之前，在具体句法层面上对<extended primary>进行替换，如第12.2.4节所定义的那样。

<operand>、<operand1>、<operand2>、<operand3>、<operand4> 和 <operand5>为运算符和方法名称提供了特殊的语法形式。该特殊语法的引入，例如，使得算术操作和布尔操作可以拥有其通常语法形式。也就是说，使用者可以写成“（1＋1）=2”，而不必强制使用，例如，equal(add(1,1),2)。对每个操作哪个类别是合法的决定于数据类型定义。

<simple expression>必须只能包含文字、运算符和在预定义包中定义的方法，如附件D所定义的那样。

语义

表达式中的中缀运算符或方法有运算符或方法的正常语义，但带中缀或引用前缀的语法。

表达式中的单运算符或方法有运算符或方法的正常语义，但带前缀或引用前缀的语法。

中缀运算符或方法拥有优先次序，以决定运算符或方法的绑定。当绑定不明确时，从左到右进行绑定。

当解释表达式时，它返回一个数据项（一个值、对象或pid）。返回的数据项指的是表达式的结果。

表达式的（静态）类别为数据项的类别，它将通过对表达式的解释来返回，通过对规范的分析来确定，不考虑解释语义。表达式的动态类别为表达式结果的类别。因多态指配，主动表达式的静态类别和动态类别可以不同（见第12.3.3节）。对常数表达式，表达式的动态类别为其静态类别。

注 — 为避免繁琐的文本，“类别”这词指的总是一个静态类别，除非之前有“动态”这词。为清楚起见，在某些情况下显性地写成“静态类别”。

一个简单的表达式是一个*Constant-expression*。

模型

该形式的表达式为：

<expression> <infix operation name> <expression>

是下列的衍生语法：

<quotation mark> <infix operation name> <quotation mark> (<expression>，<expression>)

其中<quotation mark> <infix operation name> <quotation mark> 代表一个*Operation-name*。

同样，

<monadic operation name> <expression>

是下列的衍生语法：

<quotation mark> <monadic operation name> <quotation mark> (<expression>)

其中<quotation mark> <monadic operation name> <quotation mark>代表一个*Operation-name*。

12.2.2 文字

抽象语法

Literal :: *Literal-identifier*
Literal-identifier = *Identifier*

*Literal-identifier*标识一个*Literal-signature*。

具体语法

<literal> ::=
<literal identifier>
<literal identifier> ::=
[<qualifier>= <literal name>

无论何时规定<literal identifier>，以同样的方法衍生*Literal-identifier*中的惟一的*Literal-name*，结果类别源自正文。*Literal-identifier*源自正文（见第6.3节），因此，如果<literal identifier>过载（即同一名称用于多个文字或操作），那么*Literal-name*用同一名称和与文字一致的结果类别来标识一个可见的文字。具有相同<name>但结果类别不同的两个文字具有不同的*Literal-name*。

必须有可能将每个未限定的<literal identifier>准确地绑定至一个已定义的*Literal-identifier*，它满足构件中的各条件，<literal identifier>在该构件中使用。

无论何处<literal identifier>的<qualifier>含有一个带关键字**type**的<path item>，在该关键字之后的<sort name>不是*Literal-identifier*的*Qualifier*的一部分，但它用于生成*Identifier*的惟一的*Name*。在这种情况下，从关键字**type**之前的<path item>的列表来生成*Qualifier*。

语义

Literal 返回对应其*Literal-signature*的惟一的数据项。

<literal>的类别是其*Literal-signature*中的*Result* 。

12.2.3 同义词

具体语法

<synonym> ::=
<synonym identifier>

语义

同义词是缩写形式，用于表示在任何地方定义的表达式。

模型

<synonym>代表由<synonym definition>定义的<constant expression>，<synonym definition>由<synonym identifier>标识。根据<synonym definition>的正文，在<constant expression>中使用的<identifier>代表在抽象句法中的*Identifier*。

12.2.4 扩展的首位

扩展的首位是一种缩写形式的语法符号。不过，除了特殊的语法形式，扩展的首位没有特殊的属性，并表示一个操作及其参数。

具体语法

<extended primary> ::=
<indexed primary>
| <field primary>
| <composite primary>

<indexed primary> ::=
<primary> (<actual parameter list>)
| <primary> <left square bracket> <actual parameter list> <right square bracket>

<field primary> ::=
<primary> <exclamation mark> <field name>
| <primary> <full stop> <field name>
| <field name>

<field name> ::=
<name>

<composite primary> ::=
[<qualifier>] <composite begin sign> <actual parameter list> <composite end sign>

模型

<indexed primary>是下列的衍生具体句法：

<primary> <full stop> Extract (<actual parameter list>)

根据第12.2.1节，抽象句法由该具体表达式决定。

<field primary>是下列的衍生具体句法：

<primary> <full stop> *field-extract-operation-name*

其中*field-extract-operation-name*由字段名称和“Extract”以该次序连接而成。根据第12.2.1节，抽象句法由该具体表达式决定。依据本模型的转换在修改第12.1.4节中方法的特征之前进行实施。

当<field primary>具有<field name>形式时，它是下列的衍生语法：

this ! <field name>

<composite primary>是下列的衍生具体句法：

<qualifier> Make (<actual parameter list>)

如果存在任何实际的参数，或：

<qualifier> Make

否则，仅当它出现在<composite primary>时，才插入<qualifier>。根据12.2.1节，抽象句法由该具体表达式决定。

12.2.5 等同表达式

抽象语法

Equality-expression	::	*First-operand*
		Second-operand
First-operand	=	*Expression*
Second-operand	=	*Expression*

Equality-expression 表示其*First-operand* 的引用或值和其*Second-operand*的引用或值相等。

具体语法

<equality expression> ::=

＜operand2＞ { ＜equals sign＞ | ＜not equals sign＞ } ＜operand3＞

仅当它的一个操作数的类别在类别上兼容于其他操作数的类别时，＜equality expression＞才是合法的具体句法。

语义

解释*Equality-expression* 之后继续进行其*First-operand* 和*Second-operand*的解释。

如果在解释之后，两个操作数都为对象，那么*Equality-expression*表示引用等同性。当且仅当两个操作数都为Null或都引用同一对象数据项时，它才返回预定义布尔值“真”。

如果在解释之后，两个操作数都为pid，那么*Equality-expression*表示代理等同性。当且仅当两个操作数都为Null或都引用同一代理数据项时，它才返回预定义布尔值“真”。

如果在解释之后，其中一个操作数为值，那么*Equality-expression*表示值的等同性，如下所述：

a） 如果*First-operand*的动态类别在类别上兼容于*Second-operand*的动态类别，那么＜equality expression＞将equal运算符的应用结果返回给*First-operand*和*Second-operand*，其中equal是*Operation-identifier*，由＜operation application＞中的＜operation identifier＞表示：

equal(＜operand2＞，＜operand3＞)

b） 否则，＜equality expression＞将equal运算符的应用结果返回给*Second-operand*和*First-operand*，其中equal是*Operation-identifier*，由＜operation application＞中的＜operation identifier＞表示：

equal(＜operand3＞，＜operand2＞)

具体句法形式：

＜operand2＞ ＜not equals sign＞ ＜operand3＞

是下列的衍生具体句法：

not (＜operand2＞ **=** ＜operand3＞ **)**

其中**not**是预定义布尔数据类型的一个操作。

12.2.6 条件表达式

抽象语法

Conditional-expression	::	*Boolean-expression* *Consequence-expression* *Alternative-expression*
Boolean-expression	=	*Expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

一个*Conditional-expression* 是一个*Expression*，它解释为*Consequence-expression* 或*Alternative-expression*。

Consequence-expression 的类别必须等同于*Alternative-expression*的类别。

具体语法

＜conditional expression＞ ::=
if ＜Boolean expression＞
then ＜consequence expression＞
else ＜alternative expression＞
fi

＜consequence expression＞ ::=
＜expression＞

＜alternative expression＞ ::=
＜expression＞

＜consequence expression＞的类别必须等同于＜alternative expression＞的类别。

语义

一个条件表达式代表一个*Expression*，它可以被解释为*Consequence-expression*或*Alternative-expression*。

如果*Boolean-expression*返回预定义的布尔值“真”，那么不对*Alternative-expression*进行解释。如果*Boolean-expression*返回预定义的布尔值“假”，那么不对*Consequence-expression*进行解释。

一个条件表达式有一个类别，它是结果表达式的类别（并且也是二选一表达式的类别）。

条件表达式的结果是*Consequence-expression*或*Alternative-expression*的解释结果。

条件表达式的静态类别是*Consequence-expression*的静态类别（它也是*Alternative-expression*的类别）。条件表达式的动态类别是条件表达式解释结果的动态类别。

12.2.7 操作应用

抽象语法

Operation-application :: *Operation-identifier* [*Expression*]*

Operation-identifier = *Identifier*

*Operation-identifier*表示一个*Operation-signature*，它或者是一个*Static-operation-signature*，或者是一个*Dynamic-operation-signature*。在*Operation-identifier* 之后的*Expression*列表中的每个*Expression*在类别上都必须兼容于*Operation-signature*的*Formal-argument*列表中的对应类别（按位置）。

每个*Operation-signature*都与一个*Procedure-definition*相关，如第12.1.8节所述。

按位置与*Procedure-definition*（与*Operation-signature*相关）中的*Inout-parameter* 或 *Out-parameter*对应的每个*Expression*都必须是一个*Variable-identifier*，它与*Operation-signature*的*Formal-argument*列表中的对应类别（按位置）具有相同的*Sort-reference-identifier*。

具体语法

```
<operation application> ::=
                        <operator application>
                    |   <method application>
<operator application> ::=
                        <operation identifier> [<actual parameters>]
<method application> ::=
                        <primary> <full stop> <operation identifier> [<actual parameters>]
```

无论何时规定<operation identifier>，*Operation-identifier*中的惟一的*Operation-name*也以同样的方式产生。变元类别列表源自实际参数，结果类别源自正文（见第6.3节）。因此，如果<operation name>过载（也就是说，同一名称用在了多个文字或操作上），那么*Operation-name*用同一名称、变元类别、与操作应用一致的结果类别来确定一个可见的操作。具有相同<name>、但一个或多个变元类别或结果类别不同的两个操作具有不同的*Operation-name*。

必须有可能将每个未限定的<operation identifier>确切地绑定至一个定义的*Operation-identifier*上，它满足构件中的条件，在该构件中使用<operation identifier>。

当操作应用拥有下列语法形式时：

<operation identifier> [<actual parameters>]

则在从正文衍生*Operation-identifier* 期间，形式：

this <full stop> <operation identifier> [<actual parameters>]

也被考虑。在尝试通过正文进行解析之前，应用第12.3.2节中的模型。

一旦<operation identifier>的<qualifier>包含一个带关键字**type**的<path item>，则该关键字之后的<sort name>不作为*Operation-identifier*的*Qualifier*的一部分，但用于生成*Identifier*的惟一的*Name*。

如果<expression>的括起来的列表中的所有<expression>都是<constant expression>，那么<operation application>代表一个*Constant-expression*，如第12.2.1节定义。

只有当<operation identifier>代表一个方法时，<method application>才是合法的具体句法。

不能省略＜actual parameters＞中的＜expression＞，＜actual parameters＞对应*Procedure-definition*中的*Inout-parameter* 或 *Out-parameter*，*Procedure-definition*与*Operation-signature*相关，并必须是一个＜variable access＞ 或＜extended primary＞。

注 — 如果已经省略所有的实际参数，那么可以省略＜operation application＞中的＜actual parameters＞。

语义

通过正文进行解析（见第6.3节）确保选中一个操作，这样，实际变元的类型将为兼容于形式变元类型的成对类别。

带*Operation-identifier*（表示一个*Static-operation-signature*）的操作应用通过将解释传送给*Procedure-definition*来解释，*Procedure-definition*与*Operation-signature*相关，并对过程图进行解释（说明包含在第9.4节中）。

带*Operation-identifier*（表示一个*Dynamic-operation-signature*）的操作应用通过以下步骤来解释：

a） 解释实际参数；

b） 如果对应*Virtual-argument*的实际参数结果为Null，那么引起预定义的异常InvalidReference；

c） 所有的*Dynamic-operation-signature*都被归入一个集合中，这样，由*Operation-name*形成的操作特征（*Operation-name*源自＜operation identifier＞中的＜operation name＞）和实际参数的解释结果在类别上兼容于候选的*Dynamic-operation-signature*；

d） 选择惟一的Dynamic-operation-signature，它在类别上兼容于该集合中的所有其他Dynamic-operation-signature；以及

e） 解释传送给与所选Operation-signature相关的Procedure-definition，并对过程图进行解释（说明包含在第9.4节中）。

通过以下要求来确保存在这样一个惟一的特征，即*Dynamic-operation-signature*的集合形成一个框架（见第12.1.4节）。

在对操作自身进行解释之前，对*Operation-application*中实际参数*Expression*的列表按从左到右的次序进行解释。

如果操作特征的变元类别是一个共型，那么在第12.1.9.5节中定义的范围检查适用于*Expression*的结果。如果在解释时得到的范围检查结果为预定义的布尔值“假”，那么引起预定义的异常OutOfRange（见D.3.16）。

当结束对被调用过程的解释时，继续对含有＜operation application＞的转换的解释。操作应用结果为解释被引用过程定义所返回的结果。

如果操作特征的结果类别是一个共型，那么在第12.1.9.5节中定义的范围检查适用于操作应用的结果。如果在解释时得到的范围检查结果为预定义的布尔值“假”，那么引起预定义的异常OutOfRange（见D.3.16）。

＜operation application＞有一个类别，它是结果的类别，通过对相关过程的解释来获得。

模型

具体句法形式：

＜expression＞ ＜full stop＞ ＜operation identifier＞ [＜actual parameters＞]

是下列的衍生具体句法：

＜operation identifier＞ *new-actual-parameters*

其中*new-actual-parameters*是仅包含＜expression＞的＜actual parameters＞，如果＜actual parameters＞不出现的话；否则，通过在＜actual parameters＞中的第一个可选表达式之前插入＜expression＞来获得*new-actual-parameters* 。

如果＜method application＞的＜primary＞不是一个变量或**this**，那么存在一个至隐含变量的＜primary＞隐含指配，用操作的第一个参数的类别（即方法类别）。指配置于动作之前，在其中出现＜method application＞。隐含变量替换＜method application＞中的＜primary＞。

12.2.8 范围检查表达式

抽象语法

Range-check-expression :: *Range-condition Expression*

具体语法

<range check expression> ::=
<operand2> **in type** { <sort identifier> <constraint> | <sort> }

<operand2>的类别必须等同于由<sort identifier>或<sort>确定的类别。

语义

一个*Range-Check-Expression*是一个预定义布尔类别的表达式，如果*Expression*满足对应<constraint>的*Range-condition*，如第12.1.9.5节所定义，那么其结果为“真”；否则，其结果为“假”。

模型

规定<sort>是规定数据类型的<constraint>（定义<sort>）的衍生语法。如果该数据类型不以<constraint>定义，那么不对<range check expression>进行评估，并且<range check expression>是用于规定预定义布尔值“真”的衍生语法。

12.3 数据的主动使用

本节定义数据和声明变量的使用，定义如何解释一个涉及变量的表达式，以及定义从基本系统获得结果的祈使表达式。

一个变量有一个类别和该类别相关的数据项。与变量相关的数据项可以通过为变量指配一个新的数据项来改变。与变量相关的数据项可以通过访问变量来用在一个表达式中。

任何包含一个变量的表达式都被认为是“主动的”，原因是通过解释表达式而获得的数据项可以根据最后指配给变量的数据项的不同而不同。解释一个动态表达式的结果将依赖于系统的当前状态。

12.3.1 变量定义

变量具有相关的数据项，或是“未定义的”。

抽象语法

Variable-definition :: *Variable-name*
Sort-reference-identifier
[*Constant-expression*]
Variable-name = *Name*

如果*Constant-expression*出现，那么它必须是下列之一：

1） *Sort-reference-identifier* 表示的相同类别；或

2） 如果表示的类别是一个对象类别OS，那么通过**value** OS来表示类别；或

3） 如果表示的类别是一个值类别VS，那么通过**object** VS来表示类别。

具体语法

<variable definition> ::=
dcl [**exported**] <variables of sort> {，<variables of sort> }* <end>

<variables of sort> ::=
<variable name> [<exported as>] { ，<variable name> [<exported as>] }*
<sort> [<is assigned sign> <constant expression>]

<exported as> ::=
as <remote variable identifier>

<exported as>只能用于在其<variable definition>中带**exported**的变量。在一个代理中的两个出口变量不同提到同一<remote variable identifier>。

*Constant-expression*由下列表示：

a） 如果在<variable definition>中给出了一个<constant expression>，那么为该<constant expression>；

b） 否则，如果定义<sort>的数据类型有一个<default initialization>，那么为<default initialization>的<constant expression>。

否则，不出现*Constant-expression*。

语义

在创建变量和*Constant-expression*出现时，变量与下列相关：

1） Constant-expression的结果，如果变量的类别与Constant-expression的类别相同时；

2） 引用Constant-expression的对象（不同于任何其他对象），如果变量有一个对象类别，Constant-expression有一个值类别；

3） Constant-expression引用的值，如果变量有一个值类别，Constant-expression有一个对象类别。

否则，如果没有*Constant-expression*应用，那么变量没有相关的数据项；也就是说，变量是“未定义的”。

如果*Sort-reference-identifier*是一个 *Syntype-identifier*，那么出现*Constant-expression*，*Constant-expression* 的结果不满足*Range-condition*，引起预定义的异常OutOfRange（见D.3.16）。

关键字**exported**允许变量作为出口变量来使用，如第10.6节所设计的那样。

12.3.2 变量访问

抽象语法

Variable-access = *Variable-identifier*

具体语法

<variable access> ::=
<variable identifier>
| **this**

this必须只能出现在方法定义中。

语义

一个变量访问被解释为给出与标识之变量相关的数据项。

一个变量访问有一个静态的类别，它是由变量访问标识之变量的类别。它有一个动态类别，是与标识之变量相关的数据项的动态类别。

一个变量访问有一个结果，它是最后与变量相关的数据项。如果变量是“未定义的”，那么解释一个有关预定义的异常UndefinedVariable（见D.3.16）的激发节点。

模型

根据第12.1.8节，使用关键字**this**的<variable access>由引入的匿名替代，作为<arguments>中主导参数的名称。

12.3.3 指配和指配尝试

一个指配创建一个从变量到表达式解释结果的关联。在一次指配尝试中，只有当变量的动态类别和表达式的动态类别兼容时才创建该关联。

抽象语法

Assignment :: *Variable-identifier*
Expression
Assignment-attempt :: *Variable-identifier*
Expression

在一个指配中，*Expression*的类别必须在类别上兼容于*Variable-identifier*的类别。

在一次*Assignment-attempt*中，*Variable-identifier* 的类别必须在类别上与*Expression*的类别兼容。

如果变量用一个*Syntype*声明并且*Expression*是一个*Constant-expression*，那么在上面第12.1.9.5节中定义的、适用于*Expression*的范围检查必须是预定义的布尔值“真”。

具体语法

<assignment> ::=
<variable> <is assigned sign> <expression>

<variable> ::=
<variable identifier>
| <extended variable>

如果<variable>是一个<variable identifier>，那么在具体句法中的<expression>代表在抽象句法中的*Expression*。<extended variable>是衍生的语法，并在考虑与抽象句法的关系之前，在具体句法层面被替换为如第12.3.3.1节中所定义的那样。

如果<variable identifier>已用对象类别声明，并且<expression>的类别是<variable identifier>类别的一个父类别（直接的或间接的），那么<assignment>代表一次*Assignment-attempt*。否则，<assignment>代表一个*Assignment*。

语义

一个指配被解释为创建一个指配中标识的变量与指配中表达式的结果之间的关联。变量的前一个关联被遗弃。

该关联的创建方式依赖于<variable identifier>的类别和<expression>的类别：

a） 如果<variable identifier>有一个值类别，那么通过解释*copy*方法来将*Expression*的结果拷贝至当前与*Variable-identifier*相关的值上，*copy*方法通过数据类型定义来定义，它引入<variable identifier>的类别，作为实际参数给出*Variable-identifier*和*Expression*。如果*Expression*为Null，那么引起预定义的异常InvalidReference（见D.3.16）。

b） 如果<variable identifier>有一个对象类别，并且*Expression*的结果是一个对象，那么*Variable-identifier*与对象相关，对象是*Expression*的结果。不允许<expression>的类别为共型，共型限制了<variable identifier>的类别元素。

c） 如果<variable identifier>有一个对象类别，并且*Expression*的结果是一个值，那么通过解释*clone*运算符可以构建一个*Expression*结果的克隆，*clone*运算符通过数据类型定义来定义，它引入<variable identifier>的类别，作为实际参数给出*Expression*。*Variable-identifier*与克隆值的一个引用相关。不允许<expression>的类别为共型，共型限制了<variable identifier>的类别元素。

d） 如果<variable identifier>有一个pid类别，并且*Expression*的结果是一个pid，那么*Variable-identifier*与pid相关，pid是*Expression*的结果。

如果变量用一个共型声明，那么在第12.1.9.5节中定义的范围检查适用于表达式。如果该范围检查返回的预定义布尔值为“假”，那么引起预定义的异常OutOfRange（见D.3.16）。

当解释*Assignment-attempt*时，如果*Expression*的动态类别在类别上兼容于*Variable-identifier*的类别，那么解释涉及*Variable-identifier*和*Expression*的指配。否则，*Variable-identifier*与Null关联。

注 — 通过一次指配尝试，有可能确定一个*Expression*的动态类别。

12.3.3.1 扩展的变量

一个扩展的变量是一种缩写形式的语法符号；不过，除了特殊的语法形式，一个扩展的变量没有特殊的属性，并表示一个操作及其参数。

具体语法

<extended variable> ::=
<indexed variable>
| <field variable>

<indexed variable> ::=
<variable> **(** <actual parameter list> **)**
| <variable> <left square bracket> <actual parameter list> <right square bracket>

<field variable> ::=
<variable> <exclamation mark> <field name>
| <variable> <full stop> <field name>

模型

<indexed variable>是以下的衍生具体句法：

<variable> <is assigned sign> <variable> <full stop> Modify **(** *expressionlist* **)**

其中*expressionlist*通过将<expression>增加至<actual parameter list>来构建。根据第12.2.1节，抽象句法由该具体表达式决定。相同的模型适用于<indexed variable>的第二种形式。

具体句法形式为：

<variable> <exclamation mark> <field name> <is assigned sign> <expression>

是以下的衍生具体句法：

<variable> <full stop> *field-modify-operation-name* **(** <expression> **)**

其中*field-modify-operation-name*由字段名称和“Modify”连接而成。根据第12.2.1节，抽象句法由该具体表达式决定。相同的模型适用于<indexed variable>的第二种形式。

12.3.3.2 缺省初始化

缺省初始化允许对一个特定类别的所有变量以同一数据项进行初始化，在创建变量是进行初始化。

具体语法

<default initialization> ::=

default [<virtuality>] [<constant expression>]

<data type definition>或<syntype definition>必须最多含有一个<default initialization>。

<constant expression>只能在<virtuality>是**redefined** 或 **finalized**时被省略。

语义

缺省初始化可以被加入数据类型定义的<operations>中。缺省初始化规定，任何以类别（由数据类型定义或共型定义引入）声明的变量最初需与<constant expression>的结果相关。

模型

缺省初始化是一种缩写形式，用于为所有的这些变量规定一个显性的初始化，变量声明为<sort>类别，但此处不为<variable definition>提供一个<constant expression>。

如果在<syntype definition>中没有提供<default initialization>，那么共享有<parent sort identifier>的<default initialization>，前提是其结果在范围中。

通过<object data type definition>定义的任何类别都隐性地为其提供一个为Null的<default initialization>，除非一个显性的<default initialization>出现在<object data type definition>中。

任何pid类别都被当做是隐性地为其提供一个为Null的<default initialization>。

如果在重新定义的缺省初始化中省略了<constant expression>，那么不增加显性的初始化。

12.3.4 祈使表达式

祈使表达式从基本的系统状态获得结果。

在本节的*Model*中描述的转换在对入口进行扩充的同时进行。附于动作的标签（其上有一个祈使表达式出现）被移至第一个任务上，该任务在所述的转换期间插入。如果若干个祈使表达式出现在一个表达式上，那么各任务按祈使表达式出现在表达式中的相同次序进行插入。

抽象语法

Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*
	|	*Any-expression*

具体语法

<imperative expression> ::=

<now expression>
| <import expression>
| <pid expression>
| <timer active expression>
| <any expression>
| <state expression>

祈使表达式是访问系统时钟、入口变量结果、代理相关pid、计时器状态的表达式，或者是用于提供非规定数据项的表达式。

12.3.4.1 Now表达式

抽象语法

Now-expression :: ( )

具体语法

<now expression> ::=
now

语义

now表达式是访问系统时钟变量的表达式，以便确定绝对系统时间。

now表达式代表询问系统时钟所提供时间的当前值的表达式。时间的初始值和单位是依赖于系统的。**now**的两个事件在同一转换中是否能给出相同的值是依赖于系统的。不过，它总满足：

now <= **now**;

now表达式有时间类别。

12.3.4.2 入口表达式

具体语法

入口表达式的具体语法在第10.6节中定义。

语义

除了第10.6节中定义的语义，入口表达式被解释为是对入口表达式隐含变量的一次变量访问（见第12.3.2节）。

模型

入口表达式有隐含的语法，用于结果的输入，如第10.6节所定义，并有一个隐含变量的隐含*Variable-access*，用于正文中的入口，在该处出现<import expression>。

在表达式中使用的<import expression>是一种缩写形式，用于在动作之前插入一个任务，在该处表达式出现，表达式用于将<import expression>的结果指配给隐含变量，而后在表达式中使用该隐含变量。如果在一个表达式中出现若干次<import expression>，那么对每个事件使用一个变量。

12.3.4.3 Pid表达式

抽象语法

Pid-expression = *Self-expression*
| *Parent-expression*
| *Offspring-expression*
| *Sender-expression*
Self-expression :: ( )
Parent-expression :: ( )
Offspring-expression :: ( )
Sender-expression :: ( )

具体语法

<pid expression> ::=

self
| **parent**
| **offspring**
| **sender**

<create expression> ::=
create <create body>

<create expression>代表一个*Create-request-node*，对其的进一步讨论见第11.13.2节。

语义

pid表达式访问下列匿名变量之一：self、parent、offspring和sender（见第9节，模型）。**Self**、**parent**、**offspring** 和**sender** pid表达式产生一个结果，它是最后一个与对应的隐性变量相关的pid，如第9条定义。

<pid expression>的动态类别是其结果的动态类别。

Parent、**offspring**或**sender** pid表达式有一个静态类别，它是Pid。

如果<create expression>包括<agent identifier>，那么它有一个静态类别，它是通过<agent identifier>表示的代理的pid类别。如果<create expression>包括<agent type identifier>，那么它有一个静态类别，它是通过<agent type identifier>标识的代理类型的pid类别。如果<create expression>包括**this**，那么它有一个静态类别，它是代理或代理类型的pid类别，创建表达式出现在其中。如果<create expression>包括**this**，并且它出现在不在代理或代理类型内部的正文中（例如在全局过程中），那么它有静态类别Pid。

self表达式是静态类别，它是代理或代理类型的pid，**self**表达式出现在其中。如果它出现在不在代理或代理类型内部的正文中（例如在全局过程中），那么它有静态类别Pid。

模型

在表达式中使用<create expression>是一种缩写形式，用于在动作之前插入创建请求，在此处出现<create expression>，后跟一个对同一类别隐性声明匿名变量的**offspring**指配，类别同<create expression>的静态类别。隐性变量而后用在表达式中。如果在一个表达式中出现若干次<create expression>，那么对每个事件使用一个不同的变量。在这种情况下，插入之创建请求和变量指配的次序与<create expression>的次序相同。

如果<create expression>包含<agent type identifier>，那么包含<agent type identifier>的、应用于创建语句的转换也适用于隐性创建语句，它来自对<create expression>的转换（见第11.13.2节）。

12.3.4.4 计时器主动表达式

抽象语法

Timer-active-expression :: *Timer-identifier*
*Expression**

Timer-active-expression 中*Expression*列表的类别必须按位置对应*Sort-reference-identifier*列表，直接依据由*Timer-identifier*标识的*Timer-name*（见第11.15节）。

具体语法

<timer active expression> ::=
active (<timer identifier> [(<expression list>)])

语义

如果通过计时器标识符标识并以相同结果设置的计时器（通过表达式列表表示（如果有的话））是主动的，那么计时器主动表达式是预定义布尔类别的一个表达式，其结果为“真”（见第11.15节）。否则，计时器主动表达式的结果为“假”。表达式按给定的次序进行解释。

如果在计时器定义中规定的类别是共型，那么用于<expression list>中相应表达式的、在第12.1.9.5节中定义的范围检查必须是“真”的预定义布尔值；否则，引起预定义的异常OutOfRange（见D.3.16）。

12.3.4.5 Any表达式

*Any-expression*对于行为建模来说是非常有用的，在其中，声明一个特定的数据项将意味着过度规范。从通过*Any-expression*返回的结果来看，不可以对其他通过*Any-expression*解释返回的结果做出任何假定。

抽象语法

Any-expression :: *Sort-reference-identifier*

具体语法

<any expression> ::=
any (<sort> **)**

<sort>必须含有元素。

语义

如果类别或共型是类别的一个值，那么*Any-expression*返回一个通过*Sort-reference-identifier*指定的类别或共型的非指定元素。如果*Sort-reference-identifier*表示一个*Syntype-identifier*，那么结果将落在共型范围内。如果通过*Sort-reference-identifier*指定的类别或共型是一个对象类别或pid类别，那么*Any-expression*返回Null。

12.3.4.6 状态表达式

抽象语法

State-expression :: ()

具体语法

<state expression> ::=
state

语义

状态表达式表示字符串文字，它包含最近包含范围单元的最近进入状态的名称拼法。如果没有这样一个状态，那么<state expression>表示空串（“”）。

12.3.5 值返回过程调用

值返回过程调用的抽象句法和静态语义约束见第11.13.3节。

具体语法

<value returning procedure call> ::=
[**call**] <procedure call body>
| [**call**] <remote procedure call body>

如果<value returning procedure call>语法上与某个同名的操作（或变量）（后跟一个参数列表）的关系不明确，那么不得省略关键字**call**。

注1 — 该不明确性不由正文解决。

<value returning procedure call>不可以出现在<continuous signal area>或<enabling condition area>的<Boolean expression>中。

<value returning procedure call>中的<procedure identifier>必须确定一个拥有<procedure result>的过程。

不得省略对应形式**in**/**out** 或 **out**参数的<actual parameters>中的<expression>，它必须是一个<variable identifier>。

在应用**this**模型后，<procedure identifier>必须表示一个含有开始转换的过程。

如果使用**this**，那么<procedure identifier>必须表示一个嵌套的过程。

<procedure call body>代表一个*Value-returning-call-node*，其中*Procedure-identifier*由<procedure identifier>来代表，表达式列表由实际参数列表来代表。<remote procedure call body>代表一个*Value-returning-call-node*，其

中*Procedure-identifier*只含有过程的*Procedure-identifier*，它通过下述模型隐含地定义。*Value-returning-call-node*的语义见第11.13.3节。

模型

如果<procedure identifier>不在嵌套的代理中定义，那么过程调用将被转换为一个本地的、隐含创建的、过程子类型的调用。

this指的是当过程被特殊化时，用特殊化过程的标识符来替换<procedure identifier>。

当<value returning procedure call>含有一个<remote procedure call body>时，隐含地定义一个匿名的过程，其中通过<procedure identifier>表示的过程定义的<procedure result>中的<sort>是该匿名过程的返回类别。该匿名过程有一个单个的<start area>，包含一个带<remote procedure call body>的<return area>，作为其<expression>。

注2 — 本转换不再适用于隐性过程定义。

13 普通的系统定义

系统规范可以拥有可选的部分以及带未指定结果的系统参数，以便满足各种不同的要求。这样一个系统规范被称为普通的系统规范。它的普通属性通过外部的同义词来规定（它类似于过程定义的形式参数）。通过选择其合适的子集来对普通的系统规范进行裁剪，并为每个系统参数提供一个数据项。得到的系统规范不包含外部的同义词，并称为一个特定的系统规范。

普通的系统定义是这样一个系统定义，它包含一个通过<external synonym definition item>定义的同义词（见第12.1.9.6节）、一个通过<external operation definition>定义的操作（见第12.1.8节）、一个通过<external procedure definition>定义的过程（见第9.4节），或在转换选项中的<informal text>（见第13.2节）。通过提供<external synonym definition item>的结果、提供<external operation definition>和<external procedure definition>的行为、将<informal text>转换为形式构件，可以从普通的系统定义来创建特定的系统定义。如何得到它以及它与抽象句法之间的关系，不是本语言定义的一部分。

13.1 可选定义

具体语法

<select definition> ::=

select if (<Boolean simple expression> **)** <end>
{ <agent type reference>
| <agent reference>
| <signal definition>
| <signal list definition>
| <signal reference>
| <remote variable definition>
| <remote procedure definition>
| <data definition>
| <data type reference>
| <interface reference>
| <timer definition>
| <variable definition>
| <procedure definition>
| <procedure reference>
| <select definition>
| <macro definition>
| <exception definition> }+
endselect <end>

<option area> ::=

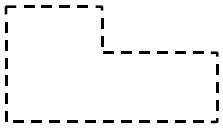
<option symbol> ***contains***
{ **select if (** <Boolean simple expression> **)**
{ <agent type diagram>
| <agent type reference area>
| <agent area>
| <channel definition area>
| <agent text area>
| <procedure text area>
| <composite state type diagram>
| <composite state type reference area>
| <state partition area>
| <procedure area>
| <create line area>
| <option area> } + }

<option symbol> ::=

{ <dashed line symbol> }***set***

<dashed line symbol> ::=

<option symbol>必须形成一个虚线多边形，具有实的角，例如：



<option symbol>逻辑上包含任何通过其边界分割的一维图形符号的全部（即在内部具有一个端点）。

<select definition>的<Boolean simple expression>中的惟一可见的名称是外部同义词的名称，它们在任何<select definition>或<option area>的外面定义，并且数据类型的文字和操作在预定义包内定义，如附件D所定义的那样。

<select definition>可以只包含在该处语法上允许的那些定义。

除了在<agent body area>内，<option area>可以出现在任何地方。<option area>可以只包含在该处语法上允许的那些区域和图形。

语义

如果<Boolean simple expression>的结果是预定义布尔值“假”，那么不得选择在<select definition>或<option symbol>中包含的构件。在其他情况下，可以选择构件。

模型

在转换时删去<select definition>和<option area>，并由包含的所选构件替代（如果有的话）。也移去在非选择<option area>内的、连接至某个区域的任何连接器。

13.2 可选转换串

具体语法

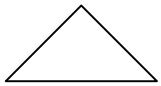
<transition option area> ::=

<transition option symbol> ***contains*** <alternative question>
is followed by <decision body>

<alternative question> ::=

<simple expression>
| <informal text>

<transition option symbol> ::=



<decision body>中的<flow line symbol>连接至<transition option symbol>的底部。

源自<transition option symbol>的<flow line symbol>可以拥有一个公共的起始路径。

<decision body>的<answer>中的每个<constant expression>必须是一个<simple expression>。<transition option area>中的<decision body>中的<answer>必须相互排斥。如果<alternative question>是一个<expression>，那么<decision body>中的<answer>的范围条件必须与<alternative question>具有相同的类别。

<alternative question>中的<informal text>和<character string>与<decision body>中的<answer>之间存在语法的不明确性。如果<alternative question>和所有的<answer>是<character string>，那么所有这些都被解释为<informal text>。如果<alternative question>或任何<answer>是<character string>，并且它与转换选项的正文不匹配，那么<character string>表示<informal text>。

不得省略<transition option area>的<decision body>的<answer part>中的<answer>。

语义

如果<answer>包含有<alternative question>的结果，那么选择<answer part>中的构件。如果没有一个<answer>包含有<alternative question>的结果，那么选择<else part>中的构件。

如果未提供<else part>，并且未选择外出路径，那么选择是无效的。

模型

转化时删除<transition option area>，并由所含的所选构件替代。

附 件 A

非终结符索引

下列非终结符有意未定义和未使用： <macro call>、<page>、<comment area>、<text extension area>、<sdl specification>。

附 件 B

留待将来使用

附 件 C

留待将来使用

附 件 D

SDL预定义数据

D.1 引言

SDL中的预定义数据基于抽象数据类型，它依据其抽象属性进行定义，而不是依据某具体实现。尽管抽象数据类型的定义给出了一种实现该数据类型的方法，只要保持相同的抽象行为，并不强制要求某实现选择该实现抽象数据类型的方法。

预定义数据类型，包括定义属性“真”和“假”的布尔类别，在本附件中定义。“真”和“假”这两个布尔术语不必（直接地或间接地）定义为相当。在数据类型定义之外使用的每个布尔常量表达式都必须解释为“真”或“假”。如果不可能将这样一个表达式简化为“真”或“假”，那么规范是不完全的，允许对数据类型有多个解释。

预定义数据以隐含使用的包Predefined定义（见第7.2节）。该包在本附件中定义。

D.2 符号

出于此目的，本附件通过描述操作的抽象属性来扩展SDL的具体句法，操作是通过数据类型增加的。不过，该额外的语法只用于解释，并不扩展在主文本中定义的语法。因此，使用在本附件中定义之语法的规范不是有效的SDL。

此处描述的抽象属性不规定预定义数据的某个特定表示。相反，解释必须符合这些属性的要求。当解释一个＜expression＞时，对表达式赋值将产生一个值（如作为＜operation application＞的结果=。如果满足以下条件，那么两个表达式E1和E2是等同的：

a） 存在一个＜equation＞ E1 == E2；或

b） 源自给定＜quantified equations＞集的其中一个等式是E1 == E2；或

c） i） E1等同于EA；并且

ii） E2等同于EB；并且

iii） 存在一个等式，或存在一个源自给定集量化等式的等式EA == EB；或

d） 通过用具有相同类的一个项来替代E1的一个子项，作为产生项E1A的子项，有可能表明E1A和E2具有相同的类。

否则两个表达式是不等同的。

两个等同的表达式代表相同的值。

如果两个等同的表达式代表的值相同，那么表达式的解释符合这些属性要求；两个不等同的表达式代表不同的值。

D.2.1 公理

公理确定哪些项代表相同的值。从数据类型定义中的公理，确定变元值和运算符结果值之间的关系，并因此给出运算符的含义。公理或者作为布尔公理给出，或者以代数等式形式给出。

通过＜axiomatic operation definitions＞定义的操作被看做是一个有关特殊化的完整定义。也就是说，当对一个通过包Predefined定义的数据类型进行特殊化并以特殊化类型对操作进行重新定义时，涉及操作名称的所有公理都将被替换为特殊化类型中的相应定义。

具体语法

＜axiomatic operation definitions＞ ::=
axioms ＜axioms＞

＜axioms＞ ::=
＜equation＞ { ＜end＞ ＜equation＞}* [＜end＞]

＜equation＞ ::=
＜unquantified equation＞
| ＜quantified equations＞
| ＜conditional equation＞
| ＜literal equation＞
| ＜noequality＞

＜unquantified equation＞ ::=
＜term＞ == ＜term＞
| ＜Boolean axiom＞

＜term＞ ::=
＜constant expression＞
| ＜error term＞

＜quantified equations＞ ::=
＜quantification＞ (＜axioms＞)

＜quantification＞ ::=
for all ＜value name＞ { , ＜value name＞ =* **in** ＜sort＞

注 — 出于本附件之目的，**for**被认为是一个SDL关键字。

本附件按如下所述修改＜operations＞（见第12.1.1节）。

＜operations＞ ::=
＜operation signatures＞
{ ＜operation definitions＞ | ＜axiomatic operation definitions＞ }

＜axiomatic operation definitions＞只能用于描述运算符的行为。

出现在＜term＞ 中的、为非限定名称的＜identifier＞可以是：

a） 一个＜operation identifier＞（见第12.2.7节）；

b） 一个＜literal identifier＞（见第12.2.2节）；

c） 一个＜value identifier＞，如果在包含＜term＞的＜quantified equations＞的＜quantification＞中存在一个具有该名称的定义，那么它必须具有一个合适的正文类别。

语义

基础项指的是一个不包含任何值标识符的项。一个基础项代表一个特殊的、公认的值。对类别中的每个值，存在至少一个代表该值的基础项。

每个等式是一个有关项的代数等值的语句。左边项和右边项声明为等同，因此，在一个项出现的地方，其他项可以被替换。当一个值标识符出现在一个等式中时，对于值标识符的每个事件，在该等式中它可以同时被相同的项所替换。对该替换，项可以是作为值标识符的、具有相同类别的任何基础项。

值标识符通过量化等式中的值名称来引入。值标识符可用于表示任何属于量化类别的数据值。如果不管所选的、用于替换的值是什么，对等式中值标识符的每个事件都同时替换为相同的值，那么保留等式。

通常，没有必要或没有理由区分基础项和基础项的结果。例如，统一整数元素的基础项可以写为“1”。通常有若干个基础项，它们表示相同的数据项，例如，整数基础项“0＋1”、“3－2”和“（7＋5）/12”，并且通常考虑使用基础项的一种简单形式（在本情况下为“1”）来表示数据项。

值的名称总是通过量化等式来引入，相应的值具有一个值标识符，它是值的名称，通过嵌套的量化等式的类别标识符来限定。例如：

```
for all z in X (for all z in X ... )
```

只引入一个值标识符，命名为类别X的z。

在公理的具体句法中，不允许规定值标识符的限定词。

通过量化等式引入的每个值标识符具有一种类别，它是通过＜sort＞在量化等式中确定的类别。

一个项具有一种类别，它是值标识符的类别，或者是（文字）运算符的结果类别。

除非可以从等式推论出两个项表示相同的值，否则每个项表示一个不同的值。

D.2.2 条件等式

条件等式允许等式规范只有在具备某些限制条件的情况下成立。限制条件以简单等式的形式书写。

具体语法

＜conditional equation＞ ::=
＜restriction＞ { ，＜restriction＞ }* ==> ＜restricted equation＞

＜restricted equation＞ ::=
＜unquantified equation＞

＜restriction＞ ::=
＜unquantified equation＞

语义

只有当限制等式中的任何值标识符表示一个数据项时，条件等式定义表示相同数据项的各项，它可以通过其他等式表示，以便满足限制条件要求。

包括条件等式的、数据类型等式集的语义通过以下步骤得到：

a） 通过产生每个可能的基础项等式（可以自量化等式得到），移去量化。由于这既适用于显性量化，也适用于隐性量化，因此只产生基础项中的一个非量化等式集。

b） 如果能够从非量化等式证明所有的限制条件（只在基础项中）不是限制等式，那么条件等式称为可证明的条件等式。如果存在一个可证明的条件等式，那么它将被可证明的条件等式的限制等式所代替。

c） 如果等式集中剩余有条件等式，并且这些条件等式没有一个是可证明的条件等式，那么删去这些条件等式；否则返回步骤b）。

d） 剩余的非量化等式集定义了数据类型的语义。

D.2.3 等式

具体语法

＜noequality＞ ::=
noequality

模型

引入一个名为S的类别的任何＜data type definition＞ 在其＜operator list＞中具有下列隐含的＜operation signature＞，除非＜noequality＞在＜axioms＞中存在：

equal (S，S) -＞ Boolean;

其中Boolean是预定义的布尔类别。

引入一个名为S的类别以便于其在＜operator list＞中仅包含＜axiomatic operation definitions＞的任何＜data type definition＞ 具有一组隐含的等式：

```
for all a,b,c in S (
  equal(a, a) == true;
  equal(a, b) == equal(b, a);
  equal(a, b) and equal(b, c)   ==> equal(a, c) == true;
  equal(a, b) == true           ==> a == b;)
```

以及一个隐含的＜literal equation＞：

```
for all L1,L2 in S literals (
  spelling(L1) /= spelling(L2) ==> L1 = L2 == false;)
```

D.2.4 布尔公理

具体语法

＜Boolean axiom＞ ::=

＜Boolean term＞

语义

布尔公理是真理的陈述，它对于所定义的数据类型在所有条件下都成立。

模型

形式为：

＜Boolean term＞;

的公理是具体句法等式

＜Boolean term＞ == ＜＜ **package** Predefined/**type** Boolean ＞＞ true;

的派生句法。

D.2.5 条件项

语义

在一个等式中使用的条件项在语义上等价于一组等式，其中所有用Boolean项目表示的量化标识符均已被消去。这一组等式可以这样来构成，即在整个条件项等式中同时用适当类别的各个基本项取代＜conditional expression＞中的＜value identifier＞。在这组等式中，＜conditional expression＞总是由一个Boolean基本项取代。在下文中，这组等式被称为扩展基本集。

一个条件项等式等价于这样的一组等式，它包括：

a) 对于在扩展基本集中＜conditional expression＞等于true的每一个等式，扩展基本集中的那个等式用（基本的＜consequence expression＞来取代其＜conditional expression＞；以及

b) 对于在扩展基本集中＜conditional expression＞等于false的每一个等式，扩展基本集中的那个等式用（基本的＜ alternative expression＞来取代其＜conditional expression＞。

注意，在具有下面形式的等式的特殊情况下：

ex1 == **if** a **then** b **else** c **fi**;

这相当于下面的两个条件等式：

a == true ==> ex1 == b;

a == false ==> ex1 == c;

D.2.6 错误项

使用“错误”的目的是为了能够完全确定一个数据类型的属性，即使在不能为运算符结果赋予特定的含义的情况下。

具体语法

＜error term＞ ::=

raise ＜exception name＞

＜error term＞不得作为 ＜restriction＞的一部分使用。

从一组等式推出 ＜literal identifier＞等于 ＜error term＞是不可能的。

语义

项可以是＜error term＞，这样可以规定一个运算符产生错误的情况。如果在解释期间出现了这些情况，那么会出现具有＜exception name＞的异常。

D.2.7 未排序的文字

具体语法

＜unordered＞ ::=

unordered

该附件修改了有关文字列表类型构造器的具体句法（见第12.1.7.1节），如下所示。

＜literal list＞ ::=

[＜protected＞] **literals** [＜unordered＞]
＜literal signature＞ { , ＜literal signature＞ }* ＜end＞

模型

如果使用了＜unordered＞，那么第12.1.7.1节中的模型就不适用。结果是，不隐含地为该数据类型定义排序运算符“＜”、“＞”、“＜=”、“＞=”、first、last、pred、succ和 num。

D.2.8 文字等式

具体语法

＜literal equation＞ ::=

＜literal quantification＞
(＜equation＞ { ＜end＞ ＜equation＞ }* [＜end＞])

＜literal quantification＞ ::=

for all ＜value name＞ { , ＜value name＞ }* **in** ＜sort＞ **literals**
| **for all** ＜value name＞ { , ＜value name＞ }* **in** { ＜sort＞ | ＜value identifier＞ } **nameclass**

该附件修改了有关＜spelling term＞的具体句法（见第12.1.9.2节），如下所示。

＜spelling term＞ ::=

spelling ({ ＜operation name＞ | ＜value identifier＞ })

＜spelling term＞中的＜value identifier＞ 必须是由＜literal quantification＞定义的＜value identifier＞ 。

语义

文字映射是一种缩写形式，用于定义多个（可能无限多个）公理，遍及一个类别的所有文字或一个名称类中的所有名称。文字映射允许将类别的文字映射至类别的值。

在文字量化中使用＜spelling term＞来引用包含文字拼法的字符串。该机制允许用字符串运算符来定义文字等式。

模型

＜literal equation＞是一组＜axioms＞的缩写形式。在＜literal equation＞ 中包含的每个 ＜equation＞中，由＜literal quantification＞ 中的＜value name＞定义的＜value identifier＞被替代。在每个衍生的＜equation＞中，相同＜value identifier＞的每个事件都由相同＜literal quantification＞的＜sort＞的＜literal identifier＞（如果使用文字的话）或相同的、所指名称类的 ＜literal identifier＞ （如果使用名称类的话）所替代。衍生的＜axioms＞组包含所有可能以这种方式衍生的＜equation＞。

＜literal equation＞的衍生＜axioms＞ 被加入到在关键字**axioms**之后定义的＜axioms＞中（如果有的话）。

如果＜literal quantification＞包含一个或多个＜spelling term＞，那么可用字符串文字来替换＜spelling term＞（见D.3）。字符串用于取代包含＜spelling term＞的＜literal equation＞ 之后的＜value identifier＞，并扩充为如第12.1.9.2节中所定义的那样，用 ＜value identifier＞ 取代＜operation name＞。

注 — 文字等式不影响＜operation signature＞中定义的空运算符。

D.3 预定义包

在下列定义中，考虑将所有在预定义包中定义的名称的参考看做为由限定条件<<**package** Predefined>>加的前缀。为了提高可读性，该限定条件被省略。

```
/* */
package Predefined
/*
```

D.3.1 布尔类别

D.3.1.1 定义

```
*/
value type Boolean;
  literals true, false;
  operators
     "not"  ( this Boolean )                -> this Boolean;
      "and" ( this Boolean, this Boolean )  -> this Boolean;
      "or"  ( this Boolean, this Boolean )  -> this Boolean;
      "xor" ( this Boolean, this Boolean )  -> this Boolean;
      "=>"  ( this Boolean, this Boolean )  -> this Boolean;
axioms
      not( true )      == false;
      not( false )     == true ;
/* */
      true  and true   == true ;
      true  and false  == false;
      false and true   == false;
      false and false  == false;
/* */
      true  or  true   == true ;
      true  or  false  == true ;
      false or  true   == true ;
      false or  false  == false;
/* */
      true  xor true   == false;
      true  xor false  == true ;
      false xor true   == true ;
      false xor false  == false;
/* */
      true  => true    == true ;
      true  => false   == false;
      false => true    == true ;
      false => false   == true ;
endvalue type Boolean;

/*
```

D.3.1.2 应用

布尔类别用来表示“真”和“假”值。它常常用来做为比较的结果。

在SDL中广泛地使用布尔类别。

D.3.2 字符类别

D.3.2.1 定义

```
*/
value type Character;
  literals
     NUL, SOH, STX, ETX, EOT, ENQ,  ACK,  BEL,
     BS,  HT,  LF,  VT,  FF,  CR,   SO,   SI,
     DLE, DC1, DC2, DC3, DC4, NAK,  SYN,  ETB,
     CAN, EM,  SUB, ESC, IS4, IS3,  IS2,  IS1,
     ' ', '!', '"', '#', '$', '%',  '&',  '''',
     '(', ')', '*', '+', ',', '-',  '.',  '/',
     '0', '1', '2', '3', '4', '5',  '6',  '7',
     '8', '9', ':', ';', '<', '=',  '>',  '?',
     '@', 'A', 'B', 'C', 'D', 'E',  'F',  'G',
     'H', 'I', 'J', 'K', 'L', 'M',  'N',  'O',
     'P', 'Q', 'R', 'S', 'T', 'U',  'V',  'W',
     'X', 'Y', 'Z', '[', '\', ']',  '^',  '_',
     '`', 'a', 'b', 'c', 'd', 'e',  'f',  'g',
```

```
     'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
     'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
     'x', 'y', 'z', '{', '|', '}', '~', DEL;
/* “''” 是一个省略符，“ ” 是一个空格，“~”是一个波浪号 */
/* */
  operators
     chr  ( Integer ) -> this Character;
/*  "<"、"<="、">"、">="和“num”是隐含定义的（见第12.1.7.1节）*/
axioms
     for all a,b in Character (
     for all i in Integer (
/* 字符的定义 */
     chr(num(a)) == a;
     chr(i+128)  == chr(i);
     ));
endvalue type Character;

/*
```

D.3.2.2 应用

字符类别用于表示国际参考字母表的字符（ITU-T T.50建议书）。

D.3.3 串类别

D.3.3.1 定义

```
*/
value type String < type Itemsort >;
/* 串从1开始“索引”*/
  operators
     emptystring                                    -> this String;
     mkstring   ( Itemsort                    )  -> this String;
     Make       ( Itemsort                    )  -> this String;
     Length     ( this String                    )  -> Integer;
     First      ( this String                    )  -> Itemsort;
     Last       ( this String                    )  -> Itemsort;
     "//"       ( this String, this String       )  -> this String;
     Extract    ( this String, Integer           )  -> Itemsort raise InvalidIndex;
     Modify     ( this String, Integer, Itemsort )  -> this String;
     Substring  ( this String, Integer, Integer)  -> this String raise InvalidIndex;
     /* substring (s,i,j) 给出了一个长度为j、从第i个元素开始的串 */
     remove     ( this String, Integer, Integer)  -> this String;
     /* remove (s,i,j) 给出了一个子串长度为j、从第i个被移去元素开始的串 */
axioms
     for all e in Itemsort ( /*e - Itemsort的元素 */
     for all s,s1,s2,s3 in String (
     for all i,j in Integer (
/* 构造器为空、mkstring和“//” */
/* 构造器项之间的等同性 */
     s // emptystring  == s;
     emptystring // s  == s;
      (s1 // s2) // s3 == s1 // (s2 // s3);
/* */
/* 通过将之用于所有的构造器来定义“length” */
     <<type String>>length(emptystring) == 0;
     <<type String>>length(mkstring(e)) == 1;
     <<type String>>length(s1 // s2)     == length(s1) + length(s2);
     Make(s)
                                          == mkstring(s);
/* */
/*通过将之用于所有的构造器来定义“Extract”，错误情况单独处理 */
     Extract(mkstring(e),1)                    == e;
     i <= length(s1) ==> Extract(s1 // s2,i) == Extract(s1,i);
     i > length(s1) ==> Extract(s1 // s2,i)   == Extract(s2,i-length(s1));
     i<=0 or i>length(s) ==> Extract(s,i)   == raise InvalidIndex;
/* */
/* 通过其他操作来定义“first”和“last”*/
     first(s)   == Extract(s,1);
     last(s)    == Extract(s,length(s));
/* */
/* 通过j上的归纳，定义“substring(s,i,j)”，错误情况单独处理 */
```

```
     i>0 and i-1<=length(s)              ==>
                         substring(s,i,0)    == emptystring;
/* */
     i>0 and j>0 and i+j-1<=length(s)    ==>
                         substring(s,i,j)    == substring(s,i,j-1) // mkstring(Extract(s,i+j-1));
/* */
     i<=0 or j<0 or i+j-1>length(s)      ==>
                         substring(s,i,j)    == raise InvalidIndex;
/* */
/* 通过其他操作来定义“Modify”。 */
     Modify(s,i,e) == substring(s,1,i-1) // mkstring(e) // substring(s,i+1,length(s)-i);
/* 定义“del”。 */
     remove(s,i,j)   == substring(s,1,i-1) // substring(s,i+j,length(s)-i-j+1);
     )));
endvalue type String;

/*
```

D.3.3.2 用法

Make、Extract和Modify运算符典型地将使用在第12.2.4和12.3.3.1节中定义的缩写形式，来了解串的值以及串的赋值。

D.3.4 字符串类别

D.3.4.1 定义

```
*/
value type Charstring
     inherits String < Character > ( '' = emptystring )
     adding ;
        operators ocs in nameclass
             '''' ( (' ':'&') or '''''' or ('(': '~') )+ '''' -> this Charstring;
/* 从空格“ ”到波浪号“~”的任何字符、任何长度的字符串  */
axioms
     for all c in Character nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                    ==> cs == mkstring(c);
/* 串“A”是字符“A”形成的等 */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                     ==> cs == cs1 // cs2;
     ));
endvalue type Charstring;

/*
```

D.3.4.2 用法

字符串类别定义了用字符组成的串。一个字符串文字可以含有印刷字符和空格。

采用mkstring可以把非印刷字符作为一个串来使用，例如mkstring（DEL）。

例如：

synonym newline_prompt Charstring = mkstring(CR) // mkstring(LF) // '$>';

D.3.5 整数类别

D.3.5.1 定义

```
*/
value type Integer;
  literals unordered nameclass (('0':'9')*) ('0':'9'));
  operators
     "-"   ( this Integer               ) -> this Integer;
     "+"   ( this Integer, this Integer ) -> this Integer;
     "-"   ( this Integer, this Integer ) -> this Integer;
     "*"   ( this Integer, this Integer ) -> this Integer;
     "/"   ( this Integer, this Integer ) -> this Integer raise DivisionByZero;
     "mod" ( this Integer, this Integer ) -> this Integer raise DivisionByZero;
     "rem" ( this Integer, this Integer ) -> this Integer;
     "<"   ( this Integer, this Integer ) -> Boolean;
     ">"   ( this Integer, this Integer ) -> Boolean;
     "<="  ( this Integer, this Integer ) -> Boolean;
     ">="  ( this Integer, this Integer ) -> Boolean;
```

```
     power ( this Integer, this Integer ) -> this Integer;
     bs in nameclass '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )
               -> this Integer;
axioms  noequality
     for all a,b,c in Integer (
/* 构造器为0、1、＋和一元的－ */
/* 构造器项之间的等同性 */
      (a + b) + c              == a + (b + c);
      a + b                    == b + a;
      0 + a                    == a;
      a + (- a)                == 0;
      (- a) + (- b)            == - (a + b);
      <<type Integer>> - 0     == 0;
       - (- a)                 == a;
/* */
/* 通过其他操作来定义二元的“－” */
     a - b                     == a + (- b);
/* */
/* 通过将之应用于所有的构造器来定义“*” */
       0 * a                   == 0;
       1 * a                   == a;
       (- a) * b               == - (a * b);
       (a + b) * c             == a * c + b * c;
/* */
/* 通过将之应用于所有的构造器来定义“<” */
     a < b                                    == 0 < (b - a);
     <<type Integer>> 0 < 0                   == false;
     <<type Integer>> 0 < 1                   == true ;
     0 < a== true ==> 0 < (- a)               == false;
     0 < a and 0 < b== true ==> 0 < (a + b)   == true ;
/* */
/* 通过其他操作来定义“>”、“equal”、“<=”和“>=” */
     a > b                  == b < a;
     equal(a, b)            == not(a < b or a > b=;
     a <= b                 == a < b or a = b;
     a >= b                 == a > b or a = b;
/* */
/* 通过其他操作来定义“/” */
     a / 0                                          == raise DivisionByZero;
     a >= 0 and b > a          == true ==> a / b == 0;
     a >= 0 and b <= a and b > 0  == true ==> a / b == 1 + (a-b) / b;
     a >= 0 and b < 0            == true ==> a / b == - (a / (- b));
     a < 0 and b < 0               == true ==> a / b == (- a) / (- b);
     a < 0 and b > 0                == true ==> a / b== - ((- a) / b);
/* */
/* 通过其他操作来定义“rem” */
     a rem b == a - b * (a/b);
/* */
/* 通过其他操作来定义“mod” */
     a >= 0 and b > 0               ==> a mod b  == a rem b;
     b < 0                            ==> a mod b  == a mod (- b);
     a < 0 and b > 0 and a rem b = 0 ==> a mod b  == 0;
     a < 0 and b > 0 and a rem b < 0 ==> a mod b  == b + a rem b;
     a mod 0 == raise DivisionByZero;
/* */
/* 通过其他操作来定义幂*/
     power(a, 0)              == 1;
     b > 0 ==> power(a, b)    == a * power(a, b-1);
     b < 0 ==> power(a, b)    == power(a, b+1) / a; );
/* */
/* 文字的定义。 */
     <<type Integer>> 2 == 1 + 1;
     <<type Integer>> 3 == 2 + 1;
     <<type Integer>> 4 == 3 + 1;
     <<type Integer>> 5 == 4 + 1;
     <<type Integer>> 6 == 5 + 1;
     <<type Integer>> 7 == 6 + 1;
     <<type Integer>> 8 == 7 + 1;
     <<type Integer>> 9 == 8 + 1;
/* */
/* 0-9之外的文字 */
     for all a,b,c in Integer nameclass (
     spelling(a) == spelling(b) // spelling(c),
     length(spelling(c)) == 1              ==> a == b * (9 + 1) + c;
```

```
    );
/* */
/* 整数的十六进制和二进制表示 */
     for all b in Bitstring nameclass (
     for all i in bs nameclass (
     spelling(i) == spelling(b)          ==> i == <<type Bitstring>>num(b);
                 ));
endvalue type Integer;

/*
```

D.3.5.2 用法

整数类别用于带十进制、十六进制或二进制符号的数学整数。

D.3.6 自然数共型

D.3.6.1 定义

```
*/
syntype Natural = Integer constants >= 0; endsyntype Natural;

/*
```

D.3.6.2 用法

当仅需要正的整数时，才使用自然数共型。所有的运算符都将是整数运算符，但是当一个值用做一个参量或被赋值时，需要对该值进行检验。一个负的值将是一个错误。

D.3.7 实数类别

D.3.7.1 定义

```
*/
value type Real;
  literals unordered nameclass
        ( ('0':'9')* ('0':'9') ) or ( ('0':'9')* '.'('0':'9')+ );
  operators
     "-"   ( this Real            ) -> this Real;
     "+"   ( this Real, this Real ) -> this Real;
     "-"   ( this Real, this Real ) -> this Real;
     "*"   ( this Real, this Real ) -> this Real;
     "/"   ( this Real, this Real ) -> this Real raise DivisionByZero;
     "<"   ( this Real, this Real ) -> Boolean;
     ">"   ( this Real, this Real ) -> Boolean;
     "<="  ( this Real, this Real ) -> Boolean;
     ">="  ( this Real, this Real ) -> Boolean;
     float ( Integer              ) -> this Real;
     fix   ( this Real            ) -> Integer;
axioms noequality
     for all r,s in Real (
     for all a,b,c,d in Integer (
/* 构造器为浮点数和“/” */
/* 构造器之间的等同性总允许形成如下形式:
        float(a) / float(b) 其中 b > 0 */
     r / float(0)                                      == raise DivisionByZero;
     r / float(1)                                      == r;
     c /= 0 ==> float(a) / float(b)                    == float(a*c) / float(b*c);
     b /= 0 and d /= 0 ==>
      (float(a) / float(b)) / (float(c) / float(d))    == float(a*d) / float(b*c);
/* */
/* 通过将之应用于所有的构造器来定义一元的“－” */
     - (float(a) / float(b))                           == float(- a) / float(b);
/* */
/* 通过将之应用于所有的构造器来定义“＋” */
      (float(a) / float(b)) + (float(c) / float(d))    ==float(a*d + c*b) / float(b*d);
/* */
/* 通过其他操作来定义二元的“－” */
     r - s                                             == r + (- s);
/* */
/* 通过将之应用于所有的构造器来定义“*” */
      (float(a) / float(b)) * (float(c) / float(d))    == float(a*c) / float(b*d);
/* */
/* 通过将之应用于所有的构造器来定义“<” */
     b > 0 and d > 0 ==>
```

```
      (float(a) / float(b)) < (float(c) / float(d)) == a * d < c * b;
/* */
/* 通过其他操作来定义“>”、“equal”、“<=”和“>=” */
     r > s      == s < r;
     equal(r, s) == not(r < s or r > s=;
     r <= s     == r < s or r = s;
     r >= s     == r > s or r = s;
/* */
/* 通过将之应用于所有的构造器来定义“fix” */
     a >= b and b > 0 ==> fix(float(a) / float(b)) == fix(float(a-b) / float(b)) + 1;
     b > a  and a >= 0 ==> fix(float(a) / float(b)) == 0;
     a < 0  and b > 0 ==> fix(float(a) / float(b)) == - fix(float(-a)/float(b)) - 1;));
/* */
     for all r,s in Real nameclass (
     for all i,j in Integer nameclass (
        spelling(r) == spelling(i)            ==> r == float(i);
/* */
        spelling(r) == spelling(i)            ==> i == fix(r);
/* */
        spelling(r) == spelling(i) // spelling(s),
        spelling(s) == '.' // spelling(j)     ==> r == float(i) + s;
/* */
        spelling(r) == '.' // spelling(i),
        length(spelling(i)) == 1              ==> r == float(i) / 10;
/* */
        spelling(r) == '.' // spelling(i) // spelling(j),
        length(spelling(i)) == 1,
        spelling(s) == '.' // spelling(j)     ==> r == (float(i) + s) / 10;
     ));
endvalue type Real;

/*
```

D.3.7.2 用法

实数类别用来表示实数。

凡是能够用一个整数除以另一个整数来表示的数都可以用实数类别来表示。

不能够以这种方式表示的数（无理数 — 例如2的平方根）不是实数类别的一部分。不过，对实际工程，通常可以采用足够准确的近似。

D.3.8 数组类别

D.3.8.1 定义

```
*/
value type Array < type Index; type Itemsort >;
  operators
     Make                                  -> this Array ;
     Make    ( Itemsort                  ) -> this Array ;
     Modify  ( this Array,Index,Itemsort ) -> this Array ;
     Extract ( this Array,Index          ) -> Itemsort raise InvalidIndex;
axioms
     for all item, itemi, itemj in Itemsort (
     for all i, j in Index (
     for all a, s in Array (
        <<type Array>>Extract(make,i)                        == raise InvalidIndex;
        <<type Array>>Extract(make(item),i)                  == item ;
        i = j ==> Modify(Modify(s,i,itemi),j,item)           == Modify(s,i,item);
        i = j ==> Extract(Modify(a,i,item),j)                == item ;
        i = j == false ==> Extract(Modify(a,i,item),j)       == Extract(a,j);
        i = j == false ==> Modify(Modify(s,i,itemi),j,itemj) ==
                                                    Modify(Modify(s,j,itemj),i,itemi);
/* 等同性 */
       <<type Array>>Make(itemi) = Make(itemj)                == itemi = itemj;
       a=s == true, i=j == true, itemi = itemj ==>
              Modify(a,i,itemi) = Modify(s,j,itemj)          == true;
/* */
       Extract(a,i) = Extract(s,i) == false ==> a = s        == false;)));
endvalue type Array;

/*
```

D.3.8.2 用法

数组可以用来定义一个类别，它由另一个类别来索引。例如：

```
value type indexbychar inherits Array< Character, Integer >
endvalue type indexbychar;
```

定义了一个含有整数的数组，用字符索引。

数组通常采用缩写形式的Make、Modify和Extract，它们在第12.2.4和12.3.3.1节中定义。例如：

```
dcl charvalue indexbychar;
task charvalue := (. 12 .);
task charvalue('A') := charvalue('B')-1;
```

D.3.9 向量

D.3.9.1 定义

```
*/
value type Vector < type Itemsort; synonym MaxIndex >
     inherits Array< Indexsort, Itemsort >;
syntype Indexsort = Integer constants 1:MaxIndex endsyntype;
endvalue type Vector;
/*
```

D.3.10 Powerset类别

D.3.10.1 定义

```
*/
value type Powerset < type Itemsort >;
  operators
     empty                                      -> this Powerset;
     "in"   ( Itemsort, this Powerset       ) -> Boolean;       /* is member of         */
     incl   ( Itemsort, this Powerset       ) -> this Powerset; /* include item in set  */
     del    ( Itemsort, this Powerset       ) -> this Powerset; /* delete item from set */
     "<"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper subset of  */
     ">"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper superset of*/
     "<="   ( this Powerset, this Powerset  ) -> Boolean;       /* is subset of         */
     ">="   ( this Powerset, this Powerset  ) -> Boolean;       /* is superset of       */
     "and"  ( this Powerset, this Powerset  ) -> this Powerset; /* intersection of sets */
     "or"   ( this Powerset, this Powerset  ) -> this Powerset; /* union of sets        */
     length ( this Powerset                 ) -> Integer;
     take   ( this Powerset                 ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Powerset (
/* 构造器为空和incl */
/* 构造器项之间的等同性 */
     incl(i,incl(j,p))                    == incl(j,incl(i,p));
     i = j ==> incl(i,incl(j,p))          == incl(i,p);
/* 通过将之应用于所有的构造器来定义“in” */
     i in <<type Powerset>>empty          == false;
     i in incl(j,ps)                      == i=j or i in ps;
/* 通过将之应用于所有的构造器来定义“del” */
     <<type Powerset>>del(i,empty)        == empty;
     i =  j ==> del(i,incl(j,ps))       == del(i,ps);
     i /= j ==> del(i,incl(j,ps))       == incl(j,del(i,ps));
/* 通过将之应用于所有的构造器来定义“<” */
     a <  <<type Powerset>>empty           == false;
     <<type Powerset>>empty < incl(i,b) == true;
     incl(i,a) < b                         == i in b and del(i,a) < del(i,b);
/* 通过其他操作来定义“>” */
     a > b == b < a;
/* 通过将之应用于所有的构造器来定义“=” */
     empty = incl(i,ps)                == false;
     incl(i,a) = b    == i in b and del(i,a) = del(i,b);
/* 通过其他操作来定义“<=”和“>=” */
     a <= b                                == a < b or a = b;
     a >= b                                == a > b or a = b;
/* 通过将之应用于所有的构造器来定义“and” */
     empty and b                           == empty;
     i in b   ==> incl(i,a) and b        == incl(i,a and b);
     not(i in b) ==> incl(i,a) and b       == a and b;
/* 通过将之应用于所有的构造器来定义“or” */
```

```
     empty or b                                  == b;
     incl(i,a) or b                              == incl(i,a or b);
/* 定义长度 */
     length(<<type Powerset>>empty=              == 0;
     i in ps ==> length(ps)                      == 1 + length(del(i, ps));
/* 定义“take” */
     take(empty)                                 == raise Empty;
     i in ps ==> take(ps)                        == i;
     ));
endvalue type Powerset;

/*
```

D.3.10.2 用法

Powersets用于表示数学集合。例如：

value type Boolset inherits Powerset< Boolean > **endvalue type** Boolset;

可用于表示一个变量，它可以是空的或包含（“真”）、（“假”）或（“真”，“假”）。

D.3.11 持续时间类别

D.3.11.1 定义

```
*/
value type Duration;
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
     protected duration (Real                                  ) -> this Duration;
     "+"        ( this Duration, this Duration                 ) -> this Duration;
     "-"        ( this Duration                                ) -> this Duration;
     "-"        ( this Duration, this Duration                 ) -> this Duration;
     ">"        ( this Duration, this Duration                 ) -> Boolean;
     "<"        ( this Duration, this Duration                 ) -> Boolean;
     ">="       ( this Duration, this Duration                 ) -> Boolean;
     "<="       ( this Duration, this Duration                 ) -> Boolean;
     "*"        ( this Duration, Real                          ) -> this Duration;
     "*"        ( Real, this Duration                          ) -> this Duration;
     "/"        ( this Duration, Real                          ) -> Duration;
axioms noequality
/* 构造器为duration (Real) */
     for all a, b in Real nameclass (
     for all d, e in Duration nameclass (
/* 通过将之应用于所有的构造器来定义“+” */
     duration(a) + duration(b)       == duration(a + b);
/* */
/* 通过将之应用于所有的构造器来定义一元的“-” */
     - duration(a)                  == duration(-a);
/* */
/* 通过其他操作来定义二元的“-” */
     d - e                           == d + (-e);
/* */
/* 通过将之应用于所有的构造器来定义“equal”、“>”、“<”、“>=”和“<=” */
     equal(duration(a), duration(b)) == a = b;
     duration(a) > duration(b)   == a > b;
     duration(a) < duration(b)   == a < b;
     duration(a) >= duration(b)== a >= b;
     duration(a) <= duration(b)== a <= b;
/* */
/* 通过将之应用于所有的构造器来定义“*” */
     duration(a) * b             == duration(a * b);
     a * d                       == d * a;
/* */
/* 通过将之应用于所有的构造器来定义“/” */
     duration(a) / b           == duration(a / b);
/* */
     spelling(d) == spelling(a) ==>
                           d  == duration(a);
     ));
endvalue type Duration;

/*
```

D.3.11.2 用法

持续时间类别可用于将值加到当前时刻，来设置计时器。持续时间类别的文字与实数类别的文字相同。持续时间一个单元的含义将取决于所定义的系统。

持续时间值可以用实数值来乘或除。

D.3.12 时间类别

D.3.12.1 定义

```
*/
value type Time;
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
     protected time ( Duration    ) -> this Time;
     "<"  ( this Time, this Time ) -> Boolean;
     "<=" ( this Time, this Time ) -> Boolean;
     ">"  ( this Time, this Time ) -> Boolean;
     ">=" ( this Time, this Time ) -> Boolean;
     "+"   ( this Time, Duration) -> this Time;
     "+"   ( Duration, this Time) -> this Time;
     "-"   ( this Time, Duration) -> this Time;
     "-"   ( this Time, this Time ) -> Duration;
axioms noequality
/* 构造器是time */
     for all t, u in Time nameclass (
     for all a, b in Duration nameclass (
/* 通过将之应用于所有的构造器来定义“>”和“equal” */
     time(a) > time(b)          == a > b;
     equal(time(a), time(b))    == a = b;
/* */
/* 通过其他操作来定义“<”、“<=”和“>=” */
     t < u                      == u > t;
     t <= u                     == (t < u= or (t = u);
     t >= u                     == (t > u) or (t = u);
/* */
/* 通过将之应用于所有的构造器来定义“+” */
     time(a) + b                == time(a + b);
     a + t                      == t + a;
/* */
/* 通过其他操作来定义“-”：时间、持续时间 */
     t - b                      == t + (-b);
/* */
/* 通过将之应用于所有的构造器来定义“-”：时间、时间 */
     time(a) - time(b)          == a - b;
/* */
     spelling(a) == spelling(t) ==>
                           a == time(t);
     ));
endvalue type Time;

/*
```

D.3.12.2 用法

now表达式返回一个具有时间类别的值。可以将一个持续时间加到一个时间值上或从一个时间值减去，以得到另一个时间值。从一个时间值减去另一个时间值可以得到一个持续时间。时间值用于设置计时器的终止时间。

时间的起点取决于系统。一个时间单元就是时间量，通过将一个持续时间单元加到一个时间上来表示。

D.3.13 Bag类别

D.3.13.1 定义

```
*/
value type Bag < type Itemsort >;
  operators
     empty                        -> this Bag;
     "in"   ( Itemsort, this Bag  ) -> Boolean; /* is member of */
     incl   ( Itemsort, this Bag  ) -> this Bag;  /* include item in set    */
     del    ( Itemsort, this Bag  ) -> this Bag;  /* delete item from set   */
     "<"    ( this Bag, this Bag  ) -> Boolean; /* is proper subbag of     */
     ">"    ( this Bag, this Bag  ) -> Boolean; /* is proper superbag of   */
```

```
     "<="  ( this Bag, this Bag  ) -> Boolean; /* is subbag of  */
     ">="  ( this Bag, this Bag  ) -> Boolean; /* is superbag of         */
     "and" ( this Bag, this Bag  ) -> this Bag; /* intersection of bags  */
     "or"  ( this Bag, this Bag  ) -> this Bag; /* union of bags         */
     length ( this Bag           ) -> Integer;
     take   ( this Bag           ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Bag (
/* 构造器为空和incl */
/* 构造器项之间的等同性 */
     incl(i,incl(j,p))              == incl(j,incl(i,p));
/* 通过将之应用于所有的构造器来定义“in” */
     i in <<type Bag>>empty         == false;
     i in incl(j,ps)                == i=j or i in ps;
/* 通过将之应用于所有的构造器来定义“del” */
     <<type Bag>>del(i,empty)       == empty;
     i =  j ==> del(i,incl(j,ps))   == ps;
     i /= j ==> del(i,incl(j,ps))   == incl(j,del(i,ps));
/* 通过将之应用于所有的构造器来定义“<” */
     a <  <<type Bag>>empty         == false;
     <<type Bag>>empty < incl(i,b)  == true;
     incl(i,a) < b                  == i in b and del(i,a) < del(i,b);
/* 通过其他操作来定义“>” */
     a > b == b < a;
/* 通过将之应用于所有的构造器来定义“=” */
     empty = incl(i,ps)             == false;
     incl(i,a) = b                  == i in b and del(i,a) = del(i,b);
/* 通过其他操作来定义“<=”和“>=” */
     a <= b                         == a < b or a = b;
     a >= b                         == a > b or a = b;
/* 通过将之应用于所有的构造器来定义“and” */
     empty and b                    == empty;
     i in b       ==> incl(i,a) and b  == incl(i,a and b);
     not(i in b)  ==> incl(i,a) and b  == a and b;
/* 通过将之应用于所有的构造器来定义“or” */
     empty or b                     == b;
     incl(i,a) or b                 == incl(i,a or b);
/* 定义长度 */
     length(<<type Bag>>empty=      == 0;
     i in ps     ==> length(ps)     == 1 + length(del(i, ps));
/* 定义take */
     take(empty)                    == raise Empty;
     i in ps     ==> take(ps)       == i; ));
endvalue type Bag;
/*
```

D.3.13.2 用法

Bag用于表示多集合。例如：

value type Boolset **inherits** Bag< Boolean > **endvalue type** Boolset;

可用于表示一个变量，它可以是空的或包含（“真”）、（“假”）、（“真”，“假”）、（“真”，“真”）、（“假”，“假”），……。

Bag用于表示ASN.1构造的集合。

D.3.14 ASN.1比特和比特串类别

D.3.14.1 定义

```
*/
value type Bit
     inherits Boolean ( 0 = false, 1 = true );
     adding;
  operators
     num ( this Bit ) -> Integer;
     bit ( Integer) -> this Bit raise OutOfRange;
axioms
     <<type Bit>>num (0)          == 0;
     <<type Bit>>num (1)          == 1;
     <<type Bit>>bit (0)          == 0;
     <<type Bit>>bit (1)          == 1;
     for all i in Integer (
       i > 1 or i < 0 ==> bit (i) == raise OutOfRange;
```

```
      )
endvalue type Bit;
/* */
value type Bitstring
  operators
     bs in nameclass
     '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )-> this Bitstring;
/* 除了比特串从0开始索引之外，以下运算符等同于串的运算符 */
     mkstring     (Bit                                ) -> this Bitstring;
     Make         (Bit                                ) -> this Bitstring;
     Length       ( this Bitstring                    ) -> Integer;
     First        ( this Bitstring                    ) -> Bit;
     Last         ( this Bitstring                    ) -> Bit;
     "//"         ( this Bitstring, this Bitstring    ) -> this Bitstring;
     Extract      ( this Bitstring, Integer           ) -> Bit raise InvalidIndex;
     Modify       ( this Bitstring, Integer, Bit      ) -> this Bitstring;
     substring    ( this Bitstring, Integer, Integer ) -> this Bitstring raise InvalidIndex;
     /* substring (s,i,j)给出一个长度为j、从第i个元素开始的串 */
     remove       ( this Bitstring, Integer, Integer  ) -> this Bitstring;
     /* remove (s,i,j)给出一个带长度为j、从第i个元素开始移去的子串的串 */
/*The following operators are specific to Bitstrings*/
     "not"  ( this Bitstring                 ) -> this Bitstring;
     "and"  ( this Bitstring, this Bitstring ) -> this Bitstring;
     "or"   ( this Bitstring, this Bitstring ) -> this Bitstring;
     "xor"  ( this Bitstring, this Bitstring ) -> this Bitstring;
     "=>"   ( this Bitstring, this Bitstring ) -> this Bitstring
     num    ( this Bitstring                 ) -> Integer;
     bitstring ( Integer                     ) -> this Bitstring raise OutOfRange;
     octet  ( Integer                        ) -> this Bitstring raise OutOfRange;
axioms
/* 比特串从索引0开始 */
/* 运算符定义，名称等同于串运算符 */
     for all b in Bit ( /*b is bit in string*/
     for all s,s1,s2,s3 in Bitstring (
     for all i,j in Integer (
/* 构造器为''B、mkstring和“//” */
/* 构造器项之间的等同性 */
     s // ''B             == s;
     ''B// s              == s;
      (s1 // s2) // s3 == s1 // (s2 // s3);
/* 通过将之应用于所有的构造器来定义“length” */
     <<type Bitstring>>length(''B)               == 0;
     <<type Bitstring >>length(mkstring(b))      == 1;
     <<type Bitstring >>length(s1 // s2)         == length(s1) + length(s2);
     Make(s)                                     == mkstring(s);
/* 通过将之应用于所有的构造器来定义“Extract”，错误情况单独处理 */
     Extract(mkstring(b),0)                      == b;
     i < length(s1)       ==> Extract(s1 // s2,i) == Extract(s1,i);
     i >= length(s1)      ==> Extract(s1 // s2,i) == Extract(s2,i-length(s1));
     i<0 or i=>length(s) ==> Extract(s,i)        == raise InvalidIndex;
/* 通过其他操作来定义“first”和“last” */
     first(s)   == Extract(s,0);
     last(s)    == Extract(s,length(s)-1);
/* 通过j上的归纳来定义substring(s,i,j)，错误情况单独处理 */
     i>=0 and i < length(s)                 ==>
          substring(s,i,0)  == ''B;
/* */
     i>=0 and j>0 and i+j<=length(s) ==>
          substring(s,i,j)  == substring(s,i,j-1) // mkstring(Extract(s,i+j));
/* */
     i<0 or j<0 or i+j>length(s)      ==>
         substring(s,i,j) == raise InvalidIndex;
/* */
/* 通过其他操作来定义“Modify” */
     Modify(s,i,b) == substring(s,0,i) // mkstring(b) // substring(s,i+1,length(s)-i);
/* 定义“remove” */
     remove(s,i,j)   == substring(s,0,i) // substring(s,i+j,length(s)-i-j);
     )));
/* 终止定义从0开始索引的串运算符 */
/* */
```

```
/* 依据''B, 'xxxx'B为比特串定义''H和'x'H */
     <<type Bitstring>>''H  == ''B;
     <<type Bitstring>>'0'H == '0000'B;
     <<type Bitstring>>'1'H == '0001'B;
     <<type Bitstring>>'2'H == '0010'B;
     <<type Bitstring>>'3'H == '0011'B;
     <<type Bitstring>>'4'H == '0100'B;
     <<type Bitstring>>'5'H == '0101'B;
     <<type Bitstring>>'6'H == '0110'B;
     <<type Bitstring>>'7'H == '0111'B;
     <<type Bitstring>>'8'H == '1000'B;
     <<type Bitstring>>'9'H == '1001'B;
     <<type Bitstring>>'A'H == '1010'B;
     <<type Bitstring>>'B'H == '1011'B;
     <<type Bitstring>>'C'H == '1100'B;
     <<type Bitstring>>'D'H == '1101'B;
     <<type Bitstring>>'E'H == '1110'B;
     <<type Bitstring>>'F'H == '1111'B;
/* */
/* 定义比特串特定的运算符 */
     <<type Bitstring>>mkstring(0)      == '0'B;
     <<type Bitstring>>mkstring(1)      == '1'B;
/* */
     for all s, s1, s2, s3 in Bitstring (
     s  =  s                                  == true;
     s1 = s2                                  == s2 = s1;
     s1 /= s2                                 == not ( s1 = s2 );
     s1 = s2 == true            ==> s1        == s2;
     ((s1 = s2) and (s2 = s3))  ==> s1 = s3 == true;
     ((s1 = s2) and (s2 /= s3)) ==> s1 = s3 == false;
/* */
     for all b, b1, b2 in Bit (
     not(''B)                   == ''B;
     not(mkstring(b) // s)      == mkstring( not(b) ) // not(s);
/* 定义“or”。 */
/* 两个串“or”运算得到的串的长度为两个串中长度较长的串的长度 */
     ''B or ''B                 == ''B;
     length(s) > 0 ==> ''B or s == mkstring(0) or s;
     s1 or s2                   == s2 or s1;
     (b1 or b2) // (s1 or s2)   ==(mkstring(b1) // s1) or (mkstring(b2) // s2);
/* */
/* 基于“or”和“not”定义剩余运算符 */
     s1 and s2                  == not (not s1 or not s2);
     s1 xor s2                  == (s1 or s2) and not(s1 and s2);
     s1 => s2                   == not (s1 and s2);
     ));
/* */
/* 定义'xxxxx'B文字 */
     for all s in Bitstring (
     for all b in Bit (
     for all i in Integer (
     <<type Bitstring>>num (''B)          == 0;
     <<type Bitstring>>bitstring (0)    == '0'B;
     <<type Bitstring>>bitstring (1)    == '1'B;
     num (s // mkstring (b))            == num (b) + 2 * num (s);
     i > 1                ==> bitstring (i) == bitstring (i / 2) // bitstring (i mod 2);
     i >= 0 and i <= 255  ==> octet (i) == bitstring (i) or '00000000'B;
     i < 0                ==> bitstring (i) == raise OutOfRange;
     i < 0 or i > 255  ==> octet (i)    == raise OutOfRange;
     )))
/* 定义'xxxxx'H文字 */
     for all b1,b2,b3,h1,h2,h3 in bs nameclass (
     for all bs1, bs2, bs3, hs1, hs2, hs3 in Charstring (
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     bs1 /= bs2                              ==> b1 = b2   == false;
/* */
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     hs1 /= hs2 ==> h1 = h2 == false;
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     spelling(b3) = '''' // bs1 // bs2 // '''B',
```

```
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     spelling(h3) = '''' // hs1 // hs2 // '''H',
     length(bs1) = 4,
     length(hs1) = 1,
     length(hs2) > 0,
     length(bs2) = 4 * length(hs2),
     h1 = b1                                  ==> h3 = b3      == h2 = b2;
/* */
/* 连接至串产生器 */
     for all b in Bit literals (
     spelling(b1) = '''' // bs1 // bs2 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     spelling(b)  = bs1                       ==> b1           == mkstring(b) // b2;
     )));
endvalue type Bitstring;
/*
```

D.3.15 ASN.1 八位字节和八位字节串类别

D.3.15.1 定义

```
*/
syntype Octet = Bitstring size (8);
endsyntype Octet;
/* */
value type Octetstring
     inherits String < Octet > ( ''B = emptystring )
     adding
     operators
     os in nameclass
          '''' ( ((('0' or '1')8)*'''B') or ((((0':'9') or ('A':'F'))2)*'''H') )
          -> this Octetstring;
     bitstring  ( this Octetstring) -> Bitstring;
     octetstring( Bitstring) -> this Octetstring;
axioms
     for all b,b1,b2 in Bitstring (
     for all s in Octetstring (
     for all o in Octet(
     <<type Octetstring>> bitstring(''B)          == ''B;
     <<type Octetstring>> octetstring(''B)        == ''B;
     bitstring( mkstring(o) // s )                == o // bitstring(s);
/* */
     length(b1) > 0,
     length(b1) < 8,
     b2 == b1 or '00000000'B ==> octetstring(b1)  == mkstring(b2);
/* */
     b == b1 // b2,
     length(b1) == 8           => octetstring(b) == mkstring(b1) // octetstring(b2);
     )));
/* */
     for all b1, b2 in Bitstring (
     for all o1, o2 in os nameclass (
     spelling( o1 ) = spelling( b1 ),
     spelling( o2 ) = spelling( b2 ) ==> o1 = o2 == b1 = b2
     ));
endvalue type Octetstring;
/*
```

D.3.16 预定义的异常

```
*/
exception
     OutOfRange,          /* 范围检查失败。 */
     InvalidReference,    /* 不正确地使用了Null。本信号的pid错。 */
     NoMatchingAnswer,    /* 没有回答能够匹配不带else部分的决策。 */
     UndefinedVariable, /* 使用了“未定义的”变量。 */
     UndefinedField,      /* 访问选择或结构中未定义的字段。 */
     InvalidIndex,      /* 访问串或数组时索引错。 */
     DivisionByZero;      /* 试图对整数或实数除零。 */
     Empty;               /* 没有元素可返回。 */
/* */
endpackage Predefined;
```

附 件 E

留待例子使用

附 件 F

SDL正式定义

单独发布。

附 录 I

Z.100的状态、相关文档和建议书

本附录包含一份有关ITU-T发布的SDL相关文档状态的列表。列表包括ITU-T Z.100、Z.105、Z.106、Z.107、Z.109建议书以及任何方法学相关文档的所有部分。它还列出了其他相关的文档，如ITU-T Z.100建议书。

一旦同意对SDL进行修改并批准了新的文档，那么就应通过适当的方式对该列表进行更新（例如一个勘误表）。

SDL-2000由以下建议书定义，它们由ITU-T第10研究组于1999年11月19日批准。

— ITU-T Z.100建议书（2002），规范和描述语言（*SDL*）。

— ITU-T Z.100建议书附件A，非终结符索引。

— ITU-T Z.100建议书附件D，*SDL*预定义数据。

— ITU-T Z.100建议书附件F（2000），*SDL*正式定义（由ITU-T第10研究组于2000年11月24日批准）。

在批准之时对附件B、C和E没有特殊的计划。

— ITU-T Z.100建议书增补1（1997），*SDL*＋方法学：*MSC*和*SDL*的使用（带ASN.1）。

— ITU-T Z.105建议书（2001），结合*ASN.1*模块的*SDL*（SDL/ASN.1）。

— ITU-T Z.106建议书（2002），*SDL*公用交换格式。

— ITU-T Z.106建议书（199），带嵌入式*ASN.1*的*SDL*。

— ITU-T Z.109建议书（1999），结合*UML*的*SDL*。

— ITU-T Z.110建议书（2000），ITU-T使用形式描述技术的准则（由ITU-T第10研究组于2000年11月24日批准）。

有关SDL的更多信息，包括有关书和其他出版物的信息，请查阅以下网址：http://www.sdl-forum.org。

附 录 II

SDL维护指南

II.1 SDL维护

本附录描述有关SDL维护的术语和规则（于1993年11月由第10研究组会议批准），以及相关的“修改请求程序”。

II.1.1 术语

a） *错误*指的是与ITU-T Z.100建议书的内部不一致性。

b） *文本修正*指的是修改ITU-T Z.100建议书的文本或图，旨在修正文字或印刷错误。

c） *开放项*指的是一个确定的但未解决的问题。可以通过修改请求或研究组或工作方的协议来确定一个开放项。

d） *不足*指的是一个确定的问题，ITU-T Z.100建议书对其SDL语义没有（明确地）定义。

e） *阐明*指的是修改ITU-T Z.100建议书的文本或图，它指出，如果不阐明的话，以往的文本或图可能引起歧义。阐明应力图使ITU-T Z.100建议书对应SDL的语义，如研究组或工作方所理解的那样。

f） *修正*指的是修改ITU-T Z.100建议书的文本或图，旨在修改SDL的语义。

g） *解除的特性*指的是将在ITU-T Z.100建议书的下一次修订中从SDL中删去的特性。

h） *扩展*指的是一个新的特性，它不得修改ITU-T Z.100建议书中所定义特性的语义。

II.1.2 维护规则

在下文中，当提到ITU-T Z.100建议书时，应认为包括各附件、附录和增补，以及补遗、修正、勘误表或实施者指南，对ITU-T Z.105建议书、ITU-T Z.106建议书、ITU-T Z.107建议书和ITU-T Z.109建议书的文本也一样。

a） 当在ITU-T Z.100建议书中检测到一个错误或不足时，必须予以纠正或阐明。修正一个错误的影响应尽可能小。错误修正和阐明应列入ITU-T Z.100建议书的修改主列表，并应立即生效。

b） 除从上一研究周期开始修正和解决开发项错误之外，修改和扩展只能被认为时修改请求的结果，它由一个真实的用户团体支持。提出修改请求后，研究组或工作方应在用户组代表的协助下展开调查，以便明确需求和效益，确定现有的SDL特性是不合适的。

c） 不是源自错误修正的修改和扩展应广泛发布，在采纳修改之前，应对用户和工具制造者的意见进行彻底检查。除非特殊情况要求尽快实施此类修改，否则在修订ITU-T Z.100建议书之前不建议进行此类修改。

d） 在经修订的ITU-T Z.100建议书发布之前，将维护ITU-T Z.100建议书的修改主列表，包括ITU-T Z.100建议书和所有的附件，除了正式的定义。一旦研究组做出决定，将发布附录、增补、勘误表、实施者指南或附件。为确保ITU-T Z.100建议书修改主列表的有效发布，它应通过适当的电子方式作为COM报告出版。

e） 对ITU-T Z.100建议书中的不足，应商议确定一个正式的定义。这将阐明或修正ITU-T Z.100建议书修改主列表中记录的不足。

II.1.3 修改请求程序

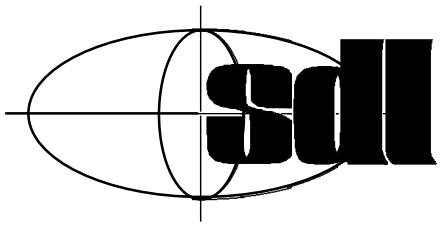
设计修改请求程序旨在使ITU-T的内部和外部SDL用户能够提出有关ITU-T Z.100建议书精确含义的问题，为SDL或ITU-T Z.100建议书的修改提出建议，并提供有关SDL提议之修改意见的反馈信息。在实施之前，SDL专家小组应发布有关SDL的提议之修改。

修改请求应使用修改请求表（见以下），或提供表格列出的信息。应明确指明请求的种类（错误修正、阐明、简化、扩展、修改或解除的特性）。还有一点也很重要，对错误修正之外的任何修改，应指明支持修改请求的用户数量。

负责ITU-T Z.100建议书的ITU-T研究组应开会对所有的修改请求进行讨论。对修正或阐明，可以无需与用户商议而将修改放入修正列表中。否则，需汇编一个开放项列表。发布给用户的信息应：

- 作为ITU-T白皮分发报告；
- 通过电子邮件发给SDL邮件列表（如ITU-T非正式列表和sdlnews@sdl-forum.org）；
- 第10研究组专家同意的其他手段。

研究组专家应确定支持的水平，以及对每个修改的反对意见，并对用户的反应做出评估。如果有坚实的用户支持，并且对由较多用户提议的修改没有强烈的反对意见，那么修改将只放入修改接受列表中。最后，所有接受的修改将纳入修订的ITU-T Z.100建议书中。用户应认识到，在修改被纳入并被负责ITU-T Z.100建议书的研究组批准之前，ITU-T不推荐使用它们。



修改请求表

请提供以下详细信息。		
修改类型：	❑ 错误修正	❑ 阐明
	❑ 简化	❑ 扩展
	❑ 修改	❑ 解除
修改请求的简短叙述		
修改请求的简短理由		
在你的组织中是否都有该观点？	❑ 是	❑ 否
你是否已咨询其他用户?	❑ 是	❑ 否
你代表多少用户？	❑ 1-5	❑ 6-10
	❑ 11-100	❑ ＞ 100
你的名字和地址		

需要的话，请附另外页说明详细内容。

SDL（Z.100）大会报告起草人，c/o ITU-T，Place des Nations，CH-1211 Geneva 20，瑞士。
传真：+ 41 22 730 5853，电子邮件：SDL.rapporteur@itu.int。

附 录 III

SDL-92至SDL-2000的系统转化

虽然不是所有的SDL-92规范都能自动地转化为SDL-2000，但在许多情况下，转化应足够简单。

1. 修正有关大小写和新关键字的拼法：

   a） 以相应的小写＜keyword＞（或大写＜keyword＞）替换所有的关键字；

   b） 以惟一的＜word＞替换所有包含国家字符的＜word＞，如Z.100（03/93）中定义；

   c） 以其相当的小写替换所有的＜name＞；

   d） 如果＜name＞与小写＜keyword＞存在冲突，那么用大写字符替换首字符。

   在许多情况下，一个更简单的过程是可能的，例如，总使用定义其对应＜identifier＞的＜name＞拼法。这使得只有在状态名称改变的情况下，才改变语义，并且该名称用在＜state expression＞中，如SDL-92增补1中所介绍的那样。

2. 用SDL-2000相应的＜qualifier＞替换所有的＜qualifier＞（也就是说，路径项列表总是嵌套在＜composite special＞、＜qualifier begin sign＞和＜qualifier end sign＞中。

3. 将＜agent formal parameters＞、＜procedure formal parameters＞、＜procedure result＞、＜macro formal parameter＞、＜formal operation parameters＞和＜procedure signature＞中关键字fpar的所有用法和返回值转化为相应的SDL-2000语法。

4. 用节点位置＜channel definition area＞替换所有的信号路由。

5. 在每个没有信号路由或信道的功能块中，为每个进程增加通道，列出由该进程发送或接收的所有信号。作为选择，可以依据有关SDL-92隐含信号路由的模型，增加隐含信道，如增补1所介绍的那样，也必须增加相应功能块的通道。

6. 替换所有的功能块划分事件。SDL-92并不规定如何选择一个一致的子集，因此该步骤可能需要外部的知识。假定总需要选择子结构的转变，可能反映SDL-92的典型用法。

   a） 子结构中的所有功能块直接移至存储器功能块中。

   b） 如果功能块实体与子结构实体中存在冲突，那么重新为实体赋予一个惟一的名称。

   c） 调整嵌套功能块中实体的所有标识符，以便使用新的限定词。

7. 通过一个输出动作列表，利用via all替换所有的输出动作。如果＜via path＞是一个功能块实例集之间的信道，那么不可能是自动的转化。

8. 以复合状态和复合状态类型分别替换业务和业务类型。如果业务有重叠的输入动作（尽管其有效的输入集是脱节的），必须移去一个复制的转变。移去业务之间的所有信号路由；引用这些信号路由的输出应引用进程类型的通道。用＜return area＞替换＜stop symbol＞。业务的计时器、输出过程和输出变量必须在代理中定义。

9. 用等同的、使用参数化类型的定义，替换所有的、涉及产生器转化的数据类型定义。

10. 数据公理的转换不可能自动完成，但只有少数用户在公理上定义了他们自己的数据类型。不过，以下情况可以方便地实现转化：

    a） 预定义数据表达式仍然有效，包括使用String、Array和PowerSet，在调整其类型拼法的大小写之后。

    b） 带文字的新类型定义（而不是公理上定义的运算符）可以被转换为带＜literal list＞的＜value data type definition＞。

    c） 带结构属性的新类型定义可以被转换为带＜structure definition＞的＜value data type definition＞。

如果某个SDL-92规范使用了不同自动转化为SDL-2000中等同构件的构件，那么必须对该构件进行仔细检查，如果它需要与本建议书保持一致的话。

ITU-T 系列建议书

A系列	ITU-T工作的组织
B系列	表述方式：定义、符号、分类
C系列	综合电信统计
D系列	一般资费原则
E系列	综合网络运行、电话业务、业务运行和人为因素
F系列	非话电信业务
G系列	传输系统和媒质、数字系统和网络
H系列	视听和多媒体系统
I系列	综合业务数字网
J系列	电视、声音节目和其他多媒体信号的传输
K系列	干扰的防护
L系列	线缆的构成、安装和保护及外部设备的其他组件
M系列	TMN和网络维护：国际传输系统、电话电路、电报、传真和租用电路
N系列	维护：国际声音节目和电视传输电路
O系列	测量设备技术规程
P系列	电话传输质量、电话装置及本地线路网络
Q系列	交换和信令
R系列	电报传输
S系列	电报业务终端设备
T系列	远程信息处理业务的终端设备
U系列	电报交换
V系列	电话网上的数据通信
X系列	数据网和开放系统通信
Y系列	全球信息基础设施和网际协议问题
Z系列	**用于电信系统的语言与一般软件问题**

瑞士印刷
2003年，日内瓦