This electronic version (PDF) was scanned by the International Telecommunication Union (ITU) Library & Archives Service from an original paper document in the ITU Library & Archives collections.

La présente version électronique (PDF) a été numérisée par le Service de la bibliothèque et des archives de l'Union internationale des télécommunications (UIT) à partir d'un document papier original des collections de ce service.

Esta versión electrónica (PDF) ha sido escaneada por el Servicio de Biblioteca y Archivos de la Unión Internacional de Telecomunicaciones (UIT) a partir de un documento impreso original de las colecciones del Servicio de Biblioteca y Archivos de la UIT.

هذه النسخة الإلكترونية (PDF) للاتصالات الدولي الاتحاد في والمحفوظات المكتبة قسم أجراه الضوئي بالمسح تصوير نتاج (ITU) والمحفوظات المكتبة قسم في المتوفرة الوثائق ضمن أصلية ورقية وثيقة من نقلاً.

此电子版（PDF版本）由国际电信联盟（ITU）图书馆和档案室利用存于该处的纸质文件扫描提供。

Настоящий электронный вариант (PDF) был подготовлен в библиотечно-архивной службе Международного союза электросвязи путем сканирования исходного документа в бумажной форме из библиотечно-архивной службы МСЭ.

© International Telecommunication Union

国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

蓝 皮 书

卷 X.3

建议Z.100的附件F.1：SDL形式定义 介绍

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

蓝 皮 书

卷 X.3

建议Z.100的附件F.1：SDL形式定义 介绍

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

ISBN 92-61-03775-5

ARCHIVES U.I.T. GENÈVE

© ITU

中国印刷

CCITT 图书目录
第九次全体会议(1988 年)

蓝 皮 书

卷 I

卷 II

卷 III

卷Ⅲ.2 — 国际模拟载波系统 建议G.211-G.544(第XV研究组)

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

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

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

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

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

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

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

卷Ⅳ

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

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

卷Ⅳ.3 — 国际声音节目和电视传输电路的维护 N系列建议(第Ⅳ研究组)

卷Ⅳ.4 — 测量设备技术规程 O系列建议(第Ⅳ研究组)

卷Ⅴ — 电话传输质量 P系列建议(第XⅡ研究组)

卷Ⅵ

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

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

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

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

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

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

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

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

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

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

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

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

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

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

卷Ⅶ

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

卷Ⅶ.2 — 电报交换 U系列建议(第Ⅸ研究组)

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

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

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

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

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

卷Ⅷ

卷Ⅷ.1 — 电话网上的数据通信 V系列建议(第XⅦ研究组)

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

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

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

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

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

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

卷Ⅷ.8 — 数据通信网:查号 建议X.500-X.521(第Ⅶ研究组)

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

卷X

卷X.1 — 功能规格和描述语言(SDL) 使用形式描述方法(FDT)的标准 建议Z.100和附件A、B、C和E,建议Z.110(第X研究组)

卷X.2 — 建议Z.100的附件D:SDL用户指南(第X研究组)

卷X.3 — 建议Z.100的附件F.1:SDL形式定义 介绍(第X研究组)

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

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

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

卷X.7 — 人机语言(MML) 建议Z.301-Z.341(第X研究组)

蓝皮书卷 X.3 目录

建议 Z.100 的附件 F.1

卷 首 说 明

1 在 1985-1989 研究期内委托给各研究组的研究课题可查阅该研究组的第一号文稿。

2 本卷中的“主管部门”一词是电信主管部门和经认可的私营机构两者的简称。

目　录

卷 X.3

建议 Z.100 附件 F.1

SDL 形式定义

1 前言

SDL 的形式定义提供了一种语言定义，它对建议正文中给出的定义作了补充。本附件的使用对象是那些需要非常精确和详细的 SDL 定义的人，例如，SDL 语言的维护者和 SDL 工具的设计者。

形式定义由以下三卷组成

附件 F.1 （本卷）
陈述动机，描述整个结构，指导如何使用形式定义和描述所用的一套符号。

附件 F.2 给 SDL 的静态特性下定义

附件 F.3 给 SDL 的动态特性下定义

2 动机

自然语言通常带有**二义性**和**不完整性**，即，对语言中的某些句子可以给出不止一种解释，不论读者是计算机或者是人都是如此。

如果一种定义或规格说明的意义（语义）是非二义性的和完整的，则它是**形式**的。由于自然语言不能用于该目的，已经开发了专用的语言，叫做**规格说明语言**（象 SDL 和 LOTOS 等语言）。类似 CHILL 或 PASCAL 这类面向实现的语言也可以用作规格说明语言（例如，一个编译程序形式地规定了另一种语言的语义），然而最重要的常常是要把与理解无关的实现细节与规格说明的语义分开。

特别适合于规定语言的形式语言称为**元语言**。例如，巴科斯—诺尔范式（BNF）是一种元语言，它特别适合于形式地规定编程语言的语法。

不管自然语言的二义性如何，自然语言对人类来说较之形式语言通常是更易读的，并且更容易表达基本的理论基础的说明，提出说明问题的框架结构，在这个框架结构中形式规格说明就容易被理解。由于这些原因，用自然语言中的定义和形式规格语言中的定义这两种形式常常同时给出。

本附件构成了 SDL 的一种形式定义。如果在本文中所规定的某个 SDL 概念的任何性质与在 Z.100 中所规定的相应性质相抵触，而这个概念与 Z.100 中所规定的是一致的，则以 Z.100 中的定义为准，而本形式定义则需要修正。

2.1 元语言

在本形式定义中所使用的元语言是 Meta-IV [1]。选择该语言的理由如下：

- 它建立在一个非常强的和广泛研究过的数学基础之上。
- 它具有用于实物操作的非常便利和强有力的功能。
- 它具有“编程式样”的表示法，这意味着它是面向程序员和实现者的。
- 它在欧洲共同体范围内正处于标准化的过程之中。
- 它在书籍、会议录和科学杂志中已经有了很多的报导，并且已经用于 CCITT 的 CHILL 语言形式定义手册 [2] 之中。该手册中也包含了 Meta-IV 表示法的摘要。
- Meta-IV 有一些工具是可以利用的，这些工具能够进行语法检查、可见性分析、文件生成和交叉参照等。

在第 5 章中，将对用于形式定义中的 Meta-IV 的各部分进行日常使用的介绍。Meta-IV 的完整定义参见 [1]。

3 模型化技术

在考虑“SDL 的语义”是什么含意的时候，从概念上把语言定义分解为下面几部分是合适的：

- 语法规则的定义。

- 静态语义规则的定义（所谓良形式的条件），例如哪些名字允许用于给定的地方，哪些值允许赋给变量等。

- 当对语言构件进行解释时它们的语义定义（即动态语义）。

没有必要在形式定义中包括语法规则，因为 Z.100 中的 BNF 规则和语法图已经用来作为语法规则的形式定义了，这意味着对形式定义的输入在语法上已是正确的 SDL 规格说明。该输入用抽象语法来表示。这种抽象语法的基础是 SDL 正文具体语法分析树（BNF 规则），其中已经去掉了诸如分隔符和词法规则等无关的细节。因此，这里的抽象语法并不是在各建议中出现的 Z.100 的抽象语法，那里的抽象语法是对 SDL 模型概念的抽象。

例如，抽象语法的产生式规则：

1 ***Transstring*** :: ***Actstmt***$^{+}$ [***Termstmt***]

表示跃迁串（*Transstring*）由一个非空的动作语句序列（*Actstmt*$^{+}$）和任选的终止符语句（*Termstmt*）组成。（在产生式规则中也出现斜体字母）。

在抽象形式上规定 SDL 语法的完整的产生式规则集（即所谓的域定义）称为 AS_0。从某方面来说，AS_0 是在比 Z.100 中语法规则更基本的层次上来规定语言语法的。在 Z.100 中的具体正文语法包含了大量的语义信息（它是上下文有关的），这与 AS_0 有所不同的。应注意的是 AS_0 是对具体正文语法的一种抽象。具体图形语法尚未得到使用，是因为要节省时间与空间，而不是因为在本任务中有任何困难。

作为一个例子，把 Z.100 中的信号表规定如下：

<signal list> ::= <signal item> {,<signal item>}
<signal item> ::= <signal identifier> | (<signal list identifier>) | <timer identifier>

而在 AS_0 中相应的定义为：

2 ***Signallist*** :: ***Signalitem***$^{+}$
3 ***Signalitem*** = ***Id*** | ***Signallistid***

其中信号表（*Signallist*）由多个信号项（*Signaliterm*）组成，而信号项或是一个标识符（*Id*）或是一个信号表标识符（*Signallistid*）。BNF 产生式〈信号项〉（〈*Signal item*〉）是上下文有关的。与此相反，在 AS_0 中并不区分信号标识符（Signal identifier）和定时器标识符（timer identifier），因为在语法上与信号表相对比，它们都是标识符，而信号表则要加用括号以示区别。

FD 的出发点在语法上是正确的 SDL 规格说明。形式定义的任务是：

- 规定 SDL 规格说明的良形式条件。这项工作（叫做静态语义）构成了附件 F.2。

- 规定 SDL 规格说明的动态性质。这项工作（叫做动态语义）构成了附件 F.3。

步骤如图 1 所示，来自静态语义的结果（即 AS_1）在后面解释。

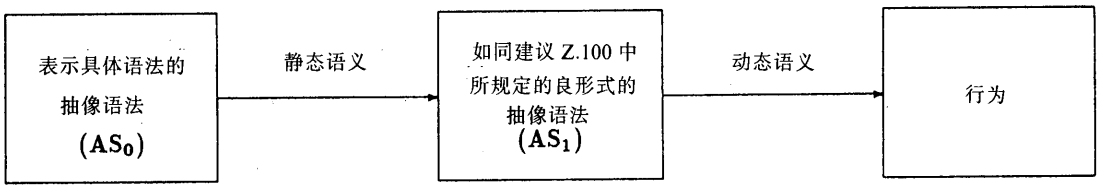

图 1
静态语义和动态语义的目标

把具体正文语法翻译为 AS_0 的步骤尚未正式作出规定，但是可以从在两种语法中名字间的对应性推导出来，就象前面介绍的信号表那样。

3.1 静态语义

在 Z.100 中，各种构件的动态语义是用抽象语法来规定的。公共部分（具体正文语法和具体图形语法）规定了具体语法规则，陈述了合适的良形式条件，并且把具体语法规则和 Z.100 中的抽象语法联系起来。这部分是利用 Meta-IV 来规定的（在公共部分的抽象语法里）。同样的抽象语法也用于形式定义中（这里把它叫做 AS_1）。在 Z.100 的附件 B 中给出了这个抽象语法的摘要。

除了规定**良形式条**件外，静态语义还必须规定怎样把一个规格说明的 AS_0 表示法变换为 AS_1 表示法，也就是说，如果所给定的 AS_0 表示法是良形式的话，有了一个 AS_0 表示法，就可由静态语义来返回到一种 AS_1 表示法。静态语义可以被看成是一个“抽象编译程序”，这里可把 AS_0 表示法看成是源语言，而把 AS_1 表示法看成是目标语言。

除了 AS_0 和 AS_1 外，静态语义使用某些用于内部的域，称为语义域，它保持对某给定实体在任何地方所需的信息。例如，当变换某个进程定义的时候，有关它的形式参数的信息就要保存在语义域中。以后在对创建请求动作进行变换时，就可以再恢复这些信息。AS_0 域也是可以用于那种目的的，因为不管怎么说，语义域就是从 AS_0 推演出来的。然而当需要查找出现在树中某处的某实体（例如进程定义）的信息时，树形表示法是不方便的。因此语义域通常采用模拟表格的映射。

例如，语义域包括一个把标识符映射为含有该标识符信息的某个描述符（进一步的解释参见第 5.4.7 节）：

```
4  Descriptordict        = Qual m→Descr
```

其中 *Qual* 是在形式定义内部使用的标识符的表示，而 *Descr* 是任意的描述符。例如该描述符可以是一个进程描述符：

```
5  Descr                 = ProcessD | ...
6  ProcessD              :: ParameterD* Validinputset Outputset
```

这里表达，一个进程描述符 *ProcessD* 包含一个参数描述符表 *ParameterD**、有效输入信号集 *Validinputset* 的信息和输出信号集 *Outputset* 的信息。这里没有给出这三种（子）描述符的定义。

变换本身是由一组 Meta-IV 函数来完成的，它要用到 AS_0，AS_1 和语义域这三个域。

3.2 动态语义

动态语义的任务是为 AS_1 形式的 SDL 规格的行为作出规定。

动态语义分为以下三个主要部分：

- 作为基础系统的模型（抽象的 SDL-机器）
- 进程图的解释
- 把 AS_1 变换为更适当的表示法，即，构成一个映射（一个语义域）。其中包含在解释执行过程中所需的信息，例如关于变量的类别、进程之间可能的通信路径、类型的等价类等信息。该映射被命名为 *Entity-dict*（或更正确地说，该映射的域取名为 *Entity-dict*）。

在动态语义中，SDL 中的并发性用*元进程*来模拟，也就是用 Meta-IV 中并发执行的元进程来模拟并发执行的 SDL 进程。

采用了下面六种不同的元进程类型：

- *系统*（*system*）
  处理信号的选路和创建各 SDL 进程（sdl-process）。
- *路径*（*path*）
  处理各通路不确定的延迟。
- *定时器*（*timer*）
  时常注意当前时间和处理超时。
- *视见*（*view*）
  时常掌握所有已显示变量的情况。
- *sdl 进程*（*sdl-process*）
  解释一个 SDL 进程的行为。
- *输入口*（*input-port*）
  安排 SDL 进程中信号的排队。每一个 sdl 进程实例恰好有一个输入口实例。

系统、路径、定时器和视见这四种元进程类型从整体上可以看成是基础系统的模型。

元进程之间无共享数据。他们之间的交互作用是通过传送*通信域*（对应于 SDL 概念中的信号）的实例（实物）所传送的值来实现的。

通信域的规定方法和其它域相同，例如，通信域输入信号的实物从它所附属的输入口实例指向某个 sdl 进程实例。通信域的规定象下面这样：

7 *Input-Signal* :: *Signal-Identifier*$_1$ [*Value*]* *Sender-Value*

输入信号（*Input-Signal*）的实例传递三项信息：所传送的SDL信号的标识符（Signal-Identifier），由SDL信号传送的若干个值［Value］*和发送者的PID值（Sender-Value）。

图2显示了完整的“元进程交互作用体制”。通信机制是同步的，所采用的符号表示法叫做CSP（参见［3］和［4］）（Communicating Sequential Processes）。

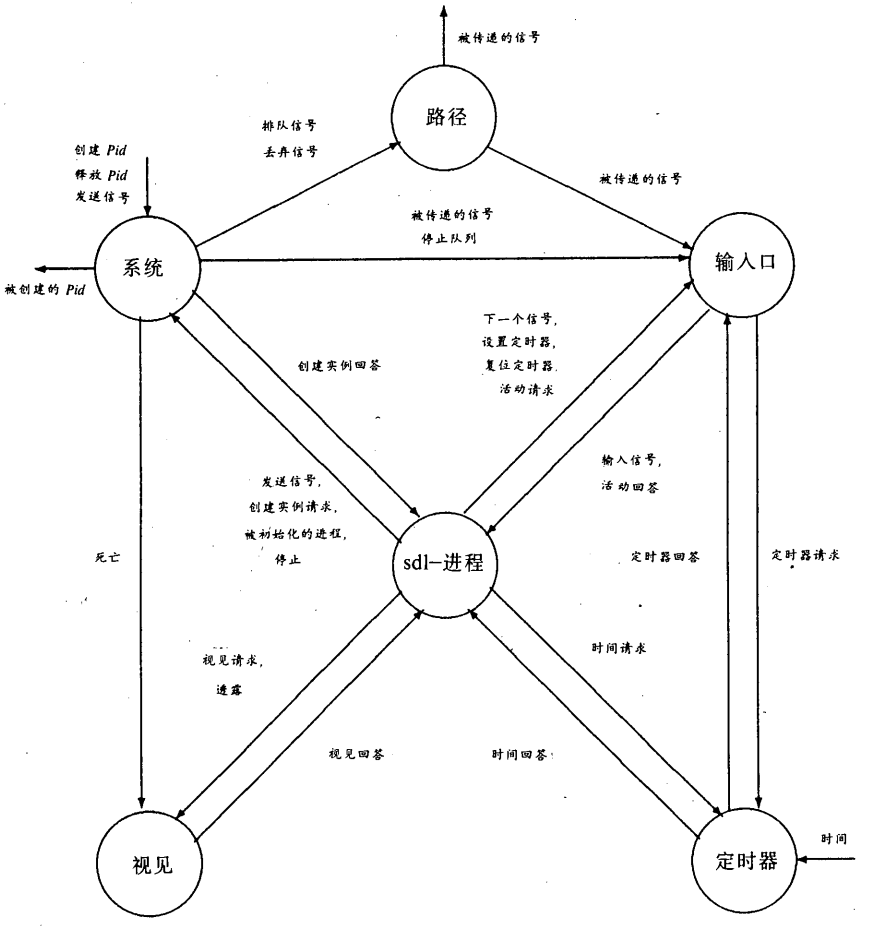

图2
通信方案

3.3 举例

图3说明在下述简化的SDL进程的形式定义中元进程之间的通信，所描述的过程为：有一信号（“b”）从环境到达，而该进程则回答一个信号（“a”）给环境作为响应。

...

状态（state）S；

输入（input）b；

输出（output）a；

…

这一通信过程借助于一张信息序列图非形式地加以说明。路径（1）和路径（2）表示路径处理器的两个实例，分别对应于从环境到 sdl 进程的路径（路径（1））和反方向的路径（路径（2））。

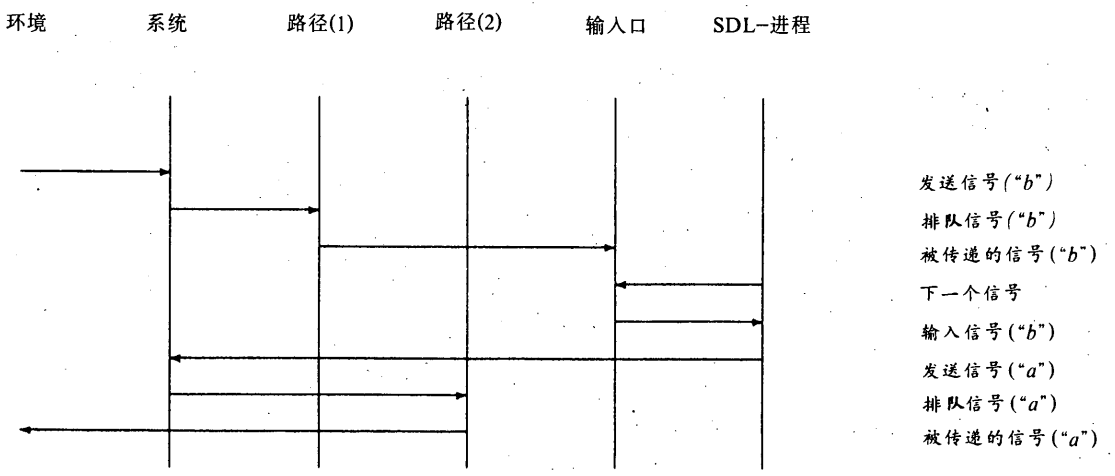

图3

元进程间通信的例子

3.4 *形式定义的物理结构*

把静态语义（附件 F. 2）分为三个主要部分：

1. AS_0的域定义
2. 语义域的域定义
3. 检查良形式条件的 Meta-IV 函数与规定如何把 AS_0变换为 AS_1。

对于用于第2和3部分中的 AS_1的域定义可在 Z. 100中找到，并在 Z. 100的附件 B 中摘要地作了介绍。在本形式定义中不再重复。

附件 F. 2也包括对 Meta-IV 的函数名和域名（包括定义性出现和应用性出现）之间的交叉索引和对所应用的良形式条件的交叉索引。

把动态语义（附件 F. 3）分为5个主要部分：

1. 通信域的域定义
2. 语义域（*Entity-dict*）的域定义
3. 用于基础系统模型的元进程定义和附加函数。

4. 用于解释 SDL 进程的元进程定义和附加函数。

5. 内部域 *Entity-dict* 的创建。因为 *Entity-dict* 是由 SDL 进程使用的，所以它应在任何 SDL 进程被解释执行之前创建。

和附件 F. 2一样，附件 F. 3的目录中也包含有域名、函数名、元进程名、错误条件等内容。

建议材料的篇幅（尤其是附件 F. 2）乍看起来似乎吓人，其实其中大部分内容都是关于域、函数和进程定义的注释。

函数和进程的定义方式如下：

1. 首先，由以下部分给出函数或进程的定义：

   (a) 有一个首部规定该进程或函数的名字和其形式参数名字

   (b) 它的主体（算法）

   (c) 用一个类型子句说明形式参数的类型（域）和结果的类型（如果有的话）。

2. 然后是附加于进程或函数定义的如下一些分项目的注释（用简明英语）：

目的 说明函数或进程的目的
参数 说明属于函数或进程的每个形式参数的目的
结果 说明返回的内容（如果有的话）
算法 逐行地说明用于此函数或进程的算法

举例

摘自附件 F. 2的最外层函数 *definition-of-SDL* 如下所示，它把静态语义（变换系统）和动态语义联系起来了（用启动元进程 *system* 的办法）。

$$definition\text{-}of\text{-}SDL(extparms, systemdef, predefsorts) \triangleq$$

```
1  (let (as1, auxinf) = transform-system(systemdef, predefsorts, extparms) in
2   if as1 = nil then
3     undefined
4    else
5     (let subsetcut = select-consistent-subset(as1, extparms) in
6      start system(as1, subsetcut, auxinf)))
```

type: $External\text{-}Information\ Sys_0\ Datadef_0^+ \Rightarrow$

目的 规定 SDL 的性质

参数

extparms 某些外部信息（参见附件 F. 2中的第2. 3节）

systemdef 表示 SDL 系统的 AS_0树

predefsorts 使用 AS_0形式的预定义数据

算法

第1行　把系统变换为抽象语法形式（AS_1形式）
第2行　如果发现静态错误（即，如果不能导出AS_1的表示法），则该行为没有定义。
第4行　如果没有发现静态错误，则
第5行　选择代表一致性子集的*功能块标识符*$_1$的集合。
第6行　创建一个系统实例，即，创建一个 Meta-IV 进程，它的行为象基础系统的行为。

4　怎样使用形式定义

4.1　SDL 用户

本形式定义不打算用来作为 SDL 的用户参考手册。SDL 的新入门者可以发现用户指南（Z.100中的附件 D）更适合于获得该语言的整体概念（和它们的理论基础），而建议 Z.100本身则作为 SDL 的参考手册。但是可能在有的情况下 Z.100是不够充分的，例如

- 如果某些性质丢失（例如，某些期待的静态条件），如果某些所宣称的性质和其它性质有矛盾，或者
- 如果某些所宣称的性质的精确意义难以理解，或者
- 如果某些性质（由于在 Z.100中缺少交叉索引）难以找到，或者
- 如果用户想获得对某些较复杂问题的更深刻的理解，例如抽象的 SDL 自动机、何时和怎样选择一致性子集、通过上下文来确定语义、继承机制等之类的问题。

在这些情况下，形式定义将是一个有用的支持文件。当然用户应该首先对形式定义的结构有深入的理解，知道函数是怎样组织的和域是干什么用的。有关 Meta-IV 表示法的一定的知识也是需要的，但是在阅读了 Meta-IV 的介绍（参见下面第5节）之后，由于对这些函数有大量的注释，通过阅读这些函数和相关的注释来理解 Meta-IV 是可能的。当查阅形式定义的时候，用户可以利用目录和交叉索引。

4.2　实现者

如前所述，Meta-IV 手段允许实现者从 Meta-IV 的规格系统地导出一种实现（即静态语义分析器、模拟器等）。对于 SDL，从附件 F.2导出静态分析器和从附件 F.3导出模拟器是可能的，建议利用 AS_1表示法（由静态分析器生成的）作为模拟的基础。理由是在 AS_0中标识符的上下文信息丢失了（在 AS_0中通常对标识符没有加以限定）和由于 SDL 中有大量的速记符号（尤其是象数据类型之类的概念），这就使得很难从 AS_0形式表达的规格导出其动态语义。

应该指出求得一种实现方法的推导过程是系统的，但不是机械的。

下列几点是必须要考虑的：

- 必须寻找适当的数据类型来表示 Meta-IV 中理想的数据类型（域），例如用于 AS_0、AS_1和语义域中的映射、列表和集合。
- 由于 SDL 中的可见性规则（标识符可在给它们下定义之前使用的事实），（为方便起见）在静态语义中使用了所谓的“定点等式”（参见附件 F.2的第3.1节）。在某个实现中，通过遍历 AS_0树若干次可

以逐步地创建语义域（例如，必须在任何信道描述符创建之前先创建信号的描述符，因为信道在它们的定义中要涉及到信号）。

- 最初的代数方法意味着形式定义处理无限个实物。AS_1也含有无限个实物。因此有必要对 AS_1稍加修改，并且对数据类型的使用施加限制，或者使用某些抽象技术可以对这些实物加以编码。

5 Meta-IV 介绍

本节包括对 Meta-IV 的非形式介绍和在形式定义中怎样使用 Meta-IV，即，Meta-IV 是借形式定义（缩写为 FD）来说明的，这意味着仅对已经用于 FD 中的 Meta-IV 的那些部分进行说明。

5.1 一般结构

FD 包括：

- 一组函数和进程的定义用来规定 SDL 的语义。进程（在 Meta-IV 和 FD 中称为处理器）用于模拟并发性，因而仅用于动态语义。在语法上，处理器的定义看上去象函数定义（不同之处仅是在处理器名字后面采用了关键字 *processor*），因而，下述函数概念的描述也适用于处理器。

- 一组域定义用来规定由函数来处理的实物的类型。引入了表示某些域定义组的术语，以便在逻辑上把它们分类。我们有描述具体语法表示法的 AS_0域、描述 SDL 抽象语法的 AS_1域。我们还有 *Dict* 域和 *Entity-dict* 域的集合用来分别表示静态和动态语义的“内部”使用的域（语义域）。在这一节中，我们经常使用“值”作为实物的同义词，用“类型”作为域的同义词。

- 一组全局常数的定义。在 FD 中，只提供两种这样的定义。在静态语义的第3.13节中对它们下定义。对于理解 FD 而言，它们是不很重要的。

各定义可按任何次序说明，而定义中引入的名字可以在正文中给它们下定义之前引用。

5.2 函数定义

函数定义包括以下三部分：

1. 首部由函数名字开始后随有一个或两个形式参数表。每个形式参数表用括号括起来。把参数归入哪一个表没有形式上的意义。有一些参数被放入一个单独的（第二个）参数表中，往往是因为对求值来说它们并不是最重要的。例如，对于由函数经常使用的语义域就是如此。这些语义域往往在一个单独的参数表中提供。

2. 函数体，它可以是一个表达式，也可以是语句序列。函数可以不给出任何结果（参见下面）。

3. 类型子句说明形式参数的类型和结果的类型。首先，说明第一参数表的类型并跟一个箭头（→或⇒），然后说明第二个参数表（如果有的话）的类型，再跟另一个箭头，最后说明结果的类型。

举例

$f(a, b)(d) \triangleq$

1 /* *expression* */

type: $DomX\ DomY \rightarrow DomZ \rightarrow DomW$

在本例子中：

f 是函数名

a，b，d 是形式参数，a 和 b 在第一参数表中，d 在第二参数表中。a 的类型是域 $DomX$，b 的类型是域 $DomY$，d 的类型是 $DomZ$。结果的类型是 $DomW$。$DomX$、$DomY$ 和 $DomW$ 这几个域必须在某些域定义中给出定义。

如果形参或结果的使用与类型子句不一致，在 Meta-IV 的规格说明中就存在错误。在上面的例子中，非形式 Meta-IV 正文（包含在/* */中的正文）用于描述某个 Meta-IV 表达式（为了节省空间起见已经省略）。非形式的 Meta-IV 正文类似于 SDL 中的非形式正文，它广泛地用于本节中的各个例子里。

通常，对**应用性**（applicative）和**强制性**（imperative）函数要加以区别。应用性函数是不涉及全局状态（变量）部分的一类函数，即，这类函数的结果仅依赖于所使用的实在参数值。因为语句会影响状态的某种变化，所以应用性函数的体被限制为一个表达式。应用性函数总会给出一个结果的。强制性函数是一类涉及甚至会改变全局状态的函数（即函数具有副作用）。如果函数是强制性的，为了反映这一点在类型子句中指定结果时，必须用⇒取代→。

即：

$f(a, b)(d) \triangleq$

1 /* *expression referring to the global state or sequence of statements* */

type: $DomX\ DomY \rightarrow DomZ \Rightarrow DomW$

在 FD 中，静态语义以及动态语义中的内部域 *Entity-dict* 的创建都是应用性的。

5.3 变量的定义

在处理器定义中的最外层对全局变量来下定义。它们对规定该变量的处理器所使用的所有函数都是可见的，即使这些函数的定义通常是在处理器定义之外。然而，由两个或多个处理器共享的函数不允许访问变量。当一给定处理器的几个实例存在的时候，由该处理器所规定的变量的几个实例也存在。（这里并没有共享的变量）。

变量定义由规定的关键字 **dcl** 开头，后面跟有变量名字表来引入的。各名字任选地后随一个初始化表达式，最后是变量类型。

例如

dcl v1 := 5 **type** *Intg*;
dcl v2 **type** *DomD*;

此处规定了两个变量 v1和 v2，v1是整型的并初始化为5。v2是 *DomD* 类型的。请注意，变量在语法上总是可以和其它名字区别开来，因为它们不用斜体字印刷。这个变量定义的另一种语法形式是：

dcl v1 := 5 **type** *Intg*,
v2 **type** *DomD*;

变量的值可通过使用内容操作符（其关键字为 c）来访问。

例如

$f() \triangleq$

1 cv1 + cv2

type: $() \Rightarrow Intg$

5.4 域

域的定义通常在文件的开头给出。域名的第一个字母是大写的，因此域名可以在语法上和其它名字区别开来。域的定义是指定的域名后随符号::（或在同义名字的情况下后随符号=，见第5.4.1节的说明），再后随一个反映它的性质的域表达式（对域的表示方法的介绍也请参见 Z.100的§1.5.1）。

举例

8 *Output-node*$_1$ :: *Signal-identifier*$_1$
[Expression$_1$*]**
[Signal-destination$_1$*]*
Direct-via$_1$

本例取自 SDL 的抽象语法（为了清楚起见，在 FD 中 AS$_1$的所有名字都带下标 "$_1$"）。它规定了一棵**有名字的树**，即，一个象记录那样的数据类型。此处，记录类型的名字是 *Output-node*$_1$，它的字段具有类型 *Signal-identifier*$_1$、[*Expression*$_1$]*、[*Signal-destination*$_1$] 和 *Direct-via*$_1$。

对有名字树的最重要的操作符是 mk-（make）操作符，把它用于组成和分解树的实物（即记录的值）。

例如，如果名字 *sigid* 表示域 *Signal-identifier*$_1$的一个实物、名字 *exprlist* 表示域[*Expression*$_1$]*的一个实物、名字 *dest* 表示类型 [*Signal-destination*$_1$] 的一个实物和名字 *via* 表示域 *Direct-via* 的一个实物的话，则域 *Output-node*$_1$ 的一个实物按下述书写形式来构成：

mk-*Output-node*$_1$(*sigid*, *exprlist*, *dest*, *via*)

它可用于 Meta-IV 表达式中。请注意，在 mk－操作符中指定的变量次序是有意义的。这也适用于函数调用。

类似地，如果我们有域 *Output-node*$_1$的一个命名为 *outputnode* 的实物，并且想访问这些字段的话，可以通过分解它来引入字段的名字（这些名字选得和上面的名字一样）：

```
let mk-Output-node1(sigid, exprlist, dest, via) = outputnode in
/* some expression using the fields */
```

借助 let 构件，我们已引入名字来表示实物 *outputnode* 中的各个字段。采用 let 构件是引入实物名字的一般方法（不仅仅是和 mk-操作符组合起来用）。第5.5节将对 let 构件作进一步的说明。

如果在表达式中没有用到某些字段，我们可以在分解中略去对应的名字。例如，如果 *sigid* 在表达式中没有用到，我们可以这样写：

```
let mk-Output-node1(, exprlist, dest, via) = outputnode in
/* some expression using exprlist and dest */
```

如果在表达式中仅要使用 *Signal-Identifier*$_1$，我们也可以使用字段选择符 s-：

let *sigid* = s-*Signal-Identifier*$_1$(*outputnode*) **in**
/* *some expression using sigid* */

只有当字段可以由域名唯一地确定时，才能使用字段选择操作符。
我们可以在函数首部而不是函数体中分解形式参数，如果这样做可使读者更容易理解的话，即：

int-create-node(**mk**-*Create-request-node*$_1$(*pid*, *exprl*))(*dict*) ≙

1 /* *body of int-create-node* */

type: *Create-request-node*$_1$ → *Entity-dict* ⇒

等价于

int-create-node(*createnode*)(*dict*) ≙

1 (**let** **mk**-*Create-request-node*$_1$(*pid*, *exprl*) = *createnode* **in**
2 /* *body of int-create-node* */)

type: *Create-request-node*$_1$ → *Entity-dict* ⇒

请注意本例中有第二参数表，它含有形式参数 *dict*，其域为 *Entity-dict*。

5.4.1 同义词

只有在域定义中的字段是由一个名字表示时，使用字段选择操作符才是可行的。例如，如果要在域 *Output-node*$_1$的实物的第二个字段上使用选择操作符，则必须用稍有不同的方法来规定 *Output-node*$_1$：

9 *Output-node*$_1$ :: *Signal-identifier*$_1$
Valuelist
[*Signal-destination*$_1$]
Direct-Via$_1$
10 *Valuelist* = [*Expression*$_1$]*

这里的 *Output-node*$_1$的域与先前下过定义的 Output-node$_1$的域完全一样。仅有的差别是我们给第二个字段起了一个名字，即，我们已经规定了域表达式 [Expression$_1$]*的一个同义词或速记符（符号“=”用于规定同义词）。常常有其它理由要求规定同义词，例如，同一个域表达式用在几个地方或为了提高可读性。例如，在 SDL 的抽象语法中，我们有 *Channel-name*$_1$，*Block-name*$_1$，Process-name$_1$等，它们都是域 *Name*$_1$的同义词。但是它却带给读者有关实物的信息：由不同的 Name$_1$所代表的实物是属于哪一类实体的。另一种典型的情况是我们有一个长的替换表，例如，对 *Expression*$_1$的抽象句法是

11 *Expression*$_1$ = *Ground-expression*$_1$ |
Active-expression$_1$
12 *Active-expression*$_1$ = *Variable-access*$_1$ |
Conditional-expression$_1$ |
Operator-application$_1$ |
Imperative-operator$_1$
13 *Imperative-operator*$_1$ = *Now-expression*$_1$ |
Pid-expression$_1$ |
View-expression$_1$ |
Timer-active-expression$_1$

上述定义较之下面的定义方法较好地反映了各类表达式的分组。

14 *Expression*$_1$ = *Ground-expression*$_1$ |
Variable-access$_1$ |
Conditional-expression$_1$ |
Operator-application$_1$ |
Now-expression$_1$ |
Pid-expression$_1$ |
View-expression$_1$ |
Timer-active-expression$_1$

5.4.2 *无名树*

在某些情况下，我们不需要为一棵树的定义命名。**无名树**广泛地用于 FD 中。但是由于它们经常不需要显式地下定义，所以它们是隐名的。

举例

在动态语义中 *Entity-dict* 的定义的第一行是：

15 *Entity-dict* = (*Identifier*$_1$ *Entityclass*) $\overrightarrow{m}$ *Entitydescr*

它表示 *Entity-dict* 包括一个从两个域 *Identifier*$_1$和 *Entityclass* 到某个描述符（*Entitydescr*）的映射。这两个域构成了一棵无名树。如果应该使用有名树，我们就必须把定义重写为：

16 *Entity-dict* = *Pair* $\overrightarrow{m}$ *Entitydescr*
17 *Pair* :: *Identifier*$_1$ *Entityclass*

举例

动态语义中的可达性（*reachability*）的定义为

18 *Reachability* = (*Process-identifier*$_1$ | ENVIRONMENT)
Signal-identifier$_1$-**set** *Path*

此处我们已经为包含三个字段的一棵无名树规定了一个同义词。这三个字段是：

1. 一个字段包含进程标识符或包含字面值 **ENVIRONMENT**

2. 一个字段包含信号标识符集合

3. 域 *Path* 的字段

正如所显示的那样，在域定义中的括号用于规定无名树和用于把可替换成份括在一起。

举例

在动态语义中，把函数 *make-formal-parameters* 下定义为：

make-formal-parameters(*parml*, *level*) ≙

1 /* *The body, which is not shown here* */

type: *Procedure-formal-parameter*$_1$* *Qualifier*$_1$ → *FormparmD** *Entity-dict*

这个函数返回两个实物：*FormparmD** 和 *Entity-dict*，这在实际上意味着返回了一棵由两个实物组成的无名树。

mk-操作符不能用于无名树，这些实物的组合与分解办法是用括号把所需的字段括起来。

举例

一个 *Reachability*（可达性）实物的组成如下，这里 *a* 表示一个 *Process-Identifier*$_1$，*b* 表示一个信号标识符集合，*d* 表示 *Path*：

(*a*, *b*, *d*)

如果为了提高可读性，要求用一个名字表示该实物（使用一个名字比使用（*a*, *b*, *d*）更容易，尤其是在一个表达式中好几次出现（*a*, *b*, *d*）时），那么我们又可以利用 **let** 构件。也就是说表达式

/* *some expression using "(a,b,d)"* */

等价于

(**let** *reach* = (*a*, *b*, *d*) **in**
/* *some expression using "reach"* */)

let 构件也用于分解无名树的实物，例如，一个命名为 reach 的 Reachability 实体的分解（此处因某种原因，我们没有利用信号标识符集）是：

let (*a*, , *d*) = *reach* **in**
/* *some expression using a and d* */

当调用一个函数的时候，往往要分解表示函数调用结果的无名树，即：

let (*parmlist*, *pathlist*) = *make-formal-parameters*(..., ...) **in**
/* *some expression using the function results parmlist and pathlist* */

等价于

let *parminf* = *make-formal-parameters*(..., ...) **in**
let (*parmlist*, *pathlist*) = *parminf* **in**
/* *some expression using the function results parmlist and pathlist* */

5.4.3 分枝构件

在某些情况下，必须能够把许多树的实物互相区别开来。例如先前下过定义的 *Imperative-operator*$_1$的同义词的实物可以是 *Now-expression*$_1$, *Pid-expression*$_1$, *View-expression*$_1$等。如果手头有一个 *Imperative-operator*$_1$，在我们对它求值之前，必须首先确定该 *Imperative-operator*$_1$的类型。为此目的，我们可以使用分情况（case）表达式或语句。例如，计算强制性 SDL 表达式的函数看上去类似于：

```
eval-imperative-expression(expr) ≜

 1  cases expr:
 2   (mk-Now-expression1()
 3      → eval-now-expression(),
 4    mk-View-expression1(vid, pidexpr)
 5      → eval-view-expression(vid, pidexpr),
 6    mk-Timer-active-expression1(tid, actlist)
 7      → eval-timer-expression(tid, actlist),
 8    T → eval-pid-expression(expr))

type:  Imperative-operator1 ⇒
```

请注意，我们是根据 *Imperative-operator* 的类型来进行分枝，而不是靠树中各字段的值。T 表示“否则”子句，它之所以用在此，是因为在 *Imperative-operator*$_1$（*Pid-expression*$_1$）中最后的候选成份是一个表示其它四种候选成份的同义词，而这里我们并不想区别这四种候选成份。这些候选成份的求值要服从于 *eval-pid-expression*。

另一种实现方法是使用布尔操作符 is-，如果给定的作为变量的实物是属于某个域的，is-操作符就返回 true（真）。例如：

```
eval-imperative-expression(expr) ≜

 1  if is-Now-expression1(expr) then
 2    eval-now-expression()
 3   else
 4    if is-View-expression1(expr) then
 5      eval-view-expression(s-Variable-identifier1(expr), s-Expression1(expr))
 6     else
 7      if is-Timer-active-expression1(expr) then
 8        (let mk-Timer-active-expression1(tid, actlist) = expr in
 9          eval-timer-expression(tid, actlist))
10       else
11        eval-pid-expression(expr)

type:  Imperative-operator1 ⇒
```

请注意，用分解的方法访问字段（第8行）和用字段选择操作符来访问字段（第5行）这两种情况在此都给出了例子。

象大多数其它编程语言和规格说明语言一样，要求分情况表达式/语句中的各候选者是“常数”（当我们根据树的类型来分分枝时，它们所取的值）。这意味着如果各候选者具有动态性质（譬如说变量或形式参数）就必须使用 if-then-else 构件。然而，if-then-else 构件还有另一种表示法，即所谓的麦卡锡（Mc-Carthy）构

件，如果有很多候选者的话，这种构件更为合适。

```
eval-imperative-expression(expr) ≜

1  (is-New-expression1(expr)
2     → eval-now-expression(),
3   is-View-expression1(expr)
4     → (let mk-View-expression1(vid, pidexpr) = expr in
5          eval-view-expression(vid, pidexpr)),
6   is-Timer-expression1(expr)
7     → (let mk-Timer-expression1(tid, actlist) = expr in
8          eval-timer-expression(tid, actlist)),
9   T → eval-pid-expression(expr))

type:  Imperative-operator1 ⇒
```

请注意，某些 FD 函数的名字也用“is-”开头。这些情况很容易和“is-”操作符区别开来，因为它们不是黑体字。

5.4.4 基本域

Meta-IV 提供了一些预先定义的基本域，下面介绍它们的符号和相关联的操作符。

5.4.4.1 布尔域（Boolean）

Meta-IV 的名字 *Bool* 用来表示真实性的值的域，即，集合 {**true**，**false**}

布尔域的操作符

符号	类型			操作
¬	*Bool*	→	*Bool*	非
∧	*Bool*	→	*Bool*	与
∨	*Bool*	→	*Bool*	或
⊃	*Bool*	→	*Bool*	蕴含
=	*Bool Bool*	→	*Bool*	相等
≠	*Bool Bool*	→	*Bool*	不等

举例

用 Meta-IV 表达式可以说明布尔操作符¬、∧、∨和⊃的性质如下：

$\neg a$ = (**if** a **then false else true**)
$a \vee b$ = (**if** a **then true else** b)
$a \wedge b$ = (**if** a **then** b **else false**)
$a \supset b$ = (**if** a **then** b **else true**)

5.4.4.2 整数（Integer）

对整数值预先规定了三个域名：

· 名字 *Intg* 表示所有整数值的域，即，集合 {…−2，−1，0，1，2，…}

· 名字 N_0表示非负整数值的域，即，集合 {0，1，2，…}

· 名字 N_1表示正整数的值的域，即，集合 {1，2，3，…}

整数的操作符：

符号	类型	操作
−	*Intg* → *Intg*	非
−	*Intg Intg* → *Intg*	减
+	*Intg Intg* → *Intg*	加
*	*Intg Intg* → *Intg*	乘
/	*Intg Intg* → *Intg*	整除
mod	N_0 N_1 → N_0	模
=	*Intg Intg* → *Bool*	相等
≠	*Intg Intg* → *Bool*	不等
<	*Intg Intg* → *Bool*	小于
≤	*Intg Intg* → *Bool*	小于或等于
>	*Intg Intg* → *Bool*	大于
≥	*Intg Intg* → *Bool*	大于或等于

5.4.4.3 字符（Character）

Meta-IV 名字 *Char* 用来表示 ASCII 字符值的域。对于可打印的字符，使用引号括起来的办法来表示实物，例如" a"," z"," "。

字符操作符

符号	类型	操作
=	*Char Char* → *Bool*	相等
≠	*Char Char* → *Bool*	不等
<	*Char Char* → *Bool*	小于
≤	*Char Char* → *Bool*	小于或等于
>	*Char Char* → *Bool*	大于
≥	*Char Char* → *Bool*	大于或等于

关系操作符作用于有关的 ASCII 数字值。

为了提高可读性，域 $Char^+$的实物可以用括在引号中的字符序列来表示。例如," abc" 与〈" a"," b"," c"〉相同（参见第5.4.6节）。

5.4.4.4 引用文（Quotation）

用 Meta-IV 的名字 *Quot* 来表示引用文的域。它们是不同的基本实物，并且用黑体字的大写字母和数字的任意序列来表示，例如，**ENVIRONMENT**，**REVERSE**。

引用文的操作符

符号	类型	操作
=	*Quot Quot* → *Bool*	相等
≠	*Quot Quot* → *Bool*	不等

相对于其它域，只有当某个（些）*Quot* 的实物可能在给定的上下文中的时候，*Quot* 的实物才可以出现在域定义中。例如，在 Z.100的抽象语法中，*Originating-block*$_1$的定义是：

1 *Originating-block*$_1$ = *Block-identifier*$_1$ | **ENVIRONMENT**

利用 *Quot* 时，*Originating-block*$_1$还可以用另一种方式下定义：

2 *Originating-block*$_1$ = *Block-identifier*$_1$ | *Quot*

然而，在域定义中使用 **ENVIRONMENT** 更准确，因为这个目标在那个上下文中是唯一可能的 *Quot* 值。

5.4.4.5 标志（Token）

在 Meta-IV 中用名字 *Token* 来表示标志的域，该域可以认为是由不同的无需加以表示的基本实物所组成的潜在的无穷集。

标志操作符

符号	类型	操作
=	*Token Token* → *Bool*	相等
≠	*Token Token* → *Bool*	不等

举例

在 Z.100的抽象语法中把 *Name*$_1$下定义为

1 *Name*$_1$ :: *Token*

在解释期间各个 *Name*$_1$所需要的唯一性质是等同性。因此，*Name*$_1$由一个标记的值组成（名字如何实际拼写是无关紧要的）。

5.4.4.6 省略（Ellipsis）

省略域（用...表示）表示一个未规定的构件，它用于域定义或表达式中。

- 每当实际的域或表达式对语义来说是不重要的时候，或
- 每当该域或表达式的详细描述在规格说明的作用域之外。

举例

Z.100的抽象语法中 *Informal-text*$_1$的定义为

1 *Informal-text*$_1$ :: ...

Informal-text$_1$不能用 *Meta*-IV 来解释，因此 Informal-text$_1$中含有某些尚未规定的内容。

5.4.5 集合域（Set Domains）

集合域由元素（element）域加以后缀关键字-**set**（短横线“-”是重要的）构成的。例如下述定义

2 *State-node*$_1$ :: *State-name*$_1$
Save-signalset$_1$
Input-node$_1$-**set**

3 *Save-signalset* :: *Signal-Identifier*$_1$-**set**

表示域 *state-node*$_1$的实物由一个状态名、一个保留信号集（包括一组信号标识符）和一组输入节点组成。集合的值可以用显式的集合构造符来构成，即用花括号把一组表达式括起来，例如用

$\{1, 3, 5, 1\}$

来表示域 *Intg*-**set** 的一个目标，它含有三个整数值1，3，5。更有用的形式是所谓的隐式集合构造符，所构造的集合包含所有满足某种条件（谓词）的那些元素。例如：

$\{i \in Intg \mid 0 \leq i \leq 5 \vee i \bmod 2 = 0\}$

所规定的集合是

$\{0, 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, ...\}$

它读作：属于竖线左边的一些值的集合（可能由一个值或一个域所限定），这些值要能够满足竖线右边的表达式。

空集用 { } 表示。

在下面对集合操作符语义的说明中，s 表示集合 {1，3，5}：

$\in$ 成员关系操作符。
测试元素域中的一个给定元素是否包含在一个集合中，例如，$1 \in s \equiv$ **true** 而 $2 \in s \equiv$ **false**，

$\notin$ 测试元素域中的一个给定元素是否不包含在一个集合中，例如，$1 \notin s \equiv$ **false** 而 $2 \notin s \equiv$ **true**。

$\cup$ 并操作符。
合并两个集合，例如，$\{2, 3\} \cup s \equiv \{1, 2, 3, 5\}$ 而 $s \cup s \equiv s$，

$\cap$ 交操作符。送回两个集合的交集，例如，$\{2, 3\} \cap s \equiv \{3\}$ 而 $\{\} \cap s \equiv \{\}$。

$\setminus$ 补操作符。
从一个集合中排除指定的值的集合，即，$s \setminus \{1, 2\} \equiv \{3, 5\}$ 而 $\{1, 2\} \setminus s \equiv \{2\}$。

$\subseteq$ 真子集操作符。
测试给定集合的元素是否包含在一个集合中，例如，$\{1, 5\} \subseteq s \equiv$ **true**，$s \supseteq \{1, 5\} \equiv$ **false**，而 $s \supseteq s \equiv$ **false**。

$\subset$ 子集操作符。
测试给定集合的元素是否包含在一个集合内或等于一个集合，即，$\{1, 5\} \subset s \equiv$ **true**，$s \subset \{1, 5\} \equiv$ **false** 而 $s \subset s \equiv$ **true**。

card 基数操作符。
送回一个集合中的元素个数，即，**card** $s \equiv 3$ 而 **card** $\{\} \equiv 0$。

union 分布式并操作符。
其变量是一些集合的集合，而结果是包含在该变量中所有集合中元素的并，即，**union** $\{s, \{5, 6\}, \{1, 5, 8\}\} \equiv s \cup \{5, 6\} \cup \{1, 5, 8\} \equiv \{1, 3, 5, 6, 8\}$。

$=$，$\neq$ 测试集合的相等性和不等性。

举例

可以用 Meta-IV 表达式来说明集合操作符 $\notin$，$\cup$，$\cap$，$\subseteq$，$\subset$，**card** 和 **union** 的性质如下：

$$
\begin{aligned}
&element \notin s1 = (\neg(element \in s1))\\
&s1 \cup s2 = \{element \mid element \in s1 \vee element \in s2\}\\
&s1 \cap s2 = \{element \mid element \in s1 \wedge element \in s2\}\\
&s1 \setminus s2 = \{element \mid element \in s1 \wedge element \notin s2\}\\
&s1 \subseteq s2 = (\forall element \in s1)(element \in s2) \wedge s1 \neq s2\\
&s1 \subset s2 = (\forall element \in s1)(element \in s2)\\
&\textbf{card}\ s1 = (\textbf{if}\ s1 = \{\}\\
&\qquad \textbf{then}\ 0\\
&\qquad \textbf{else}\ (\textbf{let}\ element \in s1\ \textbf{in}\\
&\qquad\qquad 1 + \textbf{card}\,(s1 \setminus \{element\})))\\
&\textbf{union}\ s1 \equiv \{element \mid (\exists set \in s1)(element \in set)\}
\end{aligned}
$$

限定符（$\forall$ 和 $\exists$）在第5.6节中说明。

5.4.6 表域（list Domain）

表或元组域的构成方法是在元素域后面加一个后缀"*"（在可能为空表的情况下），或加一个后缀"+"（在其它情况下）。

举例

4 *Signal-definition*$_1$ :: *Signal-name*$_1$
Sort-reference-identifier$_1$*

这个域定义表示一个信号定义由一个信号名和一个类别标识符表组成，该表可以是空表。

一个表的值可以用显式元组构造符来构造。这是一个用尖括号括起来的一组表达式。例如，

⟨11, 12, 11, 13, 14⟩

表示域 *Intg*$^+$（或 *Intg**）的一个实物，它含有5个有序的元素。

空表用 ⟨⟩ 表示。

与集合构造符类似，也有隐式表构造符。例如，在动态语义中的函数 *int-output-node* 中，我们构成一个在一个输出结点中含有所有实参（*exprl*）值的一个元组（*vall*）：

let *vall* = ⟨*eval-expression*(*exprl*[*i*])(*dict*) | 1 ≤ *i* ≤ **len** *exprl*⟩ **in**

这个元组对应于对该表中所有元素的显式列举：

let *vall* = ⟨*eval-expression*(*exprl*[1])(*dict*),
eval-expression(*exprl*[2])(*dict*),
eval-expression(*exprl*[3])(*dict*),
...⟩ **in**

请注意，元组的括号（〈和〉）与关系操作符<和>的形状有明显的不同。

在下述对表操作符的语义说明中，用 l 表示表 〈11，12，11，13，14〉：

hd 送回第一个元素（即表的头）。即，**hd** l≡11。操作符 hd 所作用的变量不允许是空表（〈〉）。
tl 送回已经移去表中第一个元素的表（即送回表尾 *tail*）。例如 **tl***l*≡ 〈12，11，13，14〉。

[*i*] 送回表中的第 i 个元素，例如，l [3] ≡11，l [5] ≡14。指数值不允许小于1或大于表的长度。

len 送回表的长度，例如，**len** l≡5。

elems 送回组成一个表的元素的集合，例如，**elems** l≡ {11，12，13，14}。

ind 送回由表的合法指数值组成的整数的集合，例如，**ind** l≡ {1，2，3，4，5}。

⤵ 连接两张表，例如 l ⤵ 〈0，1〉 ≡ 〈11，12，11，13，14，0，1〉。
conc 连接所有作为变量表成分的那些表，例如 **conc** 〈〈0,7〉,l,〈9〉〉≡〈0,7,11,12,11,13,14,9〉

=，≠ 测试表的相同性和不等性。

举例

用 Meta-IV 表达式可以说明表操作符 hd，tl，ind，elems 和 conc 的性质，如下：

$$\mathbf{hd}\ l = (\mathbf{if}\ l = \langle\rangle\ \mathbf{then\ undefined\ else}\ l[1])$$
$$\mathbf{tl}\ l = \langle l[i] \mid 2 \leq i \leq \mathbf{len}\ l \rangle$$
$$\mathbf{ind}\ l = \{i \mid 1 \leq i \leq \mathbf{len}\ l\}$$
$$\mathbf{elems}\ l = \{l[i] \mid i \in \mathbf{ind}\ l\}$$
$$\mathbf{conc}\ l = (\mathbf{if}\ l = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \mathbf{hd}\ l \frown \mathbf{conc}\ \mathbf{tl}\ l)$$

5.4.7 映射域（Map Domains）

映射域（也就是一个表格）是这样来构造的，规定作为表中项目的实物的域，后随操作符$\xrightarrow[m]{}$，再后随包含在映射（范围值）中的实物的域。

举例

$$\begin{aligned} 5\quad \textit{Entity-dict} = {} & (\textit{Identifier}_1\ \textit{Entityclass}) \xrightarrow[m]{} \textit{Entitydescr} \cup \\ & \textbf{ENVIRONMENT} \xrightarrow[m]{} \textit{Reachability-set} \cup \\ & \textbf{EXPIREDF} \xrightarrow[m]{} \textit{Is-expired} \cup \\ & \textbf{PIDSORT} \xrightarrow[m]{} \textit{Identifier}_1 \cup \\ & \textbf{NULLVALUE} \xrightarrow[m]{} \textit{Identifier}_1 \cup \\ & \textbf{TRUEVALUE} \xrightarrow[m]{} \textit{Identifier}_1 \cup \\ & \textbf{FALSEVALUE} \xrightarrow[m]{} \textit{Identifier}_1 \end{aligned}$$

上面给出了 *Entity-dict* 映射的完整的定义。它说明了怎样使用$\xrightarrow[m]{}$操作符，同时也说明了复合映射可以用域合并操作符∪来构成，即，给定域 *Entity-dict* 的一个映射：

- 我们通过应用无名树（$\textit{Identifier}_1\textit{Entityclass}$）的一个实物在映射中查找，而结果是域 *Entitydescr* 的一个实物，或
- 我们应用 *Quot* 值 **ENVIRONMENT**，而映射的结果是域 *Reachability*-**set** 的一个实物，或
- 我们应用 *Quot* 值 **EXPIREDF**，而映射的结果是域 **Is-expired** 的一个实物，或
- 我们应用 *Quot* 值 **PIDSORT**，而映射的结果是域 $\textit{Identifier}_1$的一个实物，或
- 我们应用 *Quot* 值 **NULLVALUE**，而映射的结果是域 $\textit{Identifier}_1$的一个实物，或
- 我们应用 *Quot* 值 **TRUEVALUE**，而映射的结果为域 $\textit{Identifier}_1$的一个实物，或
- 我们应用 *Quot* 值 **FALSEVALUE**，而映射的结果为域 $\textit{Identifier}_1$的一个实物。

一个值只有事先它已经被放入作为映射的一个实物，我们才能够应用这个值。这是因为这里和函数是不一样的。在函数中变量值和结果值之间的对应性是固定的，并且在把函数下定义的时候也就被确定了。

利用显式映射构造符可以直接构造映射值，这种构造符是用方括号括起来的若干对项目的值和其对应

范围值。例如，

$[1 \mapsto D,$
$2 \mapsto AA,$
$4 \mapsto BB,$
$9 \mapsto ABC,$
$5 \mapsto XYZ]$

表示域 *Intg* $\underset{m}{\rightarrow}$ *Quot* 的一个映射值。

也可以构成隐式映射，例如隐式映射

$[a \mapsto b \mid a \in N_1 \wedge a * a = b]$

等价于下面的无穷映射

$[1 \mapsto 1,$
$2 \mapsto 4,$
$3 \mapsto 9,$
$\ldots \mapsto \ldots]$

下面，对作用于映射 m 的操作符的语义作出说明，m 表示上述显式规定的第一个映射：

m(entryvalue) 从映射送回一个值，即，m(1)≡D,而 m(9)≡ABC 。

+ 把一个映射改写成另一个映射。该操作符是不可交换的，即

$m + [0 \mapsto XX, 1 \mapsto B] \equiv$
$[0 \mapsto XX, 1 \mapsto B, 2 \mapsto AA, 4 \mapsto BB, 9 \mapsto ABC, 5 \mapsto XYZ]$

而

$[0 \mapsto XX, 1 \mapsto B] + m \equiv$
$[0 \mapsto XX, 1 \mapsto D, 2 \mapsto AA, 4 \mapsto BB, 9 \mapsto ABC, 5 \mapsto XYZ]$

从一个映射中排除指定的一组项目的值，例如

$m \backslash \{1,2,3\}$ 为

$[4 \mapsto BB, 9 \mapsto ABC, 5 \mapsto XYZ]$

dom 送回一个集合，该集合恰好包含出现在给定映射中的所有项目值，例如

$\mathbf{dom}\ m \equiv \{1,2,4,5,9\}$

rng 送回一个集合，该集合恰好包含在给定映射中所含有的那些范围值，例如

$\mathbf{rng}\ m \equiv \{D, AA, BB, ABC, XYZ\}$

=, ≠ 测试两个映射的相等性与不等性 。

merge 从给定的由多个映射组成的集合中送回一个由合并该集合中所有映射而构成的映射，例如

$\{m, [0 \mapsto WE], [10 \mapsto D]\} \equiv$
$[0 \mapsto WE, 10 \mapsto D, 1 \mapsto D, 2 \mapsto AA, 4 \mapsto BB, 9 \mapsto ABC, 5 \mapsto XYZ]$

如果在集合中所包含的映射有重复的项目，则从多个可能的值中任意选择一个值。

空映射用 [] 表示（两个非常靠近的方括号）

举例

用 Meta-IV 表达式可以说明映射操作符\，＋和 **merge** 的性质，如下：

$$
\begin{aligned}
&m1 \setminus s = [a \mapsto b \mid a \in \mathbf{dom}\, m1 \setminus s \wedge m1(a) = b] \\
&m1 + m2 = [a \mapsto b \mid (a \in \mathbf{dom}\, m2 \wedge m2(a) = b) \vee (a \in \mathbf{dom}\, m1 \setminus \mathbf{dom}\, m2 \wedge m1(a) = b)] \\
&\mathbf{merge}\ m1 = (\mathbf{if}\ m1 = \{\} \\
&\qquad \mathbf{then}\ [] \\
&\qquad \mathbf{else}\ (\mathbf{let}\ element \in m1\ \mathbf{in} \\
&\qquad\quad element + \mathbf{merge}\ m1 \setminus \{element\}))
\end{aligned}
$$

5.4.8 Pid（进程标识符）域（Pid Domain）

Pid 域（对应于 SDL 中的 **Pid** 类别）用符号 Π 来构成，该域可任选地由处理器类型来限定，用以指明该域所表示的 Pid 值的种类，例如

6 *Discard-Signals* :: $\Pi(input\text{-}port)$

Discard-Signals 域（在动态语义中下定义）包含由处理器类型 *input-port* 限定的 Pid 实物。不要把 Meta-IV Pid 的值和 SDL Pid 的值（在 SDL 中是 $Ground\text{-}term_1$）相混淆，即，*SDL Pid* 值的域在动态语义中下定义为：

7 *Pid-Value* = *Value*
8 *Value* = $Ground\text{-}term_1$

当应用 **start** 语句/表达式时就创建了 Meta-IV Pid 的值。它对应于 SDL 中的创建请求动作。例如，当系统处理器创建一个带有实参 *timerf* 的定时器（*timer*）处理器实例的时候，它看上去象是：

举例

start *timer*(*timerf*)

当 **start** 构件用作为一个表达式时，它创建了一个处理器实例并送回这个实例的 Meta-IV Pid 的值（对应于 SDL 中 OFFSPRING 值）。例如，当 *sdlprocess* 处理器启动它的 *input-port* 处理器时：

start *input-port*(*selfp*, *dict*(EXPIRED))

一个 *input-port* 处理器的实例就被创建了。所得到的 Meta-IV Pid 的值被 SDL 进程用来标识 *input-port*。参数 *selfp* 和 *dict*（**EXPIRED**）则传递给所创建的实例。

通信由同步通信原语 **input** 和 **output** 来完成。在输出构件中，我们可以和一个指定的处理器实例通信，也可以和一个指定的处理器类型的未指明的实例通信。

举例

output mk-*Some-tree*(*somevalue*, *someothervalue*, ...) **to** *p*

此处 P 或为一个 Pid 值或是一个处理器类型的名字。处理器送出的值通常被封装在一有名树的实物（属于通信域）中，这样的树因而可以等同于 SDL 中的信号概念，即，可把 *Some-tree* 当作一个信号。

在输入构件中，对我们想要接收的通信实物和当该实物收到后所应采取的动作都应该加以规定。此外，我们可以规定一个名字。在实物收到之后，用此名字表示发送处理器的 Pid 值（对应于 SDL 中的 **SENDER**）或限制可能的发送者，例如

input mk-*Some-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*

在接收 *some-tree* 后，a，b 和 d 将表示由 *Some-tree* 传递的值，而对此处的 p 有三种可能的解释：

- 如果 *p* 是一个处理器类型的名字，则输入应该从那个特定处理器类型的实例来接收。
- 如果 *p* 是一个尚未下定义的名字，则该出现就是这个名字定义性的出现，而它对跟在输入子句后面的表达式或语句是可见到的。它表示发送者的 *Meta-IV Pid* 的值。
- 如果 p 是一个表达式，则它必须是类型 π，此表达式表示一个处理器实例，而此输入将是从实例接收的。

如果几个输入中的某一个可被接收，则要规定用逗号分开的多个输入构件，而输入的数目要括在括号中，例如

{input mk-*Some-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*,
input mk-*Some-other-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */***}**

在某些情况下，我们可能希望指定要么应进行输入，要么应进行输出，取决于首先通信的可能性。（这在 SDL 中是不可能的，因为 SDL 中的通信是异步的）。在这种情况下，把输出构件包括在一组通信事件中，例如

{input mk-*Some-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*,
input mk-*Some-other-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*,
output mk-*Something*(*/* expression */*, */* expression */*) **to** *pi***}**

如果通信需要重复的话，那么和输入与输出一起，常常采用**循环**（cycle）构件。例如

cycle {input mk-*Some-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*,
input mk-*Some-other-tree*(*a*, *b*, *d*) **from** *p*
⇒ */* some statements or an expression */*,
output mk-*Something*(*/* expression */*, */* expression */*) **to** *pi***}**

本例意味着在一个通信事件发生后，处理器将采取适当的动作，然后开始等待一个新事件的发生。

5.4.9 引用域（Reference Domains）

当一个 Meta-IV 变量用以下形式给出声明时

dcl v type *Intg*;

就分配了一个 Meta-IV 的存储位置，变量（v）将表示对该位置的引用。当存取该位置的内容时，就要使用 c 操作符（内容操作符），如前所述。当使用不带 c 操作符的变量时，结果是 **ref** 域的一个值，也就是对该存储单元的引用。**ref** 域利用关键字 **ref** 来规定，后面跟随一个合适的域。例如

9 *VarD* :: *Variable-identifier*$_1$ *Sort-reference-identifier*$_1$
[REVEALED] **ref** *Stg*

变量描述符包括一个对域 Stg 的引用。VarD 描述符在动态语义中下定义，并进一步在有关的注释中介绍。

5.4.10 任选域（Optional Domains）

在域定义中大量使用方括号来表示任选的意思。

举例

10 *Signal-definition*$_1$ :: *Signal-name*$_1$
Sort-reference-identifier$_1$*
[*Signal-refinement*$_1$]

表示：在树 Signal-definition$_1$的实物中，域 Signal-refinement 的实物可以出现也可以不出现。如果它不出现，则该字段将包含无类型值 **nil**。

举例

```
(let mk-Signal-definition1(name, sort, refinement) = /* some Signal-definition1 object */ in
 if refinement = nil then
   /* some actions */

 else
  (let mk-Signal-refinement1(...) = refinement in
   /* some other actions using the signal refinement */))
```

5.5 let 构件和 def 构件

如前所示，**let** 构件可用于组合与分解一些实物。**let** 构件是更多地用于每当我们想用某个名字去代表某个特定实物（往往仅是为了避免太复杂和不易读的表达式）的时候。**let** 构件中等号左边出现的名字是定义性的出现（除了域名外，因为域名必须在域定义中的某个地方下定义）。所引入的名字也可以用在等号的右边（这样就给该名字递归地下定义），以及用在 **let** 构件后面的表达式中。在下面的例子中，*name*1在/ * expression1 * /、/ * expression2 * /、/ * expression3 * /和/ * expression4 * /中是可见的（即，可以被使用），

*name*2在/ * *expression*2 * /、/ * *expression*3 * /和/ * *expression*4 * /中是可见的，*name*3在/ * *expression*3 * /和/ * *expression*4 * /中是可见的。为了限制由 **let** 所引入的名字的可见性，**let** 构件用括号括起来。在上面的例子中，一个信号的具体化构成了一个表达式，而由于使用了 **let** 构件，它用左括号开始。

有两种方法来规定一个 **let** 序例：

```
let name1 = /* expression1 */ in
let name2 = /* expression2 */ in
let name3 = /* expression3 */ in
/* expression4 */
```

或

```
let name1 = /* expression1 */,
    name2 = /* expression2 */,
    name3 = /* expression3 */ in
/* expression4 */
```

给出三个 **let** 的第一种形式通常在 FD 中，当次序是重要的时候，即当/ * *expression*2 * /使用 *name*1，且/ * *expression*3 * /使用 *name*2的情况下使用这种形式，而第二种形式用于各 **let** 独立的情况。

有几种不同形式的 **let** 构件。我们已经明白它是怎样用来分解实物的。其它有关的形式是：

```
let name ∈ setorname1 in
/* some expression using name */
let name be s.t. /* condition using name */ in
/* some expression using name */
let name ∈ setorname2 be s.t. /* condition using name */ in
/* some expression using name */
let name(parameters) = /* function body */ in
/* some expression applying name */
```

第一种形式的意思是：抽取属于由 $setorname_1$表示的集合或域的一个任意值，并用 *name* 表示该值。

第二种形式的意思是：构成一个值，即让 *name* 具有一个值使得该规定的条件对该值成立。

第三种形式是上述两种形式的组合，两种限制都适用。如果这样的值不存在，则规格说明有错误。

第四种形式的意思是：构成一个具有某些形式参数（*parameters*）和一个体的局部函数（称为 *name*）。

举例

规定3的平方根：

$$\textbf{let } r \in Real \textbf{ be s.t. } r > 0 \wedge r * r = 3 \textbf{ in}$$

举例

规定阶乘函数，n 是其形式参数：

$$\textbf{let } fact(n) = \textbf{if } n < 0 \textbf{ then error else if } n = 0 \textbf{ then } 1 \textbf{ else } n * fact(n-1) \textbf{ in}$$

当实物是由参照全局状态所构成的，则在规定其名字时（即，如果名字是借助于强制性表达式来规定的话）就使用符号 **def**，而不是符号 **let**。即，关键字 **let** 由关键字 **def** 替代，等号由冒号替代，而关键字 **in** 由

分号替代（因为 def 构件用于语句上下文中，请看第5.7节）。例如，如果我们想用一个名字表示一个创建的处理器实例的值，则写成：

```
(def pid : start input-port(somevalue);
 /* some statements using the pid value */)
```

或者，如果我们想分解一个强制性函数的结果，则我们就这样写：

```
(def mk-Some-tree(a, b) : some-imperative-function(...);
 /* some statements using a and b */)
```

还存在一个“be such that”构件的 def 变形：

```
(def r ∈ Real s.t. r > 0 ∧ r * r = c v1;
 /* some statements using r */)
```

此处我们使用 def，这是因为在 r 的求值中使用了变量（v1）。它读作：规定一个 r 的实数值使得 r 的平方等于变量 v1的内容。

应该注意，在 let 和 def 中引入的名字不是变量。它们是表示特定值的名字，而且不允许向这种名字赋新值。

5.6 量词

Meta-IV 还提供数学量词—用符号∀ 表示的全称量词、用符号∃ 表示的存在量词和由符号∃ !表示的唯一量词。这些量词可以用于量化的表达式中，如果对一个实物的指定条件（一个谓词）得到满足，则这些量化的表达式送回一个布尔值 true。

举例

```
identifiers-defined-on-system-level(p) ≜
  1  (∀mk-Identifier1(q,) ∈ p)(len q = 1)
type:  Identifier1-set → Bool
```

当且仅当对集合 p 中的所有标识符（*Identifier*$_1$），能使其限定符（q）的长度总是等于1时（第二对括号把谓词表达式括起来），这个函数才送回 true。

举例

```
one-identifier-defined-on-system-level(p) ≜
  1  (∃mk-Identifier1(q,) ∈ p)(len q = 1)
type:  Identifier1-set → Bool
```

当且仅当在集合 p 中至少存在一个标识符（*Identifier*$_1$），而其限定符（q）的长度等于1时，这个函数才送

回 true。

举例

$$exactly\text{-}one\text{-}identifier\text{-}defined\text{-}on\text{-}system\text{-}level(p) \triangleq$$

$$1 \quad (\exists!\mathbf{mk}\text{-}Identifier_1(q,) \in p)(\mathbf{len}\ q = 1)$$

$$\mathbf{type}: \quad Identifier_1\text{-}\mathbf{set} \rightarrow Bool$$

当且仅当在集合 p 中恰好存在一个标识符（$Identifier_1$），而其限定词（q）的长度等于1时，这个函数才送回 true。

换一种方式，我们也可以在谓词表达式中而不是在量词中分解标识符，例如

$$identifiers\text{-}defined\text{-}on\text{-}system\text{-}level(p) \triangleq$$

$$\begin{array}{ll} 1 & (\forall p' \in p) \\ 2 & ((\mathbf{let}\ \mathbf{mk}\text{-}Identifier_1(q,) = p'\ \mathbf{in} \\ 3 & \quad \mathbf{len}\ q = 1)) \end{array}$$

$$\mathbf{type}: \quad Identifier_1\text{-}\mathbf{set} \rightarrow Bool$$

要注意的是，撇号和短划线在 Meta-IV 名字中是合法字符。

5.7 辅助语句

- 恒等语句
  关键字 I 表示一个空语句，即，一条不做任何事情的语句。

- 未下定义的语句/表达式
  关键字 **undefined** 表示不能给出语义。

- 返回语句
  后随着一个表达式的关键字 **return** 终止了一个强制性函数的说明，而返回的结果是给定的表达式。

- 错误语句/表达式
  在 FD 中，关键字 **error** 表示一个动态的 SDL 错误。

- 赋值语句
  类似于 SDL，当向变量赋值时不能使用内容操作符（c）。

- For 和 while 语句
  和 CHILL 的概念是相同的（这是众所周知的）。要重复执行的语句用括号括起来。

- 捕捉（Trap）和离去（exit）语句/表达式
  捕捉（处理）由一个离去语句/表达式所引起的任何离去。如果给离去语句指定一个变量，当给定的

表达式与捕捉离去语句中给定的值相匹配时，才能由捕捉处理。一种捕捉离去机制的特殊形式—**tixe**构件已经被用在 *int-process-graph* 和 *int-procedure-graph* 函数中。**tixe** 构件在有关的注释中介绍。

5.8 与 CHILL 形式定义表示法的区别

- 在 CHILL 的形式定义中，预定义的域名由大写黑体字母组成（如 **BOLL**，**INTG**），而表示语义域的名字可以仅由大写字母组成。

  在 SDL 的形式定义中，所有的域名都是斜体字，第一个字母大写，并至少包含一个小写字母。

- 在 CHILL 的形式定义中，所有的实物是有限的。
  在 SDL 的形式定义中，实物可以是无限的。当某些操作符应用到这样一些实物上时，它们的语义不是良定义的，例如象基数和相等性等操作符还没有用在潜在无限的实物上。

  此外，为了表示 AS_1中的“无界的实例数”，一个特殊常数 *infinite*（无限）已经用于附件 F. 2中的 *transform-process* 里。

- 在 SDL 的形式定义中，Meta-IV 表示法已经扩展以包括基本域 *Char* 和字符串实物（参见第5. 4. 4. 3节）。

- 在附件 F. 3中的 *path*（路径）处理器中已经使用了所谓的“*output guard*”（输出卫士）。在附属于 *path* 处理器的注释和文献［4］中，描述了这一概念。

5.9 举例：用 Meta-IV 来规定精灵游戏

下面说明 Meta-IV 如何被用于规定精灵游戏的语义。有关精灵游戏的更进一步细节，参见 Z. 100 § 2. 9。

Communication *demon* → *monitor* and *monitor* → *game*

11 *Bump* :: ()

Communication *user* → *monitor*

12 *Newgame* :: ()

Communication *game* → *monitor*

13 *Gameover* :: *II*

Communication *monitor* → *game*

14 *Gameoverack* :: ()

Communication *game* → *user*

15 *Gameid* :: ()
16 *Win* :: ()
17 *Lose* :: ()
18 *Score* :: *Intg*

Communication *user* → *game*

```
19  Probe                    :: ()
20  Result                   :: ()
21  Endgame                  :: ()
```

int-demon-game() ≙

```
1  start monitor()
```

type: () ⇒ ()

monitor **processor** () ≙

```
1   (dcl userset := {} type Π-set,
2        gameset := {} type Π-set;
3    cycle (input mk-Newgame() from sender
4              ⇒ if sender ∉ c userset then
5                   (def offspring : start game(sender);
6                    gameset := c gameset ∪ {offspring};
7                    userset := c userset ∪ {sender}),
8                  else
9                   I,
10          input mk-Gameover(player) from sender
11             ⇒ (gameset := c gameset \ {sender};
12                 userset := c userset \ {player};
13                 output mk-Gameoverack() to sender),
14          input mk-Bump() from demon
15             ⇒ for all pid ∈ gameset do
16                 output mk-Bump() to pid))
```

type: () ⇒

game **processor** (*player*) ≙

```
1   (dcl count := 0 type Intg;
2    dcl even := true type Bool;
3    output mk-Gameid() to player;
4    cycle (input mk-Probe() from user
5              ⇒ if c even
6                  then (output mk-Win() to player;
7                        count := c count + 1)
8                  else (output mk-Lose() to player;
9                        count := c count − 1),
10          input mk-Result() from user
11             ⇒ output mk-Score(count) to player,
12          input mk-Endgame() from user
13             ⇒ (output mk-Gameover(player) to monitor;
14                 input mk-Gameoverack() from monitor
15                    ⇒ stop),
16          input mk-Bump() from monitor
17             ⇒ even := ¬c even))
```

type: Π ⇒ ()

ARCHIVES U.I.T. GENÈVE

参 考 文 献

[1] Dines Bjørner and Cliff B. Jones
Formal specification and software development
Prentice-Hall Publ. 1982

[2] The Formal Definition of CHILL
CCITT Manual
ITU, Geneva 1981

[3] P. Folkjær, D. Bjørner
A formal model of a generalized CSP-like language,
IFIP 8th World Computer Conference
Proceedings, North-Holland Publ. 1980

[4] C.A.R. Hoare
Communicating Sequential Processes
Prentice-Hall 1985

中国印刷 ISBN 92-61-03775-5