国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

蓝 皮 书

卷 X.5

建议Z.100的附件F.3
SDL形式定义
动态语义学

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.5

建议Z.100的附件F.3
SDL形式定义
动态语义学

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

ISBN 92-61-03795-X

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.956（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—网路管理　建议 Q.930-Q.940（第Ⅺ研究组）

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

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

卷Ⅵ.14 — 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，网间管理 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：号码簿 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他部件的结构、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.5 目录

建议 Z.100 的附件 F.3

卷 首 说 明

1　在 1989－1992 研究期内委托给各研究组的研究课题可查阅给该研究组的第 1 号文献。

2　本卷中的“主管部门”一词是电信主管部门和经认可的私营机构两者的简称。

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

目录

卷 X.5

建议 Z.100 的附件 F.3

SDL 形式定义

动态语义学

引言

形式定义的这一部分定义 SDL 的动态性质。对形式定义的整个结构的描述和对所用表示法的说明，请参阅附件 F. 1：形式定义的介绍。

一个 SDL 系统被解释为许多并发的进程。这些进程之间的通信是同步的，CSP 类的通信。在下面图中的连线表示借助于 CSP 输出的通信。系统进程创建其它进程的实例：一个视见进程的实例；一个定时器进程的实例；对于可以传输 SDL 输出的每个不同的路径要创建一个路径进程实例；对于每个实际 SDL 进程实例要创建一个“*sdl* 进程与对应输入口”的实例。在该模型中总共采用了六个不同的元进程类型：

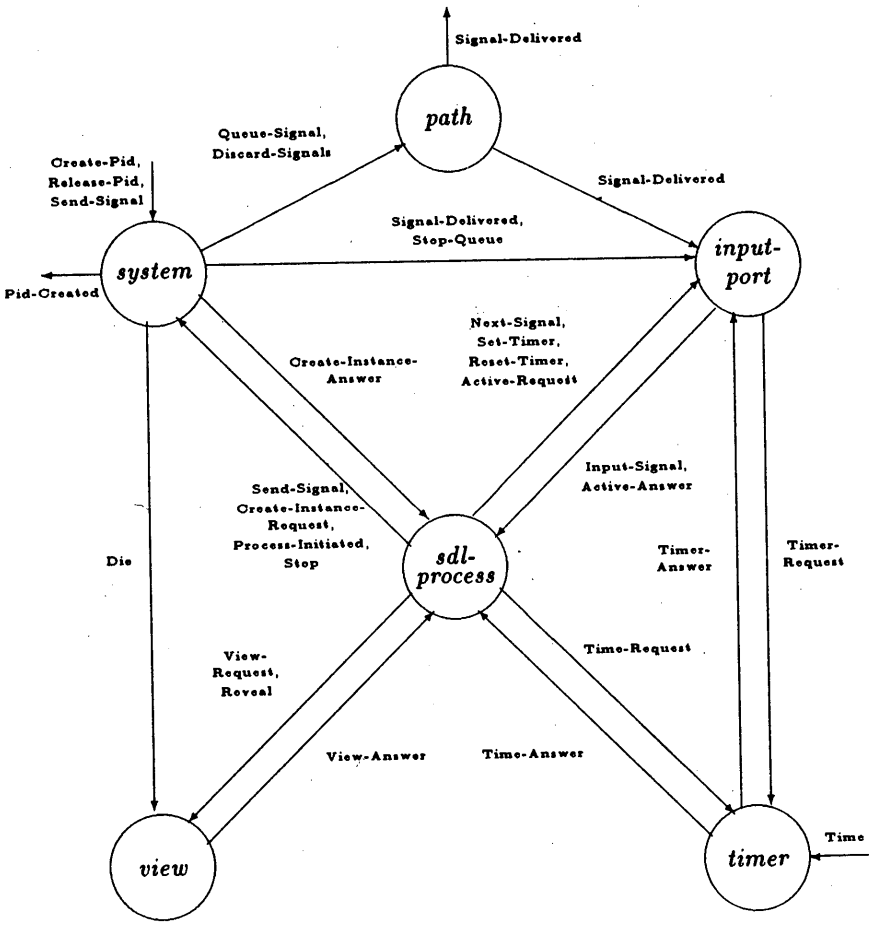

图 1

解释模型的结构

这些进程是：

1. *system*（系统进程）

   它处理信号路由和创建 *sdl-process* 进程。

2. *path*（路径进程）

   它处理信道和信号路由的不确定性延迟。请注意，由一个信号实例通过多个信号路由和多个信道所引起的所有潜在的延迟在一个路径 *path* 实例中已经叠加成为一个延迟。

3. *timer*（定时器进程）

   它跟踪当前时间并处理到时。当一个 *sdl-process* 进程正在使用 NOW 表达式时，它将从定时器 *timer* 获得时间值。

   我们假设环境每经过一个规定的时间间隔就自定时器 *timer* 发送一个时钟信号。这一机制可简称为滴答—进程（tick-process）。必须注意，滴答进程的非形式模型并不作为动态语义学的一部分，把它包括进来的原因仅仅是为了便于解释。

4. *view*（视见进程）

   它跟踪所有被透露的变量。每当一个 *sdl-process* 进程更新一个被透露变量时，它就发送一个新值给 *view* 进程。当一个进程正在使用 VIEW 表达式时，它将请求从 *view* 进程获得当前值。

5. *sdl-process*（*sdl* 进程）

   它解释一个 SDL 进程的行为。

6. *input-port*（输入口进程）

   它处理在一个 SDL 进程中信号的排队问题。对每一个 *sdl-process* 进程实例，仅仅只存在一个输入口 *input-port*。一个 *sdl-process* 进程总是在它的输入口 *input-port* 中接收信号的。

1 用于进程通信的域

1.1 *sdl-process*↔*system*

1	***Process-Initiated***	::	***Port***
2	***Port***	=	***Π(input-port)***

当一个 *sdl-process* 进程已被创建时，它就回答 *Process-Initiated*，这时它就准备好了解释它的进程图。所带有的数据就是由该进程实例所起动的输入口的 CSP 实例。

3	***Create-Instance-Request***	::	***Process-identifier*$_1$ *Value-List***
4	***Create-Instance-Answer***	::	***[Offspring-Value]***
5	***Offspring-Value***	=	***Pid-Value***
6	***Pid-Value***	=	***Value***
7	***Value***	=	***Ground-term*$_1$**

当一个进程解释该创建请求节点时，它将向系统 *system* 输出创建请求 *Greate-Instance-Request*。所带有的数据是将被起动的进程的进程标识符和实际参数表。系统 *system* 通过回送应答 *Greate-Instance-Answer* 作为响应，应答中带有被起动进程的 Pid 值。如果没有进程可被起动，则返回 **nil**。

8 *Send-Signal* :: *Signal-identifier*$_1$ *Value-List* [*Pid-Value*] *Direct-via*$_1$

当 *sdl-process* 进程的一个实例解释一个输出节点时，它将输出发送信号 *Send-Signal*。这样就传送要发送的 SDL 信号的标识符，传送附在信号上的任选值表，传送任选的目的进程实例和传送任选的要经过的信道标识符集合或信号路由标识符集合。

9 *Stop* :: ()

当 *sdl-process* 进程的一个实例解释停止结点时，它将向系统 *system* 发送 *stop*。系统 *system* 应掌握各个实例是生存的还是已死亡了。

1.2 *sdl-process*↔*input-port*

1 ***Next-Signal*** :: ***Signal-identifier*$_1$-set**
2 ***Input-Signal*** :: ***Signal-identifier*$_1$ *Value-List Sender-Value***

sdl-process 进程输出一个 *Next-Signal* 给它的输入口 *input-port*，而输入口 *input-port* 作为响应，输出 *Input-Signal*（当为非空队列时）。*Signal-identifier*$_1$-**set** 表示应留在队列中的信号（Save-set）。

3 *Set-Timer* :: *Timer-identifier*$_1$ *Timeout-value Arglist Equivalent-test*
4 *Reset-Timer* :: *Timer-identifier*$_1$ *Arglist Equivalent-test*
5 *Timeout-value* = *Value*
6 *Arglist* = *Value**
7 *Equivalent-test* :: *Ground-term*$_1$ *Ground-term*$_1$ → *Bool*

当一个 *sdl-process* 进程解释一个置定时动作时，*input-port* 就从该 *sdl-process* 进程输入置定时 *Set-Timer*，这时它用到时时间 *Timeout-value* 去起动一个定时器。一个定时器也附有一个值表，此表连同定时器标识符 *Timer-identifier*$_1$ 一起指明了该定时器实例。输入口 *input-port* 测试是否有两个请求（即 *Set-Timer*、*Reset-Timer* 或 *Active-Request*）涉及同一个定时器实例，测试的办法是比较它们的定时器标识符 *Timer-identifier* 和应用等价测试函数 *Equivalent-test* 来检查两个相关的变元表 *Arglist* 中的元素。等价测试函数 *Equivalent-test* 以两个基项 *Ground-term*$_1$ 作为变元，并返回（用“→”表示）一个布尔量 *Bool*。

8 *Active-Request* :: *Timer-identifier*$_1$ *Arglist Equivalent-test*
9 *Active-Answer* :: *Bool*

sdl-process 进程发送一个 *Active-Request* 给输入口 *input-port*，以便决定由定时器标识符 *Timer-identifier*$_1$ 标识的定时器是否是活跃的。变元表 *Arglist* 和等价测试函数 *Equivalent-test* 的解释见上。

1.3 *sdl-process↔view*

1	***Reveal***	:: ***Variable-identifier*$_1$ (*Value* | UNDEFINED) *Pid-Value***

当一个 *sdl-process* 进程更新一个被透露变量时，它将输出 *Reveal*。*Reveal* 带有被透露变量的标识符、该变量的新值和“self”（本进程）的 *Pid-value*。

2	***View-Request***	:: ***Variable-identifier*$_1$ *Pid-Value***
3	***View-Answer***	:: **(*Value* | UNDEFINED)**

当一个 *sdl-process* 进程视见一个变量时，它将输出视见请求 *View-Request*。视见请求 *view-Request* 带有被视见的变量标识符和透露它的实例的 *Pid-value*。*View* 进程将输出回答 *View-Answer* 作为响应，而回答 *View-Answer* 带有所请求的值。

1.4 *sdl-process*，*input-port↔timer*

1	***Time-Request***	:: ()
2	***Time-Answer***	:: ***Value***

当一个 *sdl-process* 进程计算 NOW 表达式的值时，它将发送时间请求 *Time-Request*。*timer* 进程将发送带有当前时间值的回答 *Time-Answer* 来作出响应。

输入口 *input-port* 连续地测试它的定时器的到期时间。为此，它需要来自定时器 *timer* 的实际时间。这种通信和 *sdl-process* 进程与定时器 *timer* 之间的通信相同。

1.5 *system↔environment*

1	***Create-Pid***	:: ***Port***
2	***Pid-Created***	:: ***Pid-Value***
3	***Release-Pid***	:: ***Pid-Value***

因为对环境所作的假设应尽可能的少，所以在环境中已经定义了创建实例的一种特殊方案。它比在一个系统中创建进程的方案要简单得多。当一个进程实例在环境中被创建时，它的 CSP 名字 *input-port* 由 *Create-Pid* 运载发送给系统。而系统将反过来向环境输出由 *Pid-Created* 运载的相关联的 SDL *Pid-Value* 作为响应。当环境中的一个进程实例停止存在时，系统将从环境接收 *Release-Pid*，它带有被停止进程的 SDL *Pid-Value*。该方案的主要目的是确认由系统来管理在环境中的 Pid 值。

1.6 *system↔view*

1	***Die***	:: ***Pid-Value***

当一个 SDL 进程已经被停止时，视见进程 *view* 就输入 *Die*，使得 *View* 进程可以从它的内部映射表（包含可以被透露的变量）中删除掉该实例的项目。

1.7 *system↔path*

1	***Queue-Signal***	:: ***Signal-identifier*$_1$ *Value-List Pid-Value Port***

一个信号的传送是通过由系统 *system* 输出一个排队信号 *Queue-Signal* 给路径进程 *path* 的实例来实现的，该 *path* 实例对应于所选择的从发送者到接收者的路由。排队信号 *Queue-Signal* 传送该信号的标识符、该信号所携带的值、表示发送者的 Pid-Value 并传送进行接收的 *input-port* 的 CSP 实例的值。

2	***Discard-Signals***	:: ***Port***

当一个 sdl-process 进程停止时，系统 *system* 要求所有的路径进程 *path* 取消指向停止的 *sdl-process* 进程的输入口 *input-port* 的信号。这是通过输出取消信号 *Discard-Signals* 来实现的。

1.8 *system*, *path↔input-port*

1	***Signal-Delivered***	:: ***Signal-identifier*$_1$ *Value-List Sender-Value***
2	***Value-List***	= [*Value*]*
3	***Sender-Value***	= ***Pid-Value***

当信号已被释放时，*path* 进程就发送该信号给 *input-port*。如果发送者和接收者是在同一功能块中，*system* 就直接发送信号给 *input-port*。

1.9 *system↔input-port*

1	***Stop-Queue***	:: ()

当一个 *sdl-process* 进程实例停止时，系统 *system* 输出 *Stop-Queue* 以便使它的输入口 *input-port* 停止。

1.10 *timer↔tick*

1	***Time***	:: ()

滴答进程没有形式地模型化。它是一个进程，每隔一定的时间间隔就向系统发出一个“滴答”脉冲。这样，它就成为定时器进程的基础。这些滴答脉冲将被当作输入流的一部分，而 SDL 系统则把输入流变换为一个输出流。

2 用于实体信息的域

实体字典 *Entity-dict* 包含有在各进程中涉及到的所有的 SDL 标识符的信息，即，每当一个进程需要一个标识符的信息时，就得利用实体字典 *Entity-dict*。最初，它是从 AS_1 推演出来的。每个进程有它自己的实体字典 *Entity-dict* 的解释。

1 *Entity-dict* =
$(Identifier_1$ **SIGNAL**$)$ $\overrightarrow{m}$ *SignalDD* ∪
$(Identifier_1$ **PROCEDURE**$)$ $\overrightarrow{m}$ *ProcedureDD* ∪
$(Identifier_1$ **TYPE**$)$ $\overrightarrow{m}$ *TypeDD* ∪
$(Identifier_1$ **SORT**$)$ $\overrightarrow{m}$ $(SyntypeDD \mid SortDD)$ ∪
$(Identifier_1$ **PROCESS**$)$ $\overrightarrow{m}$ *ProcessDD* ∪
$(Identifier_1$ **VALUE**$)$ $\overrightarrow{m}$ $(VarDD \mid OperatorDD)$ ∪
ENVIRONMENT $\overrightarrow{m}$ *Reachabilities* ∪
EXPIREDF $\overrightarrow{m}$ *Is-expired* ∪
PIDSORT $\overrightarrow{m}$ $Sort\text{-}identifier_1$ ∪
NULLVALUE $\overrightarrow{m}$ $Literal\text{-}operator\text{-}identifier_1$ ∪
TRUEVALUE $\overrightarrow{m}$ $Literal\text{-}operator\text{-}identifier_1$ ∪
FALSEVALUE $\overrightarrow{m}$ $Literal\text{-}operator\text{-}identifier_1$ ∪
SCOPEUNIT $\overrightarrow{m}$ $Qualifier_1$ ∪
PORT $\overrightarrow{m}$ $\Pi(input\text{-}port)$ ∪
SELF $\overrightarrow{m}$ $\Pi(input\text{-}port)$ ∪
PARENT $\overrightarrow{m}$ $\Pi(input\text{-}port)$

实体字典 *Entity-dict* 包括一个映射表，从标识符（$Identifier_1$）和相关的实体类组成的偶对映射到描述符。一个实体类可以是 **SIGNAL**、**PROCEDURE**、**TYPE**、**SORT**、**PROCESS** 或是 **VALUE**。

此外，它还包括怎样安排信号路由的信息，这些信号来自（或到达）系统的环境。*ENVIRONMENT* 将在下面解释。

一个描述符可以是一个信号的描述符、一个过程、一个类型、一个同义类型、一个进程、一个类别、一个变量、一个字面值或是一个运算符的描述符。请注意，不包括某些 SDL 标识符的实体（如信道和功能块）。

而且，实体字典 *Entity-dict* 还包括某些额外的实物，这些实物是基础系统和/或 sdl 进程必须知晓的。这些实物经由某些引用文 *Quot* 值被访问：

ENVIRONMENT 当在 *Entity-dict* 上应用时，结果是流向（或发源于）环境的可达集 *Reachabilities*。

EXPIREDF 当在 *Entity-dict* 上应用时，结果是由 *timer* 处理器使用的一个函数。

PIDSORT 当在 *Entity-dict* 上应用时，结果是 PiD 类别的 AS_1 标识符。

NULLVALUE 当在 *Entity-dict* 上应用时，结果是 PiD 字面值 null 的 AS_1 标识符。

TRUEVALUE 当在 *Entity-dict* 上应用时，结果是布尔字面值 true 的 AS_1 标识符。

FALSEVALUE 当在 *Entity-dict* 上应用时，结果是布尔字面值 false 的 AS_1 标识符。

SCOPEUNIT 当在 *Entity-dict* 上应用时，结果是表示当前作用域单位的限定符。

PORT 当在 *Entity-dict* 上应用时，结果是一个 sdl 进程的输入口的 Π 值。

SELF 当在 *Entity-dict* 上应用时，结果是利用该 *Entity-dict* 的 sdl 进程的 Π 值。

PARENT　　当在 *Entity-dict* 上应用时，结果是利用 *Entity-dict* 的 sdl 进程的父辈的 Π 值。

2.1 信号描述符

1 ***SignalDD*** :: ***Sort-reference-identifier*$_1$*

SignalDD 是一个信号的描述符。它包含附着于该信号的类别标识符或同义类型标识符的表。

2.2 过程描述符

1 ***ProcedureDD*** :: ***FormparmDD** **Procedure-graph*$_1$**
2 ***FormparmDD*** = ***InparmDD*** | ***InoutparmDD***
3 ***InparmDD*** :: ***Variable-identifier*$_1$**
4 ***InoutparmDD*** :: ***Variable-identifier*$_1$**

ProcedureDD 是一个过程的描述符。它包括一连串的形式参数描述符和该过程图。一个形式参数可以是一个 IN 参数，也可以是一个 IN/OUT 参数，它包含变量标识符 *Variable-identifier*$_1$。

2.3 类型描述符

1 ***TypeDD*** :: ***Sortmap Equations*$_1$**
2 ***Sortmap*** = ***Sort-identifier*$_1$ $\overrightarrow{m}$ *Term-class-set***
3 ***Term-class*** = **(*Ground-term*$_1$ | *Error-term*$_1$)-set**

TypeDD 是一个数据类型定义的描述符。它包含一个映射表（*Sortmap*），该映射表把包围该数据类型定义的作用域单位中的所有可见的类别标识符 *Sort-identifier*$_1$ 映射到该类别的等价类集合。一个等价类（*Term-class*）是一个基项的集合，也可能包含错误项。

它也包含一些等式（*Equations*$_1$），等价类就是从这些等式导出的。

2.4 类别描述符

1 ***SortDD*** :: ***Type-identifier*$_1$**
2 ***SyntypeDD*** :: ***Parent-sort-identifier*$_1$ *Range-condition*$_1$**

SortDD 和 SyntypeDD 分别是新类型和同义类型的描述符。新类型的描述符包含包围的数据类型定义的标识符，因为新类型的所有性质都保持在那个描述符中。

同义类型描述符也包含其父辈新类型的标识符和一个 AS$_1$ 范围条件。

2.5 进程描述符

1	*ProcessDD*	::	*ParameterDD* Initial Maximum Process-graph*$_1$ *Reachabilities*
2	*Reachabilities*	=	*Reachability*-**set**
3	*ParameterDD*	=	*Variable-identifier*$_1$
4	*Initial*	=	*Intg*
5	*Maximum*	=	*Intg*
6	*Reachability*	=	(*Process-identifier*$_1$ | ENVIRONMENT) *Signal-identifier*$_1$-**set** *Path*
7	*Path*	=	*Path-identifier**
8	*Path-identifier*	=	*Identifier*$_1$

ProcessDD 是一个进程的描述符。它包含参数表（*ParameterDD*）、在系统启动时创建的进程实例数（*Initial*）、所允许的最多的进程个数（*Maximum*）、进程图和可达集 *Reachabilities*。一个可达者 *Reachability* 规定了从此进程发送一个在信号标识符集 *Signal-identifier*$_1$-**set** 中的信号经过某个路径 *path* 可以达到的一个进程标识符 *Process-identifier*$_1$。路径 *path* 是用一连串的信号路由和信道标识符（*Path-identifier*）来确定的。在进程标识符 *Process-identifier*$_1$ 既为发送者又为接收者时，*Path* 为空。形式参数描述符就是该参数的变量标识符 *Variable-identifier*$_1$。

2.6 变量描述符

1	*VarDD*	::	*Variable-identifier*$_1$ *Sort-reference-identifier*$_1$ [REVEALED] [**ref** *Stg*]

VarDD 是一个变量的描述符。它包含该变量标识符、类别或同义类型的标识符、透露 **REVEALED** 属性和任选的对一个存储单元的引用。没有用于视见变量的描述符，因为视见定义 *View-definition*$_1$ 不包含（重要的）信息。每当引用一个过程时，*Entity-Dict* 就要用表示形式参数和局部声明的描述符来重写。对于 IN/OUT 参数，其描述符包含相关的实在参数和一个对存储版本的引用，在该存储中可以找到变量标识符 *Variable-identifier*$_1$ 的版本，即，因为 SDL 允许递归过程，同一个变量标识符 *Variable-identifier*$_1$ 可以有几个版本，每次递归调用都有一个版本，因而也有几个存储版本。

2.7 运算符和字面值描述符

1	*OperatorDD*	::	*Argument-list Result*
2	*Argument-list*	=	*Sort-reference-identifier*$_1$*
3	*Result*	=	*Sort-reference-identifier*$_1$

OperatorDD 是一个运算符或一个字面值的描述符。它包含变元的类别或同义类型的表和结果的类别或同义类型。

3 基础系统

3.1 系统处理器

这个处理器是对一个 SDL 描述的解释的入口点。所有其它进程都是从这个进程起动的（直接地或间接地）。它是从在附件 F. 2：静态语义中所定义的 *definition-of-SDL* 起动的。

system **processor** (as_1tree, *subset*, *auxinf*) ≙ **(3.1.1)**

```
1  (let (timeinf, terminf, expiredf, delayf) = auxinf in
2   dcl instancemap := [] type Π(sdl-process) m→
3       ((ENVIRONMENT | Process-identifier1) Pid-Value);
4   dcl queuemap := [] type Pid-Value-set m→Port;
5   dcl pidno := [] type Process-identifier1 m→N0;
6   dcl pathmap := [] type Channel-identifier1* m→Π(path);
7   dcl pidset := {} type Pid-Value-set;
8   trap exit with error in
9   (let entitydict = extract-dict(as1tree, subset, expiredf, terminf) in
10   start view();
11   start timer(timeinf);
12   start-initial-processes(entitydict);
13   pathd(delayf)(entitydict);
14   handle-inputs(entitydict)))
```

type: *System-definition*$_1$ *Block-identifier*$_1$**-set** *Auxiliary-information* ⇒

目的　　解释 SDL 系统

参数

as_1tree	系统的 AS_1 定义
subset	所选择的一致性子集
auxinf	包含下列项目（参见第 1 行）：
timeinf	由定时器处理器请求的信息。它包含一个函数，该函数在系统起动时在定时器处理器中的每个滴答脉冲瞬间更新当前的 NOW。此域在附件 F. 2 中作出定义，在定时器处理器中作进一步的描述。
terminf	是一个闭包，包含有 Pid 的类别、Pid 空字面值和布尔字面值 True 和 False 的 AS_1 标识符。
expiredf	是一个函数，如果给定的定时器已经到时，则此函数给出 true。
delayf	是一个函数，它可以随机地给出一个 Bool 值。在 *path* 处理器中用于模拟信道上的延迟。

算法

第 2 行	设 instancemap 表示一个映射表，它把 csp 处理器实例映射到进程标识符 *Process-identifier*$_1$ 或 **ENVIRONMENT** 和 *Pid* 值 *Pid-value* 的合成域。该映射表被实例化为空。它主要用于安排信号的路由和用于新实例的创建。
第 4 行	设 queuemap 表示一个映射表，把 *Pid* 值 *Pid-Value* 的等价类映射到 *sdl-processe* 的 *input-port*。该映射表被实例化为空。Queuemap 和 instancemap 的使用目的是一样的。
第 5 行	设 pidno 表示一个映射表，用于检查一个进程定义的最大实例数是否被超过了。该映射表被实例化为空。

第 6 行　　设 pathmap 表示一个映射表。它把有延迟的路径映射到 *path* 处理器的 csp 实例。当解释一个输出节点时，有延迟的路径是由一个信号实例经过的一连串信道。有必要区别可能的延迟路径，因为当顺着一个信道序列传输时仅需保证序列的次序。

第 7 行　　设 pidset 表示 *Pid-Value* 的初始空集。

第 11 行　　为了 NOW 的处理，用实在参数起动 *timer*（在 *timer* 中有进一步的解释）。

第 12 行　　起动 *sdl-processe* 进程。

第 13 行　　对于在系统中的每个通信路径，起动一个处理器实例。

第 14 行　　处理所有进一步的通信。

start-initial-processes(*entitydict*) ≙ **(3.1.2)**

```
1  (let pset = {id | (id, PROCESS) ∈ dom entitydict} in
2   for all p ∈ pset do
3   (let mk-ProcessDD(, no, , , ) = entitydict((p, PROCESS)) in
4    (for i = 1 to no do
5     handle-create-instance-request(p, nil, nil)(entitydict))))
```

type: *Entity-dict* ⇒

目的　　起动 sdl 进程

算法

第 3 行　　设 *no* 表示要被起动的进程实例数。

第 4 行　　起动所请求的实例数。

pathd(*delayf*)(*entitydict*) ≙ **(3.1.3)**

```
1  (let rs = entitydict(ENVIRONMENT) in
2   for all reach ∈ rs do
3   (let (, , p) = reach in
4    let p' = ⟨p[i] | 1 ≤ i ≤ len p − 1⟩ in
5    (if p' ∉ dom pathmap
6       then (def cspp : start path(delayf);
7             pathmap := c pathmap + [p' ↦ cspp])
8       else I));
9   for all (pd, PROCESS) ∈ dom entitydict do
10  (let mk-ProcessDD(, , , , rs) = entitydict((pd, PROCESS)) in
11   for all reach ∈ rs do
12   (let (d, , p) = reach in
13    let p' = ⟨p[i] | 2 ≤ i ≤ (d = ENVIRONMENT
14                                → len p,
15                              T → len p − 1)⟩ in
16    if len p' > 0 ∧ p ∉ dom pathmap then
17      (def cspp : start path(delayf);
18       pathmap := c pathmap + [p' ↦ cspp])
19     else
20      I)))
```

type: (() ⇒ *Bool*) → *Entity-dict* ⇒

目的　　对系统中的每一个由进程标识符 *Process-identifier*$_1$ 和路径组成的偶对，起动一个 *path* 处理器实例。更新一个从路径到 csp 实例的映射表。

参数

delayf 随机地给出一个布尔值的函数，在 *path* 处理器中用于模拟信道的延迟

算法

第 1 行 设 *rs* 表示环境的（在其中的进程的）可达性集合 *reachability-set*。这个信息是基于从环境导向系统的信道的，并且是从实体字典 *entitydict* 中抽取的。

第 2—8 行 对集合中的每个可达者起动一个处理器实例。*pcsp* 表示被起动的处理器的 csp 实例。

第 4 行 设 p' 表示引起延迟的路径（即，排除最后一项，此项是一个信号路由）。

第 7 行 相应地更新 *pathmap*。

第 9—18 行 在系统中对始发进程（originating processes）重复这个处理办法。

第 13 行 定义路径的延迟部分开始于第二个元素（因为第一个元素是一个信号路由），并且，如果是在系统内部，则结束于最后第二个元素（因为那时最后一个元素是一个信号路由）。

第 16 行 如果余下的路径是非空的（引起延迟的），仅起动一个处理器。

handle-inputs(*entitydict*) ≙ **(3.1.4)**

```
1  (cycle {input mk-Send-Signal(si, vl, r, p) from se
2            ⇒ handle-send-signal(si, vl, r, p, se)(entitydict),
3          input mk-Create-Instance-Request(prid, vl) from se
4            ⇒ handle-create-instance-request(prid, vl, se)(entitydict),
5          input mk-Stop() from se
6            ⇒ handle-stop(se),
7          input mk-Create-Pid(port) from se
8            ⇒ handle-create-from-environment(port, se)(entitydict),
9          input mk-Release-Pid(p) from se
10           ⇒ handle-stops-in-environment(p, se)})
```

type: *Entity-dict* ⇒

目的 在初始化后处理系统 *system* 的所有通信

算法

第 1 行 开始一个无限的循环。在每次循环中，所提到的多个输入中的一个将被详细说明（基于一个非确定的基础）。对每个输入的处理将在一个专门的处理函数中介绍。

handle-stops-in-environment(*p*, *se*) ≙ **(3.1.5)**

```
1  (def class s.t. p ∈ class ∧ class ∈ dom c queuemap;
2   def q : c queuemap(class);
3   instancemap := c instancemap \ {se};
4   queuemap := c queuemap \ {class};
5   discard-signals-to-port(q))
```

type: *Pid-Value Π* ⇒

目的　　通过更新在系统中的映射表来处理环境中“进程”的停止

参数

p　　要停止的“进程”的 Pid-Value

se　　“SENDER”的 Csp 实例

算法

第 3—4 行　　从包含活着的实例的映射表中消除该“进程”和它的“输入口”。

第 5 行　　处理消除属于环境中停止进程的信号，这些信号是等候在通信路径上的。

***handle-create-from-environment*(*port*, *se*)(*entitydict*)** ≙ **(3.1.6)**

```
1  (def (pid, pidclass) : getpid(entitydict);
2   instancemap := c instancemap + [se ↦ (ENVIRONMENT, pid)];
3   queuemap := c queuemap + [pidclass ↦ port];
4   output mk-Pid-Created(pid) to se)
```

type:　*Π Π → Entity-dict* ⇒

目的　　处理环境中 Pid-Value 的创建。更新系统中的映射表，并把 Pid-Value 返回给环境。通信的方式并不完全象系统中处理创建节点 CREATE-node 的方式。不管怎样，我们不能假设环境含有创建节点 CREATE-node（!）。总的概念是对环境应尽可能少做一些假设，但仍然具有一个协调一致的模型。

参数

port　　是“发送者”输入口的 Csp 实例。假定环境包含一个输入口，因为这是实现异步通信的方法。

se　　“发送者”的 csp 实例

算法

第 2—3 行　　用与环境通信的“进程”来更新包含活着的实例的映射表。

第 4 行　　把 *Pid-Value* 返回给环境。

handle-send-signal(*si*, *vl*, *r*, *p*, *se*)(*entitydict*) ≙ **(3.1.7)**

```
 1  (def (sid, sp) : c instancemap(se);
 2   (let re = if is-Identifier1(sid)
 3                then s-Reachabilities(entitydict((sid, PROCESS)))
 4                else entitydict(ENVIRONMENT) in
 5    let re' = {(, s', ) ∈ re | si ∈ s'} in
 6    let re'' = if p = {} then re' else {(, ,p') ∈ re' | p ∩ elems p' ≠ {}} in
 7    def rp : (r ≠ nil
 8                → {(rident, r) ∈ rng c instancemap | (rident, , ) ∈ re''},
 9              T → {(rident, ) ∈ rng c instancemap | (rident, , ) ∈ re''});
10    (card (rp) = 0
11       → exit ("§2.7.4: 没有发现接收者"),
12     card (rp) > 1
13       → exit ("§2.7.4: 发现了多个接收者"),
14     T → (let {(rident, ri)} = rp in
15          let (rident', , path) ∈ re'' be s.t. rident' = rident in
16          (def class s.t. ri ∈ class ∧ class ∈ dom queuemap;
17           def rcsp : c queuemap(class);
18           (let reduced-path = delaying-path(path, sid, rident) in
19            if reduced-path = ⟨⟩
20              then output mk-Signal-Delivered(si, vl, sp) to rcsp
21              else (def path' : c pathmap(reduced-path);
22                    output mk-Queue-Signal(si, vl, sp, rcsp) to path')))))))
```

type: *Signal-identifier*$_1$ *Value-List* [*Pid-Value*] *Direct-via*$_1$
Π(*sdl-process*) → *Entity-dict* ⇒

目的　　安排信号的路由

参数

si	被发送的信号
vl	信号带有的任选的值表
r	表示接收者（来自 TO 子句）的任选的 *Pid-Value*
p	任选的路径集合（得自 VIA 子句）
se	进行发送的 *sdl-process* 进程的 Csp 实例

算法

第 1 行	设 *sid* 和 *sp* 表示发送者的进程标识符 *process-identifier*$_1$ 和 *Pid-Value*。
第 2 行	测试信号是从环境（第 4 行）发送的还是从系统中的一个进程（第 3 行）发送的。在这两种情况中，*re* 都表示发送者的可达性集合 *Reachability*-set。余下的功能依次限制发送者的可达集（直到第九行）。
第 5 行	限制可以传递实际信号 *si* 的那些可达集。
第 6—6 行	基于 VIA 子句中给出的路径 *p* 来限制可达集。
第 6 行	如果缺少 VIA 子句，则没有限制。
第 8 行	检查来自环境的路径。
第 6 行	限制可达性集合为下述这些成员，这些成员在它们路径中提到了来自于 VIA 子句的 *p* 的一个成员。
第 7 行	设 *rp* 表示可能的接收者的集合。*rp* 的成员是偶对（*Process-identifier*$_1$，*Pid-Value*）。
第 8 行	处理给出了 TO 子句的情况。此偶对必须表示一个活的实例，其中，*rident* 是在一个可达集中。
第 9 行	处理没有 TO 子句的情况。在这种情况下，偶对中的 *Pid-Value* 未被指定。

第 10—14 行　测试所发现的接收者的个数。

第 11 行　定义无（可达的和活的）接收者的错误。

第 13 行　定义多于一个接收者的错误（OUTPUT 节点的不确定性）。

第 14 行　指示成功：*rp* 含有一个且仅含有一个成员。

第 15 行　选择导向唯一接收者的一条路径。如果指向同一进程的多个子信道可以带有相同的信号，这一选择是不确定的。

第 16 行　设 *rcsp* 表示处于接收状态的 *sdl-process* 进程的输入口的 csp 实例。

第 18 行　设 *reduced-path* 表示引起延迟的 *Path* 的部分（一些信道）。

第 19 行　如果信号不经过信道传递（在同一功能块内），则信号输出到接收者的输入口 *input-port* 处理器。

第 21 行　设 *path'* 表示相应的 *path* 处理器的 csp 实例。

第 22 行　把信号输出给所选择的 *path* 处理器。

***delaying-path*(*path*, *sid*, *rid*)** ≙ **(3.1.8)**

```
1  (len path ≤ 1
2     → ⟨⟩,
3   sid = ENVIRONMENT
4     → ⟨path[i] | 1 ≤ i ≤ len path − 1⟩,
5   rid = ENVIRONMENT
6     → tl path,
7   T → ⟨path[i] | 2 ≤ i ≤ len path − 1⟩)
```

type: *Path* (ENVIRONMENT | *Process-identifier*$_1$)
(ENVIRONMENT | *Process-identifier*$_1$) → *Path*

目的　把通信路径缩减为有延迟的路径 *delaying* path

参数

path　从发送者到接收者的一条完整的路径

sid　发送者的标识符

rid　接收者的标识符

结果　有延迟的路径

算法

第 1 行　如果路径为空或仅由一个信号路由标识符组成，则返回未修改的这条路径。

第 3 行　如果信号来自环境，则消去路径 *Path* 结尾处的信号路由。

第 5 行　如果目的地是环境，则消去路径 *Path* 开始处的信号路由标识符。

第 7 行　如果信号从一个功能块发送到另一个功能块，则消去 *Path* 开始处和结束处的信号路由标识符。

handle-create-instance-request(*prid*, *vl*, *se*)(*entitydict*) ≙ (3.1.9)

```
 1  (if prid ∉ dom c pidno
 2     then pidno := c pidno + [prid ↦ 0]
 3     else I;
 4   (let mk-ProcessDD(il, , maximum, , ) = entitydict((prid, PROCESS)) in
 5    let vl' = if vl = nil then ⟨nil | 1 ≤ i ≤ len il⟩ else vl in
 6    def parent : if se = nil then nil else s-Pid-Value(c instancemap(se));
 7    def exceed : (maximum = c pidno(prid));
 8    def (newpid, pidclass) : getpid(entitydict);
 9    if ¬exceed then
10      (def csppid : start sdl-process(parent, newpid, vl', prid)(entitydict);
11       (input mk-Process-Initiated(qcsppid) from csppid
12            ⇒  (instancemap := c instancemap + [csppid ↦ (prid, newpid)];
13                queuemap := c queuemap + [pidclass ↦ qcsppid];
14                pidno := c pidno + [prid ↦ c pidno(prid) + 1])))
15     else
16      I;
17    if se ≠ nil
18      then output mk-Create-Instance-Answer(if exceed then nil else newpid) to se
19      else I))

type:  Process-identifier1 [Value-List] [Π(sdl-process)] →
       Entity-dict ⇒
```

目的　　处理 *sdl-processes* 进程的创建

参数

prid　　要被起动的进程的进程标识符 *Process-identifier*$_1$

vl　　任选的实在参数表（=nil，如果在系统初始化期间被调用的话）

se　　任选的父辈（=nil，如果在系统初始化期间被调用的话）

算法

第 1 行　　把带有 0 个实例的一个 *prid* 的映射表初始化为 0。

第 6 行　　设 *parent* 表示将被传送给新实例的父辈的值。

第 8 行　　产生一个唯一的 *Pid-Value*。

第 7 行　　对一个进程定义的最大实例数是否被超过进行测试。

第 10 行　　起动 *sdl-process* 实例本身。

第 11 行　　等待被起动的 *sdl-process* 进程传来对初始化的确认。

第 12 行　　把此 *sdl-process* 进程增加到 *sdl-processes* 映射表中。

第 13 行　　把它的输入口 *input-port* 增加到输入口 *input-port* 映射表中。

第 14 行　　更新进程定义的当前实例数。

第 18 行　　如果创建是由一个创建节点 *Create-node*$_1$ 引起的，则用新进程的 *Pid-Value* 给 “SENDER” 发回一个回答。如果超过了最大实例数，则返回 nil。

getpid(*entitydict*) ≙ (3.1.10)

```
1  (let pidsortid = entitydict(PIDSORT) in
2   let mk-SortDD(tid) = entitydict((pidsortid, SORT)) in
3   let mk-TypeDD(sortmap, ) = entitydict((tid, TYPE)) in
4   let classes = sortmap(pidsortid) in
5   let nullterm = entitydict(NULLVALUE) in
6   def class s.t. class ∈ classes ∧ (nullterm ∉ class) ∧ ¬(∃term ∈ class)(term ∈ c pidset);
7   let pid ∈ class in
8   pidset := c pidset ∪ class;
9   return (pid, class))
```

type: *Entity-dict* ⇒ *Pid-Value Pid-Value*-set

目的 抽取一个还未用过的 *Pid-Value*。在 Z. 100 中为 Pid 类别定义的唯一运算符 Unique! 保证存在有无限多个 *Pid-Value*。即，Pid 类别的值是 null、unique!（null）、unique!（unique!（null））等等。Pid 项（值）的集合放在 *entitydict* 中。

注意，本模型假设，*system* 处理器为环境也提供唯一的 *Pid-Values*。否则，很难想象如何用 *Pid-Values* 来访问环境中的进程。

结果 一个未使用的 *Pid-value* 和它所属于的等价类

算法

第 1 行 从实体字典 *entitydict* 抽取 PID 类别的标识符 Identifier_1。

第 2 行 抽取在系统层上定义的类型标识符。

第 3 行 抽取 sortmap（类别映射表），它包含在系统层上定义的类别的等价类。

第 4 行 抽取 Pid 类别的等价类。请注意，对于 Pid 类别，每个等价类正好含有一个基项。

第 5 行 抽取 NULL 项的 AS_1 表示形式。

第 6 行 取一个等价类，它没有在 pidset 中表示，并且它不同于 NULL 项。

第 7 行 在这个等价类中取出一个基项。

第 8 行 把这个新值加入到 *Pid-Value* 的集合中。

handle-stop(*se*) ≙ (3.1.11)

```
1  (def (prid, p) : c instancemap(se);
2   def class s.t. p ∈ class ∧ class ∈ dom queuemap;
3   def q : c queuemap(class);
4   instancemap := c instancemap \ {se};
5   queuemap := c queuemap \ {class};
6   pidno := c pidno + [prid ↦ c pidno(prid) − 1];
7   discard-signals-to-port(q);
8   output mk-Stop-Queue() to q;
9   output mk-Die(p) to view)
```

type: Π(*sdl-process*) ⇒

目的 处理停止节点 $\textit{STOP-Node}_1$

参数

se 要被停止的 *sdl-process* 进程的 *Csp* 实例

算法

第 4 行　　从活实例映射表中删去进程。
第 5 行　　从输入口映射表中删去相应的输入口 *input-port*。
第 6 行　　通过删去被停止的进程来更新 *prid* 的当前实例数。
第 7 行　　丢弃等候在路径上准备发往被停止进程的信号。
第 8 行　　请求此输入口 *input-port* 停止。
第 9 行　　请求视见 *view* 去更新被透露变量的映射表。

discard-signals-to-port(q) $\triangleq$ **(3.1.12)**

```
1  (for all r ∈ rng c pathmap do
2   output mk-Discard-Signals(q) to r)
```

type: *Port* $\Rightarrow$

目的　　向所有的 *path* 实例输出信息，告诉它们删除掉那些等候发送给某一个输入口 *input-port* 的信号实例。

参数

q　　该输入口 *input-port* 的 Csp 实例

3.2 视见处理器

***view* processor** () $\triangleq$ **(3.2.1)**

```
1  (dcl viewmap := [] type (Pid-Value Variable-identifier1) m→( Value | UNDEFINED);
2   trap exit with error in
3   (cycle {input mk-Reveal(id, value, pid) from sdl-process
4              ⇒  viewmap := c viewmap + [(pid, id) ↦ value],
5           input mk-View-Request(id, revealpid) from viewpid
6              ⇒  (let entry = (revealpid, id) in
7                   if entry ∈ dom c viewmap
8                     then output mk-View-Answer(c viewmap(entry)) to viewpid
9                     else exit ("5.5.4.4: 透露进程未激活")),
10          input mk-Die(pid) from system
11             ⇒  (for all (pid, id) ∈ dom c viewmap do
12                  viewmap := c viewmap \ {(pid, id)})}))
```

type: () $\Rightarrow$

目的　　解释视见 VIEW 和透露 REVEAL 的概念

算法

第 1 行　　设 viewmap 表示一个映射表，从进程标识符值 *Pid-Value* 和变量标识符 *Variable-identifier*$_1$ 的一个偶对映射到一个被透露的值 *Value*。
第 3 行　　处理透露 *Reveal* 输入。
第 4 行　　用新的项更新映射表。
第 5 行　　处理一个来自 *sdl-process* 进程的 VIEW。
第 8 行　　把值返回给 *sdl-process* 进程。
第 9 行　　定义一个变量没有被透露的错误。
第 10 行　　处理 *sdl-process* 进程被停止的通知。
第 11 行　　从映射表中删去被停止进程的所有被透露变量。

path **processor** (*delayf*) $\triangleq$ **(3.3.1)**

```
1  (dcl pqueue := ⟨⟩ type (Signal-identifier1 Value-List Pid-Value Port)*;
2   cycle {input mk-Queue-Signal(si, vl, sp, rcsp) from system
3            ⇒  (pqueue := c pqueue ↷ ⟨(si, vl, sp, rcsp)⟩),
4          input mk-Discard-Signals(q) from system
5            ⇒  (pqueue := ⟨c pqueue[i] | 1 ≤ i ≤ len c pqueue ∧
6                                        (def(, , , r) : c pqueue[i];
7                                         return r ≠ q)⟩),
8          (delayf() ∧ c pqueue ≠ ⟨⟩)
9          (output mk-Signal-Delivered(s-Signal-identifier1(hd c pqueue),
10                                     s-Value-List(hd c pqueue),
11                                     s-Pid-Value(hd c pqueue)) to s-Port(hd c pqueue))
12         ⇒ (pqueue := tl c pqueue)})
```

type: $((\,) \Rightarrow Bool) \Rightarrow$

目的 解释一条路径中的可能的延迟。对于进程描述符 *ProcessDD* 中可达性集合 *Reachability*-set 内的路径的每一个值都存在一个实例。

参数

delayf 是一个函数，它随机地给出一个 Bool 值，用于模拟在多个信道上的延迟。

算法

第 3 行 把一个信号插入到路径的队列中。

第 4 行 处理一些信号的消除，这些信号要传往一个特定的输入口 *input-port*，*q*。用于此输入口 *input-port* 的 *sdl-process* 进程停止时。

第 5 行 除了传往 *q* 的项以外，设新的 pqueue 等于原先的。

第 8 行 这个子句模拟在路径上不确定的延迟。输出 **output** 由一个谓词来作为条件：如果 *delayf* 产生 true 且 pqueue 为非空，才可以进行输出。具体语法是：

（<**谓词**>）（<**通信事件**>）

⇒<**语句**>

借助强制性函数 *delayf* 表示不确定性，它的定义是在本形式定义的范围之外。如果谓词成立，则把该信号输出给输入口 *input-port* 的实例。

第 12 行 把输出信号从队列中删除。

3.4 输入口处理器

本处理器实现 *sdl-processes* 进程和定时器的无界缓冲区。无界缓冲区在处理器中以可变队列的形式出现。

***input-port* processor (*ppid*, *expiredf*)** ≙ **(3.4.1)**

```
1  (dcl queue := ⟨⟩ type (Signal-identifier1 Value-List Pid-Value)*;
2   dcl waiting := false type Bool;
3   dcl pendingset := {} type Signal-identifier1-set;
4   dcl timers := [] type (Timer-identifier1 Arglist) m→(Π(sdl-process) Value Equivalent-test);
5   cycle {input mk-Signal-Delivered(sid, vl, se) from p
6              ⇒  handle-queue-insert(sid, vl, se, p),
7          input mk-Next-Signal(saveset) from p
8              ⇒  handle-queue-extract(saveset, ⟨⟩, c queue, p),
9          input mk-Stop-Queue() from p
10             ⇒  stop,
11         input mk-Set-Timer(tid, v, al, et) from p
12             ⇒  handle-set-timer(tid, v, al, et, p),
13         input mk-Reset-Timer(tid, al, et) from p
14             ⇒  handle-reset-timer(tid, al, et),
15         input mk-Active-Request(tid, al, et) from p
16             ⇒  handle-active-request(tid, al, et, p),
17         (output mk-Time-Request() to timer;
18          handle-time-request(ppid, expiredf))})
```

type: *Pid-Value Is-expired* ⇒

目的 解释 *sdl-process* 进程的输入口。对 *sdl-process* 进程的每个实例都存在一个输入口实例。

参数

pcsp 被服务的 *sdl-process* 进程的 PId 值

expiredf 是一个函数。如果一个给定定时器已经到时，此函数给出 True。

算法

第 1 行 设队列 queue 表示 *sdl-process* 进程的无界缓冲区。队列的每个元素包含一个信号标识符 *Signal-identifier*$_1$、一个可能为空的值 *Value* 表和一个表示“发送者”(SENDER) 的 *Pid-Value*。

第 2 行 设 waiting 表示 *sdl-process* 进程是否在等待回答。在 *sdl-process* 进程发出请求 *Next-signal* 之后，不能立即得到回答，因为队列可能是空的，或者因为在队列中的所有信号都是保存集的参数的成员。在请求挂起的情况下，pendingset（参见下面）保持挂起请求的保存集 saveset。

第 3 行 设 pendingset 表示和一个挂起请求相关联的 saveset，该请求来自 *sdl-process* 进程，由 waiting 来指明。

第 4 行 设 timers 表示多个活跃定时器的一个映射表。该映射表中的 Π (*sdl-process*) 表示设置定时器的 *sdl-process* 进程。该映射表中的值 *Value* 表示到时的时间，值在到时之后成为 nil（即，该时间仍在队列中）。等价测试 *Equivalence-test* 用于比较 *Arglist* 的值（例如，比较来自定时器映射表的一个元素和在函数 *handle-queue-extract* 的队列里的一个信号）。值 *Value* 保存到时时间。当一个定时器作为一个信号返回给 *sdl-process* 进程时，就从该映射表中删去此定时器。

第 5 行 是输入口 *input-port* 的主循环的入口。

第 7 行　　注意：这个输入总是不能立即得到回答的。引入变量 waiting 和 pendingset 的原因是因为我们采用了 SAVE 构件。如果采用一个纯队列结构，则可采用一个输入条件，使得在空队列情况下排除 *Next-Signal* 的通信。

第 17 行　　在这个方案中包括一个输出。这就是反复地向定时器 *timer* 请求实际时间。

***handle-queue-insert*(*sid*, *vl*, *se*, *pcsp*)** $\triangleq$ **(3.4.2)**

```
1  (queue := c queue ⌢ ⟨(sid, vl, se)⟩;
2   if ¬waiting then
3      handle-queue-extract(c pendingset, ⟨⟩, c queue, pcsp)
4   else
5      I)
```

type : ***Signal-identifier*$_1$ *Value-List Pid-Value* Π(*sdl-process*)** $\Rightarrow$

目的　　在队列中插入一个信号。

参数

sid	要被插入的信号
vl	它的任选的值表
se	发送者
pcsp	被服务的 *sdl-process* 的 CSP 实例

算法

第 1 行　　把信号拼接到队列中。

第 2—3 行　　测试一个 *Next-Signal* 是否被挂起的，并且如果是的话，则从队列中抽出一个元素。这可能导致把一个输入信号 *Input-Signal* 传给 *sdl-process* 进程。

handle-queue-extract(*saveset*, *qf*, *qa*, *pcsp*) $\triangleq$ **(3.4.3)**

```
1  (if qa ≠ ⟨⟩ then
2     (let s = hd qa in
3      let (sid, vl, se) = s in
4      (if sid ∈ saveset
5         then handle-queue-extract(saveset, qf ⌢ ⟨s⟩, tl qa, pcsp)
6         else (output mk-Input-Signal(sid, vl, se) to pcsp;
7               queue := qf ⌢ tl qa;
8               waiting := false;
9               if (sid, ) ∈ dom c timers then
10                if (∃a, et)((sid, a) ∈ dom c timers ∧
11                          (, , et) = c timers(sid, a) ∧
12                          same-argument-values(a, vl, et)) then
13                  (def (a, et) s.t. (sid, a) ∈ dom c timers ∧
14                       (, , et) = c timers(sid, a) ∧ same-argument-values(a, vl, et);
15                   timers := c timers \ {(sid, a)})
16                else
17                 I
18               else
19                I)))
20   else
21    (pendingset := saveset;
22     waiting := true))
```

type: *Signal-identifier*$_1$-**set**
(*Signal-identifier*$_1$ *Value-List Pid-Value*)*
(*Signal-identifier*$_1$ *Value-List Pid-Value*)* Π(*sdl-process*) $\Rightarrow$

目的 从队列中抽取一个元素，并把它发送给 *sdl-process* 进程，如果 *sdl-process* 进程已准备好接收输入的话。

参数

saveset	在现在这种情况下，在队列中不能被抽取的信号集
qf	已经检查过的队列部分
qa	尚待检查的队列部分
pcsp	被服务的 *sdl-process* 进程的 CSP 实例

算法

第 1 行	如果 *qa* 为空，则停止抽取，抽取不成功。
第 2 行	否则取出 *qa* 的第一个元素并且
第 3 行	把队列分解为信号标识符 *Signal-identifier*$_1$、一个值 *Value* 表和“发送者”(SENDER)。
第 5 行	如果信号处于保存集 *saveset* 中，则把它拼接到已被检查过的队列，并且继续搜索 *qa* 的余下部分。
第 6 行	如果 *si* 不在保存集 *saveset* 中，则把下一信号 *Next-Signal* 输出给 *sdl-process* 进程。通过拼接 *qf* 和 *qa* 的余下部分来更新队列。为请求没有被挂起设置标志，然后
第 9 行	如果所抽取的信号是一个定时器，更新定时器的映射表
第 13 行	将要被消除的定时器应该有和信号相同的标识符，*si* 和与 *et* 的一个比较应该得出结论：来自该定时器的变元表和来自定时器映射表的一个元素是相同的。
第 21 行	在不成功的情况下，为用真实的 *saveset* 挂起请求设置标志。

same-argument-values(a, vl, et) $\triangleq$ **(3.4.4)**

```
1  (len a = len vl ∧
2   (∀i ∈ ind a)(et(a[i], vl[i])))
```

type: *Arglist Arglist Equivalent-test* → *Bool*

目的 测试两个 $Term_1$ 表是否等价，等价性由函数 *et* 定义

参数

a	要检查的一个表
vl	另一张表
et	等价测试函数

算法

第 1 行	两张表的长度应是相同的。
第 2 行	对每个标引号测试应该成功。

handle-set-timer(tid, v, al, et, p) $\triangleq$ **(3.4.5)**

```
1  (handle-reset-timer(tid, al, et);
2   timers := c timers + [(tid, al) ↦ (p, v, et)])
```

type: *Timer-identifier*$_1$ *Value Arglist Equivalent-test* Π(*sdl-process*) ⇒

目的 更新定时器映射表，以设置一个定时器

参数

tid	定时器的标识符
v	到时的时间
al	定时器的变元表
et	相应的等价测试函数，该函数可应用于变元表中的每个成员
p	设置定时器的 *sdl-process* 进程

算法

第 1 行	复位一个可能存在的，具有相同标识符和变元表的定时器。
第 2 行	更新定时器映射表。

handle-reset-timer(tid, al, et) $\triangleq$ **(3.4.6)**

```
1  (for all (t, a) ∈ dom c timers do
2   (if (∃a)((tid, a) ∈ dom c timers ∧
3            same-argument-values(a, al, et)) then
4      (def (, e, ) : c timers(tid, al);
5       (timers := c timers \ {(t, a)};
6        if c e = nil
7          then (handle-remove-timer-from-queue(tid, al, et, ⟨⟩, c queue))
8          else I))
9     else
10     I))
```

type: *Timer-identifier*$_1$ *Arglist Equivalent-test* ⇒

目的　　通过更新定时器映射表和队列使一个定时器复位

参数

tid　　定时器标识符

al　　定时器的变元表

et　　相应的等价测试函数，该函数可作用于变元表中的每个成员

算法

第 2 行　　选择一个适当的定时器，其变元表使得（*tid*，*a*）是在定时器映射表的域中，并且通过应用等价测试函数 *et* 证明 *a* 与 *al* 匹配。

第 5 行　　从定时器映射表中删去（*tid*，*a*）。

第 7 行　　从队列中消除 *tid*，如果它已被放在队列中的话（在定时器映射表的范围的 *Value* 字段中作了标记）。

***handle-remove-timer-from-queue*(*sid*, *al*, *et*, *qf*, *qa*)** $\triangleq$ **(3.4.7)**

```
1  (let (si, vl, ) = hd qa in
2   if si = sid ∧ same-argument-values(vl, al, et)
3     then queue := qf ⌢ tl qa
4     else handle-remove-timer-from-queue(sid, al, et, qf ⌢ ⟨hd qa⟩, tl qa))
```

type: *Signal-identifier*$_1$ *Arglist Equivalent-test*
(*Signal-identifier*$_1$ [*Value**] *Pid-Value*)*
(*Signal-identifier*$_1$ [*Value**] *Pid-Value*)* $\Rightarrow$

目的　　从队列中消去一个元素

参数

sid　　要被消去的信号

al　　定时器的变元表

et　　相应的等价测试函数，该函数可应用于变元表中的每个成员

qf　　已检验过的队列的部分

qa　　尚待检验的队列部分

算法

第 1 行　　设 *si* 表示 *qa* 的第一个元素的信号标识符 *Signal-identifier*$_1$。

第 3 行　　如果 *si* 是将被消除的信号，则由拼接 *qf* 和 *qa* 的余下部分来更新队列。否则

第 4 行　　继续搜索 *qa* 的余下部分，请注意，在函数的最外层调用时，*sid* 总是出现在队列中，因此终止递归的测试是不需要的！

***handle-active-request*(*tid*, *al*, *et*, *pcsp*)** $\triangleq$ **(3.4.8)**

```
1  (def stat : (∃a)((tid, a) ∈ dom c timers ∧
2                   same-argument-values(al, a, et));
3   output mk-Active-Answer(stat) to pcsp)
```

type: *Timer-identifier*$_1$ *Arglist Equivalent-test* Π(*sdl-process*) $\Rightarrow$

目的　　根据定时器映射表提供回答给 ACTIVE

参数

tid	定时器的标识符
al	定时器的变元表
et	相应的等价测试函数，该函数可应用于变元表的每个成员
pcsp	被服务的 *sdl-process* 进程的 CSP 实例

算法

第 1 行	如果所指定的定时器是在定时器映射表的域中，设 *stat* 为 true，否则 *stat* 为 false。
第 3 行	把这个值作为向 *sdl-process* 进程输出的参数。

***handle-time-request*(*ppid*, *expiredf*)** $\triangleq$ **(3.4.9)**

```
1  (input mk-Time-Answer(t) from timer
2      ⇒  (for all (tid, al) ∈ dom c timers do
3           (def (p, expt, et) : c timers(tid, al);
4            if expt ≠ nil ∧ expiredf(expt, t) then
5              (timers := c timers + [(tid, al) ↦ (p, nil, et)];
6               handle-queue-insert(tid, al, ppid, p))
7            else
8             I)))
```

type: *Value Is-expired* ⇒

目的 把所有被设置的定时器与实际时间作比较

参数

ppid	被服务的 *sdl-process* 进程的 PId 值
expiredf	是一个函数（在附件 F. 2 中构造的），如果一个给定的定时器到时，它给出 True。

算法

第 1 行	从 t 中的定时器 *timer* 接收实际时间。
第 2 行	为定时器的设置开始检查。
第 4 行	检查它是否已经在队列中，并且它是否已经到时。
第 5 行	在前一种情况中，修改定时器的映射表使之包含“在队列中”。
第 6 行	对于被服务的 *sdl-process* 进程在发送者“SENDER”等于“SELF”的情况下，把定时器插入到队列中。

3.5 定时器处理器

这种处理器已经引入用于解释 SDL 中的全局时间的概念。它导致和一个外部滴答 *tick* 处理器的一种非常简单的通信。

***timer* processor (*timeinf*)** ≙ **(3.5.1)**

```
1  (let (timef, startt) = timeinf in
2   dcl time-now := startt type Value;
3   cycle {input mk-Time() from tick
4              ⇒  time-now := timef(c time-now),
5          input mk-Time-Request() from p
6              ⇒  output mk-Time-Answer(c time-now) to p})
```

type: *Time-information* ⇒

目的 在基础系统中解释定时器的处理

参数 实物 *timeinf* 包含在附件 F.2 中产生的两个成分（第 1 行）：

timef 是一个函数，在来自环境的每个“滴答”脉冲瞬间被调用。*timef* 函数因此压缩成两个问题：Time 类别中“+”的解释和在系统中时间值的划分（即，对于每个“滴答”，NOW 中的时间增量是多少）。

startt NOW 的初始值

算法

第 2 行 设 time-now 表示系统的（唯一的）全局时间。利用一个模型，此模型包括用于解释的开始时间（*startt*）以及一个更新函数（*timef*），有希望给出 SDL 时间概念的一个正确的描述。

第 4 行 更新时间。

第 6 行 返回 NOW。

3.6 非形式滴答处理器

***tick* processor ()** ≙ **(3.6.1)**

```
1  (cycle (output mk-Time() to timer;
2          /* models informally the interval between consecutive ticks */))
```

type: () ⇒

4 SDL 进程

这一章描述 META-IV 处理器 *sdl-process* 是怎样解释一个 SDL 进程的一个实例。SDL 进程的定义来自于实体字典 entity-dict。所有进程之间的和其它的通信都由基础系统管理。

每个 SDL 进程实例都有一个局部存储器，其类型按以下定义：

```
1  Stg = Identifier_1 m→ (Value | UNDEFINED)
```

4.1 sdl-process

META-IV 处理器 *sdl-process* 用它的实在参数由处理器 *system* 来创建。这些实在参数提供了有关它的环境的信息、它自己本身的信息和它必须解释的 SDL 进程的知识。当 SDL-process 进程已被解释以后，此 *sdl-process* 进程实例就停止存在。

sdl-process processor (*parentp*, *selfp*, *actparml*, *process-id*)(*dict*) ≜ **(4.1.1)**

```
1  (let mk-Identifier_1(qual, nm) = process-id in
2   let nullterm = dict(NULLVALUE) in
3   def dict_1 : dict + [SCOPEUNIT ↦ qual ⌢ ⟨mk-Process-qualifier_1(nm)⟩] +
4       [PORT ↦ start input-port(selfp, dict(EXPIREDF))] +
5       [SELF ↦ selfp] +
6       [PARENT ↦ (parentp = nil
7                          → nullterm,
8                      T → parentp)];
9   dcl sender := nullterm type Pid-Value;
10  dcl offspring := nullterm type Pid-Value;
11  dcl stg := [] type Stg;
12  (trap exit() with error in
13   (trap exit(STOP) with output mk-Stop() to system in
14    (let mk-ProcessDD(formparml, , , graph, ) = dict_1((process-id, PROCESS)) in
15     (def dict_2 : dict_1 + [(id, VALUE) ↦ mk-VarDD(id, sort, rev, stg) |
16                             (id, VALUE) ∈ dom dict_1 ∧ is-VarDD(dict_1((id, VALUE))) ∧
17                             s-Qualifier_1(id) = dict_1(SCOPEUNIT) ∧
18                             mk-VarDD(, sort, rev, ) = dict_1((id, VALUE))];
19      init-process-decls(dict_2);
20      init-process-parms(formparml, actparml)(dict_2);
21      output mk-Process-Initiated(dict_2(PORT)) to system;
22      int-process-graph(graph)(dict_2))))))
```

type: [*Pid-Value*] *Pid-Value Value-List Process-identifier*$_1$ → *Entity-dict* ⇒

目的

解释一个 sdl-process 进程

参数

parentp	创建这一进程的父辈进程的 SDL PID 值
selfp	这个进程的 SDL PID 值
actparml	实在参数表
process-id	这个进程的 SDL 标识符

算法

第 2 行　　抽取 PID 值 NULL 的 AS$_1$ 形式。

第 3—6 行　　增加 dict 的内容，使得：

第 3 行　　用 **SCOPEUNIT** 表示当前作用域。每当对 SDL 进程的解释进入一个新作用域单位时，就要修改 **SCOPEUNIT**。

第 4 行　　用 **PORT** 表示输入口的 CSP 名字。用于创建 *input-port* 处理器的实在参数是这个进程的 SDL PID 值和测试定时器是否到时的一个函数。输入口进程要处理发送给进程的信号和操作定时器。

第 5 行　　用 **SELF** 表示进程本身的 SDL PID 值。

第 6 行　　如果有父辈进程，用 **PARENT** 表示父辈进程的 SDL PID 值。如果进程创建于系统初始化期间，则不存在父辈进程。

第 9—10 行　　声明发送者变量和后代变量。它们都初始化为 *null-term*。

第 11 行　　声明一个变量，stg，它是该 sdl-process 进程的局部存储器，并把它初始化为空。

第 12 行　　捕捉错误的 exit（退出）。

第 13 行　　通过发送一个停止信号给系统来捕捉 **exit**（**STOP**），然后终止。

第 15—18 行　　改变 *dict* 以便对每个局部变量它都有一个对局部存储器的访问。所声明的变量和形式参数都被当作局部变量。

第 19 行　　按照 sdl-process 进程的声明扩大存储器。

第 20 行　　按照实在参数的内容扩大存储器。

第 21 行　　进程现在被初始化，通过把进程已初始化的信号 *Process-Initiated* 输出给系统 *system*，让 *system* 处理器知晓关于本进程初始化的知识和输入口 *input-port* 处理器的 CSP 名字。

第 22 行　　解释 sdl 进程。

$$\textit{init-process-decls}(\textit{dict}) \triangleq \tag{4.1.2}$$

```
1  (for all (id, VALUE) ∈ dom dict do
2   if is-VarDD(dict((id, VALUE))) ∧ s-Qualifier1(id) = dict(SCOPEUNIT)
3     then update-stg(id, nil)(dict)
4     else I)
```

type: *Entity-dict* ⇒

目的　　用正被解释的 sdl 进程的变量声明来更新存储器

算法

第 1 行　　对于所有那些带有 VALUE 属性的标识符，并且它们

第 2 行　　是变量描述符，且是在本进程中作出声明的。

第 3 行　　存储器被初始化为 **nil**，属于一个变量声明的任选初值在 AS_1 中被转换为在起动跃迁前的一个任务中的赋值语句。

init-process-parms(*formparml*, *actparml*)(*dict*) ≜ **(4.1.3)**

```
1  (for all i ∈ ind formparml do
2   update-stg(formparml[i], actparml[i])(dict))
```

type: *ParameterDD* Value-List* → *Entity-dict* ⇒

目的　　用进程的形式参数和任意采用的实在参数来更新进程的本地存储器

参数

formparml　　形式参数表

actparml　　实在参数表

算法

第 1—2 行　　用每个形式参数所表示的变量和它的相关的实在参数的值来更新本地存储器。范围的检查将推迟到 *update-stg*。

4.2　一个进程图的解释

介绍对一个进程图的解释被划分成为对每个图节点类型的解释函数。

int-process-graph(*graph*)(*dict*) ≜ **(4.2.1)**

```
1  (let mk-Process-graph1(mk-Process-start-node1(trans), stateset) = graph in
2   (tixe [statenm ↦ int-state-node(statenode)(dict) |
3          statenode ∈ stateset ∧ s-State-name1(statenode) = statenm] in
4    (let mk-Transition1(nodel, termordec) = trans in
5     int-transition(nodel, termordec)(dict))))
```

type: *Process-graph*$_1$ → *Entity-dict* ⇒

目的　　解释 SDL 进程图

参数

graph　　进程图

算法

第 1 行　　把图划分为起始跃迁和一个状态集合。

第 2 行　　通过解释有关的状态节点*state-node*$_1$从解释状态节点函数*int-state-node*和解释跃迁函数*int-transition*捕捉所有的退出**exit**(状态名)。**tixe**构件是模拟用于下一状态节点中的"goto"的一种非常方便的方法。关键字**tixe**后面跟一个映射表，把状态名映射到解释状态节点函数int-state-node的调用，函数int-state-node带有把状态名作为实在参数的状态节点state-node。如果在tixe构件的作用域里碰到了一个退出语句**exit**(statenm$_1$)，也就是说或者在tixe映射表(int-state-node)的范围中或在int-transition中碰到**exit**(statenm$_1$)的话，则进程的解释继续进行，去解释具有名字 statenm$_1$ 的状态节点。

第 4 行　　start-transition 的解释。

int-state-node(**mk-***State-node*$_1$(, **mk-***Save-signalset*$_1$(*saveset*), *inputset*))(*dict*) ≙ (4.2.2)

```
 1  (output mk-Next-Signal(saveset) to dict(PORT);
 2   input mk-Input-Signal(sid, actparml, sender') from dict(PORT)
 3      ⇒  (sender := sender';
 4          (let {inpnode} = {inp ∈ inputset | s-Signal-identifier1(inp) = sid} in
 5           let mk-Input-node1(, formparml, trans) = inpnode in
 6           for all i ∈ ind formparml do
 7           (if formparml[i] ≠ nil
 8              then update-stg(formparml, actparml[i])(dict)
 9              else I);
10           (let mk-Transition1(nodel, termordec) = trans in
11            int-transition(nodel, termordec)(dict)))))
```

type: *State-node*$_1$ → *Entity-dict* ⇒

目的　从 input-port 请求一个新信号，接收它并解释对应的跃迁

参数

state-node　包含有一个保存集 *saveset* 和一个输入集 *inputset*。*saveset* 是由输入口保存的信号。*inputset* 是一组信号和相关的跃迁。

算法

第 1 行　请求输入口 input-port 输出一个不在 *saveset* 中的信号，并保存属于 saveset 的所有信号。

第 2 行　接收一个信号，此信号由一个信号标识符、一个实参表和发送者的 SDL Pid 值组成。

第 3 行　用刚收到的信号的发送者的值来更新进程的发送者变量。

第 4 行　选择一个输入节点，该节点具有与所接收信号相同的信号标识符。

第 5 行　把所选择的输入分解为信号的形参表和相关的跃迁。

第 6—9 行　对于所有的形式参数：如果一个形式参数不是 **nil**，则用与它相关的变量和实参的值来更新相应的存储器。

第 11 行　解释所选择的跃迁。

int-transition(*nodel*, *termordec*)(*dict*) ≙ (4.2.3)

```
 1  (if nodel = ⟨⟩
 2     then cases termordec:
 3             (mk-Nextstate-node1(nm)
 4                → exit(nm),
 5              mk-Stop-node1()
 6                → exit(STOP),
 7              mk-Return-node1()
 8                → exit(RETURN),
 9              mk-Decision-node1(, , )
10                → int-decision-node(termordec)(dict))
11     else (int-graph-node(hd nodel)(dict);
12           int-transition(tl nodel, termordec)(dict)))
```

type: *Graph-node*$_1$* (*Terminator*$_1$ | *Decision-node*$_1$) → *Entity-dict* ⇒

目的　　解释一个跃迁

参数

nodel　　还未被解释的进程图中的节点所组成的表

termordec　　一个终端符节点或一个判定节点

算法

第 2 行　　如果节点表为空，则解释 *termordec*。

第 3 行　　下一状态节点用带有下一状态名字的退出 exit 来解释。

第 5 行　　用带有 STOP 的 exit 解释停止节点。

第 7 行　　用带有 RETURN 的 exit 解释返回节点。

第 9 行　　调用解释判定节点函数 *int-decision-node* 来解释判定节点。

第 10 行　　如果节点表非空，则第一个节点用解释图节点函数 *int-gragh-node* 来解释。

第 11 行　　跃迁的余下部分用递归来解释。

$$\textit{int-decision-node}(\textbf{mk-}\textit{Decision-node}_1(\textit{quest}, \textit{answset}, \textit{elseansw}))(\textit{dict}) \triangleq \qquad \textbf{(4.2.4)}$$

```
 1  (let answset' = matching-answer(quest, answset)(dict) in
 2   (if answset' ≠ {}
 3      then (let {mk-Decision-answer1(, trans)} = answset' in
 4              let mk-Transition1(nodel, termordec) = trans in
 5              int-transition(nodel, termordec)(dict))
 6      else (elseansw ≠ nil
 7              → (let mk-Else-answer1(trans) = elseansw in
 8                   let mk-Transition1(nodel, termordec) = trans in
 9                   int-transition(nodel, termordec)(dict)),
10            T → exit("§2.7.5: 无匹配的回答"))))
```

type:　$\textit{Decision-node}_1 \rightarrow \textit{Entity-dict} \Rightarrow$

目的　　根据提问来解释一个判定节点，以便从回答的集合中选择一个回答，并解释相关的跃迁。或者如果在回答集合中不存在一个匹配的答案，则去解释 else 跃迁。如果不存在与提问匹配的答案也没有 else 回答，则发生了错误。

参数

quest　　判定中的提问

answset　　回答集和相关的跃迁

elseansw　　任选的 else 跃迁

算法

第 1 行　　调用匹配回答函数 *matching-answer* 来抽取匹配回答的集合。

第 2—5 行　　如果所抽取的回答集合不是空集，则它仅包含一个回答（在静态语义测试中已保证回答不会重叠），于是解释与所选择的回答相关的跃迁。

第 6—9 行　　如果没有发现匹配的回答，则解释与 else 回答相关的跃迁。

第 10 行　　如果没有发现匹配的回答而且也没有提供 else 回答，则发生了错误。

matching-answer(*quest*, *answset*)(*dict*) ≜ **(4.2.5)**

```
1  (let gterm = dict(TRUEVALUE) in
2   {mk-Decision-answer1(valsetortext, ) ∈ answset |
3    (is-Range-condition1(valsetortext) ∧ is-Expression1(quest)
4       → (let mk-Range-condition1(orid, cset) = valsetortext in
5          let operator1 = make-valuetest-operator(quest, orid, cset) in
6          is-equivalent((trap exit() with false in
7                              eval-expression(operator1)(dict)), gterm, dict(SCOPEUNIT))(dict)),
8    T → text-equality(quest, valsetortext))})
```

type: *Decision-question*$_1$ *Decision-answer*$_1$-**set** → *Entity-dict* → *Decision-answer*$_1$-**set**

目的 从所提供的回答集合中寻找与所提供的提问匹配的回答集合

参数

quest	用于判定的提问
answset	回答的集合和相关的跃迁

结果 匹配的回答和与它相关的跃迁

算法

第 1 行	抽取 AS_0 字面值 TRUE 的 AS_1 形式。
第 2—8 行	根据第 3—8 行中的谓词来选择 *answset*，从 *answset* 构造回答的集合。
第 3—6 行	如果提问和回答都不是非形式的，则构造一个值测试运算符来测试提问是否与回答匹配。
第 8 行	如果提问或回答是非形式的，则利用正文相等测试函数 text-equality 来测试提问与回答的相等性。

make-valuetest-operator(*exp*1, *orid*, *cset*) ≜ **(4.2.6)**

```
1  (let v ∈ cset in
2   let op = cases v:
3            (mk-Closed-range1(aop, c1, c2)
4               → (let mk-Open-range1(relop, co1) = c1 in
5                  let t1 = mk-Operator-application1(relop, ⟨co1, exp1⟩),
6                      t2 = make-valuetest-operator(exp1, orid, {c2}) in
7                  mk-Operator-application1(aop, ⟨t1, t2⟩)),
8             mk-Open-range1(relop, c1)
9               → mk-Operator-application1(relop, ⟨exp1, c1⟩)) in
10 if card cset = 1 then
11   op
12  else
13   (let op' = make-valuetest-operator(exp1, orid, cset − {v}) in
14    mk-Operator-application1(orid, ⟨op, op'⟩)))
```

type: *Expression*$_1$ *Operator-identifier*$_1$ *Condition*$_1$-**set** → *Expression*$_1$

目的 构造一个运算符，该运算符可以测试表达式 exp_1 是否满足由被标识的运算符、*orid* 和范围条件集合 *cset* 合成的条件。

参数

exp_1	待测试的表达式

orid 指明用于组成 *cset* 中的条件的运算符

cset exp_1 应该满足的范围条件的集合

结果 是一个运算符，它能够测试 exp_1 的值是否与条件匹配。该条件是由 *orid* 标识的运算符和 *cset* 所合成的条件。

算法

第 1 行 在 *cset* 中选择一个范围条件 v。

第 3 行 如果 v 是一个闭合范围 $Closed\text{-}range_1$，则它被分解为一个 and 运算符 *aop*，和两个开范围 $Open\text{-}range_1$ 条件 c_1 和 c_2。

第 5 行 设 t_1 表示 c_1 的值测试运算符。

第 6 行 设 t_2 表示 c_2 的值测试运算符。

第 7 行 合成一个在 *aop* 上应用 t_1 和 t_2 的一个 $Operator\text{-}application_1$，并且设 *op* 表示该应用，即，构造运算符 “**AND**”（“<=”（co1，exp1），t2）。

第 8 行 合成一个 $Operator\text{-}application_1$，它在 *relop* 上应用 exp_1 和 c_1，并且把它称为 v，即，构造运算符 “<=”（exp1，c1）。

第 11 行 如果 v 是 *cset* 中的最后元素，则返回 *op*。

第 13 行 为余下的 *cset* 构造一个值测试运算符，并把它称为 *op′*。

第 14 行 合成一个 $Operator\text{-}application_1$，其中两个运算符应用 *op* 和 *op′* 作用于 *orid*。

$$text\text{-}equality(exp\text{-}text, valueset\text{-}text) \triangleq \tag{4.2.7}$$

```
1  (/* This informal Meta-IV text denotes the equality test */;
2   /* between informal question and/or informal answer */)
```

type: $(Informal\text{-}text_1 \mid Expression_1)\ (Informal\text{-}text_1 \mid Range\text{-}condition_1) \rightarrow Bool$

$$int\text{-}graph\text{-}node(graphnode)(dict) \triangleq \tag{4.2.8}$$

```
1  ( cases graphnode:
2    (mk-Task-node1(silt)          → int-task-node(silt)(dict),
3     mk-Output-node1(,,,)         → int-output-node(graphnode)(dict),
4     mk-Create-request-node1(,)   → int-create-node(graphnode)(dict),
5     mk-Call-node1(,)             → int-call-node(graphnode)(dict),
6     mk-Set-node1(,,)             → int-set-node(graphnode)(dict),
7     mk-Reset-node1(,)            → int-reset-node(graphnode)(dict)))
```

type: $Graph\text{-}node_1 \rightarrow Entity\text{-}dict \Rightarrow$

目的 解释一个图节点

参数

graphnode 待解释的图节点

$$int\text{-}task\text{-}node(silt)(dict) \triangleq \tag{4.2.9}$$

```
1  ( cases silt:
2    (mk-Assignment-statement1(,) → int-assign-stmt(silt)(dict),
3     mk-Informal-text1()         → int-informal-text(silt)))
```

type: $(Assignment\text{-}statement_1 \mid Informal\text{-}text_1) \rightarrow Entity\text{-}dict \Rightarrow$

目的　　解释一个任务节点

参数

silt　　一个赋值语句或非形式的正文

算法

第 1 行　　*Silt* 被解释为一个赋值语句或者是非形式正文。

$$int\text{-}set\text{-}node(\textbf{mk-}Set\text{-}node_1(texp, tid, exprl))(dict) \triangleq \tag{4.2.10}$$

```
1  (def val : eval-expression(texp)(dict);
2   def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
3   let mk-SignalDD(sortl) = dict((tid, SIGNAL)) in
4   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
5   def f(t1, t2) : is-equivalent(t1, t2, dict(SCOPEUNIT))(dict);
6   if (∀i ∈ ind vall)(range-check(sortl[i], vall'[i])(dict))
7     then output mk-Set-Timer(tid, val, vall', f) to dict(PORT)
8     else exit (" §5.4.1.9: 值不在同义类型范围内"))
```

type:　$Set\text{-}node_1 \rightarrow Entity\text{-}dict \Rightarrow$

目的　　解释设置节点的过程是：检查定时器信号的实在参数看是否有范围错误，然后把设置定时器的信号输出到输入口。

参数

texp　　定时器表达式，它的值表示该定时器应该被设置的时间

tid　　待设置的定时器的标识符

exprl　　定时器的实在参数

算法

第 1 行　　计算定时器表达式和实在参数表的值。

第 4 行　　参见简化项函数 *reduce-term*。

第 5 行　　构造用于输入口 *inputport* 处理器中的 *isequivalent* 函数来检测该定时器是否已经用相同的实在参数来设置。

第 6 行　　检测定时器的实在参数值是否在它们的类别所允许的范围之内。

$$int\text{-}reset\text{-}node(\textbf{mk-}Reset\text{-}node_1(tid, exprl))(dict) \triangleq \tag{4.2.11}$$

```
1  (def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
2   let mk-SignalDD(sortl) = dict((tid, SIGNAL)) in
3   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
4   def f(t1, t2) : is-equivalent(t1, t2, dict(SCOPEUNIT))(dict);
5   if (∀i ∈ ind vall')(range-check(sortl[i], vall'[i])(dict))
6     then output mk-Reset-Timer(tid, vall', f) to dict(PORT)
7     else exit (" §5.4.1.9: 值不在同义类型范围内"))
```

type:　$Reset\text{-}node_1 \rightarrow Entity\text{-}dict \Rightarrow$

目的　　解释复位节点。其过程是：检查定时器信号的实在参数是否在它们的类别所允许的范围之内。如果是的，就把复位定时器 *Reset-Timer* 信号输出到 *input-port* 处理器。

参数

tid	要被复位的定时器的标识符
exprl	用于定时器的实在参数

算法

第 1 行	计算定时器的实在参数表的值。
第 3 行	见简化项函数 *Reduce-term*。
第 4 行	构造一个用于 *input-port* 处理器中的 is-equivalent 函数，以便测试这个定时器是否已经用相同的实在参数设置了。
第 5 行	测试定时器的实在参数的值是否在它们的类别所允许的范围内。

$$\textit{int-assign-stmt}(\textbf{mk-}\textit{Assignment-statement}_1(\textit{vid}, \textit{exp}))(\textit{dict}) \triangleq \qquad (4.2.12)$$

```
1  (def val : eval-expression(exp)(dict);
2   update-stg(vid, val)(dict))
```

type: $\textit{Assignment-statement}_1 \to \textit{Entity-dict} \Rightarrow$

目的 把表达式的值赋给变量

参数

vid	变量
exp	表达式

算法

第 1 行	计算表达式的值。
第 2 行	用 *vid* 和表达式的值更新存储器。

$$\textit{int-informal-text}(\textbf{mk-}\textit{Informal-text}_1()) \triangleq \qquad (4.2.13)$$

```
1  (/* This informal Meta-IV text denotes the interpretation of informal text */)
```

type: $\textit{Informal-text}_1 \Rightarrow$

$$\textit{int-output-node}(\textbf{mk-}\textit{Output-node}_1(\textit{sid}_1, \textit{exprl}, \textit{dest}, \textit{via}))(\textit{dict}) \triangleq \qquad (4.2.14)$$

```
1  (def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
2   def pidval : eval-expression(dest)(dict);
3   let mk-SignalDD(sortl) = dict((sid1, SIGNAL)) in
4   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
5   if (∀i ∈ ind vall')(range-check(sortl[i], vall'[i])(dict))
6     then output mk-Send-Signal(sid1, vall', pidval, via) to system
7     else exit ("§5.4.1.9: 值不在同义类型范围内"))
```

type: $\textit{Output-node}_1 \to \textit{Entity-dict} \Rightarrow$

目的 解释一个输出节点。其过程是：检查实在参数是否在它们的类别所允许的范围内，如果是的话，则把发送信号 Send-Signal 输出给系统处理器。

参数

sid_1	要被发送的信号的标识符
exprl	该信号的实在参数
dest	表示该信号应该发送到的进程的 PID 表达式
via	是一组路径标识符，用来表示该信号应该通过的路径

算法

第 1 行	计算实在参数表和 pid-value 的值。
第 4 行	见简化项函数 reduce-term。
第 5 行	检测该信号的实在参数的值是否在它们的类别所允许的范围之内。

$$\textit{int-create-node}(\textbf{mk-}\textit{Create-request-node}_1(\textit{pid}, \textit{exprl}))(\textit{dict}) \triangleq \qquad \textbf{(4.2.15)}$$

```
1  (def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
2   let mk-ProcessDD(formparms,,,,) = dict((pid, PROCESS)) in
3   let sortl = ⟨s-Sort-reference-identifier1(dict((formparms[i], VALUE))) | 1 ≤ i ≤ len formparms⟩ in
4   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
5   output mk-Create-Instance-Request(pid, vall') to system;
6   input mk-Create-Instance-Answer(offspring') from system
7      ⇒  if offspring' = nil then
8            (let nullterm = dict(NULLVALUE) in
9             offspring := nullterm)
10           else
11            offspring := offspring')
```

type : $\textit{Create-request-node}_1 \rightarrow \textit{Entity-dict} \Rightarrow$

目的 解释创建节点

参数

pid	要被创建的进程的标识符
exprl	实在参数表

算法

第 1 行	计算每个实在参数的值。
第 2 行	建立形式参数的类别引用标识符 Sort-reference-identifiers 表。
第 4 行	参见简化项函数 *reduce-term*
第 5 行	把一个创建实例请求 Greate-Instance-Request 输出到系统进程处理器。
第 6 行	输入被创建的进程的 PID 值，或者如果该进程不能被创建，就输入 nil。
第 9 行	如果该进程不能被创建，就把后代 offspring 赋值为 *nullterm*。如果已经存在那个进程的最大实例数，则不能创建新的进程。
第 11 行	如果可以创建进程，则把后代 offspring 赋值为从系统处理器接收到的 PID 值。

int-call-node(**mk-***Call-node*$_1$(*prd-id*, *exprl*))(*dict*) ≜ **(4.2.16)**

```
1  (dcl newstg := [] type Stg;
2   let mk-ProcedureDD(formparms, graph) = dict((prd-id, PROCEDURE)) in
3   let mk-Identifier1(qual, nm) = prd-id in
4   let newlevel = qual ⌢ ⟨mk-Procedure-qualifier1(nm)⟩ in
5   let decl-parm-set = {(mk-Identifier1(l, ), VALUE) ∈ dom dict | l = newlevel} in
6   let dict1 = establ-dyn-dict(formparms, exprl, newstg, decl-parm-set)(dict) in
7   let dict2 = dict1 + [SCOPEUNIT ↦ newlevel] in
8   (trap exit(RETURN) with I in
9    int-procedure-graph(graph)(dict2)))
```

type: *Call-node*$_1$ → *Entity-dict* ⇒

目的　　解释一个过程调用节点

参数

prd-id	被调用过程的标识符
exprl	用于过程调用的实在参数

算法

第 1 行	声明一个新的空的存储器变量，用作局部于过程的存储器。
第 2 行	建立过程的形式参数表和过程图。
第 3—4 行	计算过程的作用域层次。
第 5 行	抽取在该层次上作为形式参数定义或使用的变量集合。
第 6 行	通过调用函数 *establ-dyn-dict* 来构造新的 dict 并按照新的定义、形式参数和实在参数来更新新的存储器。
第 7—9 行	不做任何处理就进入新的层次，并且捕捉来自于过程图解释的 **exit**（RETURN）。

int-procedure-graph(*graph*)(*dict*) ≜ **(4.2.17)**

```
1  (let mk-Procedure-graph1(mk-Procedure-start-node1(trans), stateset) = graph in
2   tixe [statenm ↦ int-state-node(statenode)(dict) |
3         statenode ∈ stateset ∧ s-State-name1(statenode) = statenm] in
4   let mk-Transition1(nodel, termordec) = trans in
5   int-transition(nodel, termordec)(dict))
```

type: *Procedure-graph*$_1$ → *Entity-dict* ⇒

目的　　解释一个过程图

参数

graph	过程图

算法

第 1 行	把该图划分成一个起始跃迁和一组状态。
第 2 行	通过解释有关的状态节点捕捉所有的来自于函数int-state-node和函数int-transition的退出 **exit**（*statenm*）（tixe 构件的解释在函数 int-process-graph 注释中给出）。
第 4 行	解释起始跃迁。

4.3 辅助函数

下面将定义在前面几节中所用的辅助函数。这些辅助函数有计算表达式、进行范围检查、管理局部存储器和实体字典 entity dict 的动态部分（见 2.6 节）。

eval-expression(*exp*)(*dict*) ≙ **(4.3.1)**

```
1  (if exp = nil then
2     nil
3   else
4    cases exp:
5    (mk-Identifier1(,)
6       → eval-variable-identifier(exp)(dict),
7     mk-Ground-expression1()
8       → eval-ground-expression(exp)(dict),
9     mk-Operator-application1(,)
10      → eval-operator-application(exp)(dict),
11    mk-Conditional-expression1(e1, e2, e3)
12      → eval-conditional-expression(e1, e2, e3)(dict),
13    mk-View-expression1(,)
14      → eval-view-expression(exp)(dict),
15    mk-Timer-active-expression1(,)
16      → eval-active-expression(exp)(dict),
17    mk-Now-expression1()
18      → eval-now-expression(),
19    mk-Self-expression1()
20      → mk-Ground-term1(dict(SELF)),
21    mk-Parent-expression1()
22      → mk-Ground-term1(dict(PARENT)),
23    mk-Offspring-expression1()
24      → mk-Ground-term1(c offspring),
25    mk-Sender-expression1()
26      → mk-Ground-term1(c sender)))
```

type: [*Expression*$_1$] → *Entity-dict* ⇒ [*Value*]

目的　　计算一个 AS$_1$ 表达式的值

参数

AS$_1$ 表达式

结果　　表达式的值

算法

第 1 行　　如果表达式等于 nil，则结果是 nil 值。

第 19 行　　如果表达式是一个 Self、Parent、Offspring 或 Sender-expression，则返回 self、parent、offspring 或 sender 的内容的基项。

eval-variable-identifier(*id*)(*dict*) ≙ **(4.3.2)**

```
1  (let mk-VarDD(vid, , , stg) = dict((id, VALUE)) in
2   if c stg(vid) = UNDEFINED
3     then exit ("5.5.2.2: 所访问变量的值未定义")
4     else c stg(vid))
```

type: *Identifier*$_1$ → *Entity-dict* ⇒ *Value*

目的　　计算一个变量标识符的值

参数

id　　变量标识符

结果　　是那个变量的内容，如果有的话。

算法

第 1 行　　取得所访问的标识符。

第 3 行　　如果所访问的标识符的存储器的内容是未确定的，则出现了错误。

第 4 行　　返回所访问的标识符的存储器的内容。

eval-ground-expression(mk-*Ground-expression*$_1$(*ex*))(*dict*) ≙ **(4.3.3)**

```
1  (let mk-Ground-term1(e) = ex in
2   if is-Identifier1(e) then
3     ex
4    else
5     if is-Conditional-term1(e) then
6       (let mk-Conditional-term1(bex, ex1, ex2) = e in
7        eval-conditional-expression(bex, ex1, ex2)(dict))
8      else
9       (let (opid, arglist) = e in
10       let mk-OperatorDD(sortlist, sort) = dict((opid, VALUE)) in
11       if (∀i ∈ ind arglist)(range-check(sortlist[i], arglist[i])(dict)) then
12         (let arglist' = ⟨eval-expression(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
13          let t = mk-Ground-term1((opid, arglist')) in
14          if range-check(sort, t)(dict)
15            then t
16            else exit ("§5.4.1.9: 值不在同义类型范围内"))
17        else
18         exit ("§5.4.1.9: 值不在同义类型范围内"))
```

type: *Ground-expression*$_1$ → *Entity-dict* → *Value*

目的　　计算一个基项表达式的值

参数

ex　　一个基项

结果　　结果是该基项表达式的值。它是作为运算符标识符和计算过的变元表给出的。

算法

第 2 行　　如果基项由一个标识符组成，则返回该基项。

第 6 行　　如果基项是一个条件项，则分解并计算它。

第 9 行　　如果基项既不是一个标识符又不是一个条件表达式，则它必定是一个运算符的应用。

第 11 行　　按照运算符变元表相关的类别表，测试运算符变元表有无范围错误。如果没有检测到范围错误，则把运算符标识符和被计算的类别表组合成为一个基项。否则，就是出现了范围错误。

第 14 行　　按照该运算符的类别，测试基项有无范围错误。如果没有检测到范围错误，就返回该基项。

eval-operator-application(mk-*Operator-application*$_1$(*opid*, *expl*))(*dict*) ≜ **(4.3.4)**

```
1  (def vall : ⟨eval-expression(expl[i])(dict) | 1 ≤ i ≤ len expl⟩;
2   let term = mk-Ground-term1((opid, vall)) in
3   let mk-OperatorDD(sortl, result) = dict((opid, VALUE)) in
4   if (∀i ∈ ind sortl)(range-check(sortl[i], vall[i])(dict)) ∧
5     range-check(result, term)(dict)
6     then term
7     else exit ( "§5.4.1.9: 值不在同义类型的范围内"))
```

type: *Operator-application*$_1$ → *Entity-dict* ⇒ *Value*

目的　　计算一个运算符的应用

参数

opid　　运算符的标识符

expl　　此应用的变元表

结果　　运算符应用的值，也就是一个运算符标识符和已求值的变元表组成的基项的值

算法

第 1 行　　计算变元表中各变元的值。

第 2 行　　构造一个由运算符标识符和已求值的变元表组成的基项。

第 3 行　　查找实体字典 *Entity-dict* 中的运算符表示法。

第 4 行　　检测已计算出的各变元的值是否在它们各自的类别范围之内。

第 5 行　　检测第 2 行中构成的项是否在结果的类别范围之内。

第 6 行　　如果没发现范围错误，返回第 2 行中构成的项。

eval-view-expression(mk-*View-expression*$_1$(id_1, *exp*))(*dict*) ≜ **(4.3.5)**

```
1  (def pid : eval-expression(exp)(dict);
2   def pid' : reduce-term(dict(PIDSORT), pid, dict(SCOPEUNIT))(dict);
3   output mk-View-Request(id1, pid') to view;
4   (input mk-View-Answer(val) from view
5       ⇒  if val = UNDEFINED
6            then exit ("§5.5.2.2: 被视见的值未定义")
7            else val))
```

type: *View-expression*$_1$ → *Entity-dict* ⇒ *Value*

目的 求一个 VIEW 表达式的值

eval-conditional-expression(exp_1, exp_2, exp_3)(*dict*) ≙ (4.3.6)

```
1  (let trueterm = dict(TRUEVALUE) in
2   let falseterm = dict(FALSEVALUE) in
3   if is-equivalent(eval-expression(exp1)(dict), trueterm, dict(SCOPEUNIT))(dict) then
4     eval-expression(exp2)(dict)
5   else
6    if is-equivalent(eval-expression(exp1)(dict), falseterm, dict(SCOPEUNIT))(dict) then
7      eval-expression(exp3)(dict)
8    else
9     exit ("§5.5.2.3: 条件必须计算出 TRUE 或 FALSE"))
```

type: $Expression_1$ $Expression_1$ $Expression_1$ → *Entity-dict* ⇒ *Value*

目的 求一个条件表达式的值

参数

exp_1	条件表达式
exp_2	条件满足时的后继表达式
exp_3	条件不满足时的替换表达式

结果 根据是否满足这个条件而得到后继表达式的值或者替换表达式的值

算法

第 1 行	抽取 **TRUE** 的 AS_1 项。
第 2 行	抽取 **FALSE** 的 AS_1 项。
第 3 行	如果真值项与条件相等，则
第 4 行	求后继表达式的值，否则
第 6 行	如果假值项与条件相等，则
第 7 行	求替换表达式的值，否则
第 9 行	就出现一个错误（这仅在布尔类别已增加了其它值的情况下才有可能）。

eval-active-expression(mk-*Timer-active-expression*$_1$(*timer*, *exprl*))(*dict*) ≙ (4.3.7)

```
1  (let mk-Identifier1(qual, ) = timer in
2   let mk-SignalDD(sortl) = dict((timer, SIGNAL)) in
3   let trueterm = dict(TRUEVALUE),
4       falseterm = dict(FALSEVALUE) in
5   def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
6   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
7   let f(t1, t2) = is-equivalent(t1, t2, qual)(dict) in
8   output mk-Active-Request(timer, vall', f) to dict(PORT);
9   (input mk-Active-Answer(b) from dict(PORT)
10        ⇒  if b then mk-Ground-term1(trueterm) else mk-Ground-term1(falseterm)))
```

type: *Timer-active-expression*$_1$ → *Entity-dict* ⇒ *Value*

目的 检测所指定的定时器是否活跃

参数

timer	定时器的标识符
exprl	定时器的参数

结果　　如果所描述的定时器是活跃的，结果就是 TRUE 的 AS_1 值，否则就是 FALSE 的 AS_1 值。

算法

第 1 行　　建立定时器的类别表。
第 3—4 行　　抽取 TRUE 和 FALSE 的 AS_1 值。
第 5 行　　求定时器参数的值。
第 6 行　　参见简化项函数 *reduce-term*
第 7 行　　定义一个函数去检测 *val'* 和潜在活跃的定时器的参数是否等价。
第 8 行　　把一个请求 *Active-Request* 发送到输入口。
第 9 行　　从输入口接收一个带有参数 *b* 的回答 *Active-Answer*。参数 *b* 表示定时器的“活跃性”。
第 10 行　　根据 *b* 返回 TRUE 或 FALSE 的 AS_1 形式。

eval-now-expression() ≙ **(4.3.8)**

```
1  (output mk-Time-Request() to timer;
2   (input mk-Time-Answer(val) from timer
3         ⇒  val))
```

type: () ⇒ *Value*

目的　　求 now 表达式的值

结果　　表示 now 的值，见 *Timer* 处理器

算法

第 1 行　　把一个时间请求 *Time-Request* 发送到定时器处理器。
第 2 行　　接收带有 now 的值的回答 *Time-Answer*。

establ-dyn-dict(*formparml*, *exprl*, *stg*, *decl-parm-set*)(*dict*) ≙ **(4.3.9)**

```
1  (if decl-parm-set ≠ {} then
2     (let (id, VALUE) ∈ decl-parm-set in
3      def dict1 : (mk-InoutparmDD(id) ∈ elems formparml
4                     → (let i ∈ ind formparml be s.t. formparml[i] = mk-InoutparmDD(id) in
5                        let mk-VarDD(vid, sid, rev, stg') = dict((exprl[i], VALUE)) in
6                        [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg')]),
7                   mk-InparmDD(id) ∈ elems formparml
8                     → (let i ∈ ind formparml be s.t. formparml[i] = mk-InparmDD(id) in
9                        let mk-VarDD(vid, sid, rev, ) = dict((id, VALUE)) in
10                       let dict' = [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg)] in
11                       update-stg(vid, eval-expression(exprl[i])(dict))(dict + dict');
12                       dict'),
13                  T → (let mk-VarDD(vid, sid, rev, ) = dict((id, VALUE)) in
14                       let dict' = [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg)] in
15                       update-stg(vid, nil)(dict + dict');
16                       dict'));
17     return establ-dyn-dict(formparml, exprl, stg, decl-parm-set \ {(id, VALUE)})(dict) + dict1)
18    else
19     dict)
```

type: *FormparmDD** *Expression*$_1$* **ref** *Stg* (*Identifier*$_1$ VALUE)-**set** → *Entity-dict* ⇒ *Entity-dict*

目的　　在解释过程调用时完成实体字典 entity-dict 的动态部分中的必要的改变。此外，按照过程中定义的变量和 IN 形式参数来更新存储器。

参数

formparml	形式参数表
actparml	实在参数表
stg	对新存储器的引用
dcl-parm-set	应该更新的 *dict* 和存储器的 *dict* 项目的集合

结果　　更新过的实体字典 *Entity-dict*

算法

第 1 行	如果 dict 项目的集合是空的，递归停止。
第 2 行	取出 dict 的项目之一。
第 3 行	如果它对应于 IN/OUT 的形式参数之一，则：
第 4—5 行	查阅相对应的实在参数和它在 *dict* 中的描述符。
第 6 行	用实在参数的变量描述符来描述形式参数的变量。
第 7 行	如果它对应于一个 IN 形式参数，则：
第 8 行	查阅那个形式参数的标引号。
第 9—10 行	改变形式参数的变量描述符，以便引用新的存储器。
第 11 行	使用新描述符的实在参数的值被用来更新该变量的存储器。
第 13 行	如果它既不对应于一个 IN 形式参数又不对应于一个 IN/OUT 形式参数，则：
第 13—16 行	象 IN 形式参数那样进行处理，但是用 **nil**（**UNDEFINED**）来更新存储器。
第 17 行	对于余下的 *dcl-parm-set*，调用函数 *establ-dyn-dict* 来处理，并用刚构造的项目来重写结果。
第 19 行	当不再有定义或参数要检查时，就停止递归并返回旧的 dict。

$update\text{-}stg(id, val)(dict) \triangleq$ **(4.3.10)**

```
1  (let mk-VarDD(vid, sid, revealed, stg') = dict((id, VALUE)) in
2   def val' : if val = nil then
3                UNDEFINED
4              else
5                reduce-term(sid, val, dict(SCOPEUNIT))(dict);
6   if range-check(sid, val')(dict)
7     then (stg' := c stg' + [vid ↦ val'];
8           if revealed = REVEALED then
9             (output mk-Reveal(vid, val', dict(SELF)) to view)
10          else
11            I)
12    else exit(" §5.4.1.9: 值不在同义类型范围内")
```

type: $Identifier_1$ *Value* → *Entity-dict* ⇒

目的　　用所应用的变量的值来更新此变量标识符的存储器，如果声明它是被透露变量，则透露这个变量。

参数

id	是变量标识符，其存储器应该更新
val	是一个值，用来更新存储器

算法

第 1 行	查阅变量标识符的描述。
第 2 行	如果 *val* 不是 **nil**，则必须改变它以便匹配那个变量的类别标识符（见简化项函数 *reduce-term*），如果值 *val* 等于 **nil**，这个存储器就被更新为 UNDEFINED。
第 7—8 行	用新的变量及其对应值重写该被引用的存储器。
第 9 行	对于被透露的变量，Reveal被发送给视见处理器。Reveal带有该变量的标识符、“简化了”的值和本进程的 PID 值。
第 12 行	如果“简化了”的值不在该变量类别的范围之内，就出现范围错误。

***range-check*(*sort-id*, *value*)(*dict*)** ≙ **(4.3.11)**

```
1  if value ∈ {nil, UNDEFINED} then
2    true
3  else
4   ( cases dict((sort-id, SORT)):
5     (mk-SyntypeDD(, mk-Range-condition1(orid, condset))
6        → (let operator1 = make-valuetest-operator(value, orid, condset) in
7           let value' = eval-ground-expression(operator1)(dict) in
8           let trueterm = dict(TRUEVALUE) in
9           is-equivalent(eval-expression(value')(dict), trueterm, dict(SCOPEUNIT))(dict)),
10     T → true))
```

type: *Sort-reference-identifier*$_1$ [*Value*] → *Entity-dict* → *Bool*

目的 检测一个值是否在它的类别范围之内

参数

sort-id	类别标识符
value	要被检测的值

结果 如果该值是在范围之内，则结果为真，否则为假

算法

第 1 行	**nil** 或 UNDEFINED 是在所有的类别范围之内。
第 4 行	查阅类别的描述符。
第 6 行	如果类别是一个同义类型，则利用 *orid*、*condset* 和 *value* 构造一个 SDL 运算符（*operator*$_1$）来执行范围检查，参见函数 *make-valuetest-operator*。
第 8 行	抽取 true 的 AS$_1$ 形式。

5 构造实体字典 *Entity-dict* 和处理抽象数据类型

本章包含用来建立实体字典 *entitydict* 的一些函数(见实体字典 *Entity-dict* 的域定义)。实体字典 *entitydict* 为 *sdl-process* 进程所使用，也为 *System* 进程所使用。*System* 进程还应用填写项目函数 *extract-dict* 来建立这个字典。

这章被分为四个部分：

1. 建造简单的独立的描述符，例如同义类型、变量、信号等的描述符。也建造进程的描述符(即 *Process-DD*)，但是带有一个空的可达性 *Reachability* 集合。

   要为实体建造描述符，不管它们是否定义在一致性子集合所包括的作用域单位中。这是因为，数据类型的一致性检查要在所有的作用域单位中进行。

2. 建造 $Data\text{-}type\text{-}definition_1$ 的描述符 (*TypeDD*)。对每个作用域单位来说，应先创建这个描述符然后建造类别的描述符 (*SortDD*)。

3. 一致性子集的选择

4. 建造进程的可达集 *Reachabilities* (即，建造进程的所有可能的通信路径)。

在构造了所有实体的描述符之后，但在构造可达集 *Reachabilities* 之前，通过删除不在一致性子集合中的进程的描述符来选择一致性子集。*entitydict* 的构成可以被看作是静态语义学与动态语义学之间的某个中间层次。本章中的错误条件(检查一致性子集合和抽象数据类型的一致性)可以被看作是某个附加的静态条件，该条件放在动态语义学中是因为：

- 为了检查一致性具体化子集合的选择需要构造可达集 *Reachabilities*。
  严格说来，由于它不是 SDL 规格的一部分，一致性(具体化)子集合的选择不是一个错误条件；但是为了检查它的性质，要对体现该一致性子集合的功能块标识符的集合作一致性检查。

- 对等价类和对判定回答的相互排除的性质所进行的一致性检查不能借助于 AS_1 容易地表达出来。即，这些(静态的)检查被放在动态语义学中是因为需要构造等价类。

$$extract\text{-}dict(as_1 tree, blockset, expiredf, terminf) \triangleq \tag{5.1}$$

```
(let (as1pid, as1null, as1true, as1false) = terminf in
 let d = [EXPIREDF   ↦ expiredf,
          PIDSORT    ↦ as1pid,
          NULLVALUE  ↦ as1null,
          TRUEVALUE  ↦ as1true,
          FALSEVALUE ↦ as1false] in
 (let mk-System-definition1(nm, bset, cset, sigset, tp, synset) = as1tree in
  let level = ⟨mk-System-qualifier1(nm)⟩ in
  let leafprocesses = select-consistent-subset(bset, blockset, level) in
  let dict' = extract-sortdict(tp, level)(d) in
  let dict'' = merge {make-entity(entity, level)(dict') | entity ∈ (sigset ∪ synset ∪ bset)} in
  let d' = dict'' + [(id, q) ↦ d((id, q)) | (id, q) ∈ dom d ∧ (q = PROCESS ⊃ id ∈ leafprocesses)] in
  make-structure-paths(bset, cset, level)(d')))
```

type : *System-definition*$_1$ *Block-identifier*$_1$**-set** *Is-expired Term-information* → *Entity-dict*

目的　构造实体字典 *entitydict*，提供给 *sdl-process* 进程和 *system* 进程使用。该实物由 *system* 进程构成，并在每次起动一个新的 *sdl-process* 时把该实物作为实在参数给出。

参数

as$_1$*tree*　一个系统的抽象语法表示法，即域 *System-definition*$_1$ 的一个实物

blockset　是由一组功能块标识符和功能块子结构标识符表示的（假定的）一致性子集合。尽管系统作用域单位也在该一致性子集合中，但 *blockset* 中不包括它。

expiredf　是一个函数。如果给定的定时器已经到时，该函数给出 true。

terminf　基础系统所使用的某些 AS$_1$ 标识符

结果　域实体字典 *Entity-dict* 的一个实物

算法

第 1 行　分解 *Term-information*（附件 F. 2 中定义的），它包含 PiD 类别的标识符 *Identifier*$_1$、NULL 字面值、TRUE 字面值和 FALSE 字面值。

第 2—6 行　建立初始的实体字典 *entitydict*，到时函数和项的信息被放在该实体字典 *entitydict* 中。

第 8 行　给出表示系统层次的限定符。

第 9 行　检查一致性子集合是否是良形式的并抽取一致性子集合中所包含的进程的标识符。

第 10 行　建造描述符。这些描述符是在系统层次上定义的数据类型定义 *Data-type-definition*$_1$、字面值和运算符的描述符。

第 11 行　构造信号（*sigset*）、同义类型（*synset*）和功能块（*bset*）对实体字典 *entitydict* 的贡献。

第 12 行　删除不在一致性子集合中的进程的描述符。

第 13 行　把可达集 *Reachabilities* 插入到所有进程的描述符（*ProcessDD*）中。函数 *make-structure-paths* 返回它们已被插入的实体字典 *entitydict*。

***make-entity*(*entity*, *level*)(*dict*)** ≜ **(5.1.1)**

```
1  cases entity:
2   (mk-Timer-definition1(nm, sortlist)
3      → dict + [(mk-Identifier1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist)],
4   mk-Signal-definition1(,,)
5      → dict + make-signal-dict(entity, level),
6   mk-Process-definition1(,,,,,,,,,,)
7      → make-process-dict(entity, level)(dict),
8   mk-Procedure-definition1(,,,,,,)
9      → make-procedure-dict(entity, level)(dict),
10  mk-Variable-definition1(nm, sort, rev)
11     → dict + [(mk-Identifier1(level, nm), VALUE) ↦ mk-VarDD(, sort, rev, )],
12  mk-Syn-type-definition1(nm, psort, ran)
13     → dict + [(mk-Identifier1(level, nm), SORT) ↦ mk-SyntypeDD(psort, ran)],
14  mk-Block-definition1(,,,,,,,)
15     → make-block-dict(entity, level)(dict),
16  T → dict)
```

type: $Decl_1\ Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Entity\text{-}dict$

目的 返回实体字典 *Entity-dict* (*dict*)，它已经增加了对一个实体的贡献

参数

entity 实体的 AS_1 定义

level 一个限定符，用来表示包含此定义的作用域单位

算法 构造对手头实体的贡献。注意，定时器被看作是正常的信号，不需要视见变量的描述符（在此 case 语句中没有 $View\text{-}definition_1$ 的情况）。

***make-signal-dict*(mk-*Signal-definition*$_1$(*nm*, *sortlist*, *refinement*), *level*)** ≜ **(5.1.2)**

```
1  (let d = [(mk-Identifier1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist)] in
2   if refinement = nil then
3     d
4    else
5     (let mk-Signal-refinement1(subsigset) = refinement in
6      let level' = level ⌢ ⟨mk-Signal-qualifier1(nm)⟩ in
7      d + merge {make-signal-dict(s-Signal-definition1(sdef), level') | sdef ∈ subsigset}))
```

type: $Signal\text{-}definition_1\ Qualifier_1 \rightarrow Entity\text{-}dict$

目的 构造一个信号和它的子信号对实体字典 *entitydict* 的贡献。注意，信号描述符不能分辨一个信号是否是子信号。这是由于仅当通信链接的两端中选择了相同的信号时，子信号选择才是良形式的，与这些信号是否为子信号无关。

参数

$Signal\text{-}definition_1$ AS_1 信号定义包括

nm 信号的名字

sortlist 由信号传送的多个值的类别

refinement 信号具体化的部分

level 一个限定符，用来表示定义信号的作用域单位

算法

第 1 行 构造对信号的贡献，并且

第 5—7 行 用一个限定符构造对子信号的贡献，该限定符表示信号定义的作用域单位。

make-process-dict(*pdef*, *level*)(*dict*) ≙ **(5.1.3)**

```
1  (let mk-Process-definition1(nm, inst, f, pset, sigset, tp, synset, vset, , tset, graph) = pdef in
2   let mk-Number-of-instances1(init, maxi) = inst,
3       pid = mk-Identifier1(level, nm),
4       level' = level ⌢ ⟨mk-Process-qualifier1(nm)⟩ in
5   let parm = ⟨mk-Identifier1(level', s-Variable-name1(f[i])) | 1 ≤ i ≤ len f⟩ in
6   let parmd = [(parm[i], VALUE) ↦ mk-VarDD(, s-Sort-reference-identifier1(f[i]), nil, ) |
7                1 ≤ i ≤ len f] in
8   let dict' = extract-sortdict(tp, level')(dict + parmd) in
9   let dict'' = merge {make-entity(entity, level')(dict') |
10                      entity ∈ (pset ∪ sigset ∪ synset ∪ vset ∪ tset)} in
11  let mk-Process-graph1(, stateset) = graph in
12  let nodeset = union {statenodeset | mk-State-node1(, , statenodeset) ∈ stateset} in
13  let insigset = {sigid | mk-Input-node1(sigid, , ) ∈ nodeset} in
14  let localreach = (pid, insigset, ⟨⟩) in
15  if is-wf-decision-answers(graph, level)(dict) then
16    dict'' + [(pid, PROCESS) ↦ mk-ProcessDD(parm, init, maxi, graph, {localreach})]
17   else
18    exit("§2.7.5: 判定动作中的回答不是互斥的")
```

type: *Process-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

目的 返回进程和它的所有定义对实体字典 *entitydict* 的贡献

参数

pdef AS_1 进程定义

level 一个限定符，用来表示定义进程的作用域单位。

算法

第 2 行 抽取实例的初始数（*init*）和实例的最大数（*maxi*）。

第 3 行 构造 *Identifier*$_1$，用来表示进程。

第 4 行 构造限定符，用来表示进程的作用域单位。

第 5 行 构造各形式参数的标识符 *Identifier*$_1$。

第 6 行 构造形式参数对实体字典 *Entity-dict* 的贡献。注意，它们被当作普通的变量来处理。

第 8 行 用描述符来更新实体字典 *entitydict*。该描述符是进程中定义的 *Data-type-definition*$_1$ 的描述符。

第 9—10 行 构造对所包含的过程定义（*pset*）、信号定义（*sigset*）、同义类型定义（*synset*）、变量定义（*vset*）和定时器定义（*tset*）的贡献。

第 11 行 设 *stateset* 表示进程状态的集合。

第 12 行 设 *nodeset* 表示进程输入节点的集合。

第 13 行 设 *insigset* 表示在进程的一个输入节点中收到的信号集合。

第 14 行　　设本地可达性 *localreach* 表示一个可达者 *Reachability*，此可达者用来安排到本进程类型实例的信号路由。

第 15 行　　进程图中的判定动作所具有的回答必须是相互排斥的。

第 16 行　　并且用进程自身的描述符来更新所构造的实体字典 *entitydict*。请注意，在这个阶段，进程的可达者 *Reachability* 集合仅包含用于安排到本进程类型的实例的信号路由的可达者 *Reachability*。

make-procedure-dict(*procdef*, *level*)(*dict*) ≙ **(5.1.4)**

```
1  (let mk-Procedure-definition1(nm, fp, pset, tp, sset, vset, graph) = procdef in
2   let level' = level ⌢ ⟨mk-Procedure-qualifier1(nm)⟩ in
3   let (fparml, fdict) = make-formal-parameters(fp, level) in
4   let pid = mk-Identifier1(level, nm) in
5   let dict' = extract-sortdict(tp, level')(dict + fdict) in
6   let dict'' = merge {make-entity(entity, level')(dict') | entity ∈ (pset ∪ sset ∪ vset)} in
7   if is-wf-decision-answers(graph, level)(dict) then
8     dict'' + [(pid, PROCEDURE) ↦ mk-ProcedureDD(fparml, graph)]
9    else
10    exit("§2.7.5: 判定动作中的回答不是互斥的")
```

type: *Procedure-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

目的　　返回一个过程和它的所有定义对实体字典 *entitydict* 的贡献

参数

procdef　　AS$_1$ 过程定义

level　　一个限定符，表示定义该过程的作用域单位

算法

第 2 行　　构造一个表示该过程的作用域单位的限定符。

第 3 行　　构造有关形式参数是 IN 还是 IN/OUT (*fparml*) 的信息和构造这些形式参数的实体字典 *entitydict* 描述符 (*fdict*)。

第 4 行　　构造一个表示该作用域单位的限定符，该作用域单位就是该过程。

第 5 行　　与进程的做法相同（见上面）。

第 6 行　　构造用于所包含的过程定义 (*pset*)、同义类型定义 (*sset*) 和变量定义 (*vset*) 的描述符。

第 7 行　　过程图中的判定动作所具有的回答必须是相互排斥的。

第 8 行　　构造过程自身的描述符。

make-formal-parameters(*parml*, *level*) ≜ (5.1.5)

```
1  (if parml = ⟨⟩ then
2     (⟨⟩, [])
3   else
4    (let (parmrest, drest) = make-formal-parameters(tl parml, level) in
5     let id = mk-Identifier1(level, s-Variable-name1(hd parml)) in
6     let (p, d) =
7          cases hd parml:
8           (mk-In-parameter1(, sort)
9              → (mk-InparmDD(id), [(id, VALUE) ↦ mk-VarDD(, sort, nil, )]),
10          mk-Inout-parameter1(, )
11             → (mk-InoutparmDD(id), [])) in
12    (⟨p⟩ ⌢ parmrest, d + drest)))
```

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

目的

（递归地）构造并返回一个描述符表，这些描述符包含各个形式参数是 IN 还是 IN/OUT 参数的信息；也返回它们的实体 *entity* 描述符。

参数

parml	AS_1 过程的形式参数
level	一个限定符，用来表示该过程的作用域单位

算法

第 1 行	如果是在形式参数表的末端，则什么都不返回。
第 4 行	构造形参表的其余部分的描述符。
第 5 行	构造形参表中第一个参数的标识符 *Identifier*$_1$。
第 6—10 行	构造参数描述符，以及构造第一个参数对实体字典 *entitydict* 的贡献，并使之与表中其余部分对实体字典 *entitydict* 的贡献汇接。返回第一个参数的实体字典 *entitydict* 描述符并使之与表的其余部分的描述符汇接。

make-block-dict(*bdef*, *level*)(*dict*) ≜ (5.1.6)

```
1  (let mk-Block-definition1(bnm, pdefs, sigdefs, , , datatype, syntype, sub) = bdef in
2   let level' = level ⌢ (mk-Block-qualifier1(bnm)) in
3   let sortd = extract-sortdict(datatype, level')(dict) in
4   let dict' = sortd + merge {make-entity(entity, level')(sortd) | entity ∈ (sigdefs ∪ syntype ∪ pdefs)} in
5   if sub = nil then
6     dict'
7    else
8     (let mk-Block-substructure-definition1(snm, bdefs, , , sdefs, tp, syndefs) = sub in
9      let level'' = level' ⌢ (mk-Block-substructure-qualifier1(snm)) in
10     let sortd' = extract-sortdict(tp, level'')(dict') in
11     sortd' + merge {make-entity(entity, level'')(sortd') | entity ∈ (bdefs ∪ sdefs ∪ syndefs)}))
```

type: *Block-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

目的

构造并返回在一功能块中定义的那些实体的实体字典 *entitydict* 描述符。注意，这里不处理所包围的信号路由定义、信道定义、连接等等。

参数

bdef	一个 AS_1 功能块定义
level	此功能块的定义限定符

算法

第 1 行　　分解该功能块定义。

第 2 行　　构造表示该功能块的限定符。

第 3 行　　更新实体字典 *entitydict*，以包括在该功能块中定义的数据类型定义 $Data\text{-}type\text{-}definition_1$。

第 4 行　　更新实体字典 *entitydict*，以包括在该功能块中定义的信号（*sigdefs*）、同义类型（*syntype*）和进程（*pdefs*）。

第 5 行　　如果没有规定功能块子结构，则返回该功能块对实体字典 *Entity-dict* 的贡献。

第 8 行　　分解该功能块子结构。

第 9 行　　构造一个限定符以表示该功能块子结构的层次。

第 10 行　　更新实体字典 *entitydict*，以包括在该功能块子结构中定义的数据类型定义 $Data\text{-}type\text{-}definition_1$。

第 11 行　　返回这个被更新的实体字典 *entitydict*，它与来自各功能块（*bdefs*）、各信号（*sdefs*）和各同义类型（*syndefs*）的描述信息放在一起。

$$\textit{is-wf-decision-answers}(\textit{graph}, \textit{level})(\textit{dict}) \triangleq \tag{5.1.7}$$

```
1  (let (starttrans, stateset) =
2        cases graph:
3         (mk-Procedure-graph1(init, stset) → (init, stset),
4          mk-Process-graph1(init, stset)   → (init, stset)) in
5   let trans = s-Transition1(starttrans) in
6   is-wf-transition-answers(trans, level)(dict) ∧
7   (∀mk-State-node1(, , inputs) ∈ stateset)
8    ((∀input ∈ inputs)(is-wf-transition-answers(s-Transition1(input), level)(dict))))
```

type：　(*Procedure-graph*$_1$ | *Process-graph*$_1$) *Qualifier*$_1$ → *Entity-dict* → *Bool*

目的　　检查一过程或进程图的判定动作中的回答是否互相排斥

参数

graph　　过程或进程图

level　　表示过程或进程的限定符 $Qualifier_1$

结果　　如果成功，结果为真。

算法

第 1—4 行　　设 *starttrans* 表示图的起始节点，设状态集 *stateset* 表示图中的各状态。

第 5 行　　设 *trans* 表示初始的跃迁。

第 6 行　　初始跃迁的判定中的回答必须是相互排斥的，并且

第 7 行　　对于每一个状态，它必须保持该状态中的每个输入节点。

第 8 行　　输入节点中的跃迁包含在判定中相互排斥的回答。

is-wf-transition-answers(**mk-***Transition*$_1$(*trans*,), *level*)(*dict*) ≜ **(5.1.8)**

```
 1  (∀mk-Decision-node1(, answerset, elsetrans) ∈ elems trans)
 2   ((elsetrans ≠ nil ⊃ is-wf-transition-answers(elsetrans, level)(dict)) ∧
 3    (∀mk-Decision-answer1(answer1, trans1), mk-Decision-answer1(answer2, trans2) ∈ answerset)
 4     (is-wf-transition-answers(trans1, level)(dict) ∧
 5      is-wf-transition-answers(trans2, level)(dict) ∧
 6      (answer1 ≠ answer2 ∧ is-Range-condition1(answer1) ∧ is-Range-condition1(answer2) ⊃
 7       (let mk-Range-condition1(orid, cset1) = answer1,
 8            mk-Range-condition1(, cset2) = answer2 in
 9        (∀term ∈ Ground-expression1)
10         ((let dict' = dict + [SCOPEUNIT ↦ level] in
11           trap exit with true in
12           let answerterm1 = eval-ground-expression(make-valuetest-operator(term, orid, cset1))(dict'),
13               answerterm2 = eval-ground-expression(make-valuetest-operator(term, orid, cset2))(dict') in
14           ¬is-equivalent(answerterm1, answerterm2, level)(dict)))))))
```

type: *Transition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Bool*

目的　检查在跃迁中的每一个判定动作是否包含相互排斥的回答

参数

trans　该跃迁中的动作

level　表示外围作用域单位的限定符

算法

第 1 行　对于动作表中的每一个判定节点，第 2 行至第 14 行必须成立

第 2 行　else 部分中的跃迁必须包含相互排斥的回答，并且

第 3 行　对于判定中的每两个分枝，第 4 行至 14 行必须成立

第 4—5 行　两个分枝中的跃迁必须包含互相排斥的回答，并且

第 6 行　如果是不同的分枝，并且它们在回答中都包含形式正文，则

第 7—8 行　设 *orid* 表示 OR 运算符的标识符 *Identifier*$_1$，*cset*1 表示当前两个分枝之一的范围条件，*cset2* 表示另一分枝的范围条件。

第 9 行　对于每一个基项（*term*），必须满足：

第 12—13 行　从 *term* 中导出的基项（*answerterm*1）和第一个条件集合（第 12 行）一定不能与从 *term* 导出的基项（*answerterm2*）和第二个条件集合同处一个等价类中。函数 *make-valuetest-operator* 返回一个基项表达式 *Ground-expression*。由函数 *eval-ground-expression* 计算此表达式以得到一个基项。由于判定被解释之前，隐含于同义类型的范围检查将不能进行，所以任何来自于函数 *eval-ground-expression* 的 *exit* 都要被捕捉到（第 11 行）。

5.2　抽象数据类型的处理

这节包含处理抽象数据类型的函数。对项目进行处理的函数有：

extract-sortdict　构造实体字典 *entitydict* 期间要用此函数。它产生类型描述符、类别描述符、字面值描述符和运算符描述符。

reduce-term “简化项函数”，当一个项被传送到另一个作用域单位时，例如实在参数的传递、对非局部变量的赋值等等，就要使用此函数。

is-equivalent “是等价函数”，当应该比较两个项时，例如在条件表达式、范围检查和判定节点中，就要用到此函数。

$$extract\text{-}sortdict(typedef, l)(dict) \triangleq \tag{5.2.1}$$

```
1  (let mk-Data-type-definition1(tnm, union, sorts, signatureset, eqs) = typedef in
2   let tid = mk-Identifier1(l, tnm) in
3   let (psmap, eqs') =
4       if union = {} then
5         ([], {})
6        else
7         (let tid' ∈ union in
8          let mk-TypeDD(pmap, equa) = dict((tid', TYPE)) in
9          (pmap, equa)) in
10  let literald =
11      [mk-Identifier1(l, s-Literal-operator-name1(lit)) ↦ mk-OperatorDD(⟨⟩, s-Result1(lit)) |
12       lit ∈ signatureset ∧ is-Literal-signature1(lit)],
13      operatord =
14      [mk-Identifier1(l, s-Operator-name1(op)) ↦ mk-OperatorDD(s-Argument-list1(op), s-Result1(op)) |
15       op ∈ signatureset ∧ is-Operator-signature1(op)] in
16  let dict' = dict + [(id, VALUE) ↦ literald(id) | id ∈ dom literald] +
17      [(id, VALUE) ↦ operatord(id) | id ∈ dom operatord] in
18  let sortd = [(id, SORT) ↦ mk-SortDD(tid) | id ∈ sorts] in
19  let sortset = {mk-Identifier1(l, nm) | nm ∈ (sorts ∪ dom psmap)},
20      sortmap = [sort ↦ make-equivalent-classes(sort)(dict') | sort ∈ sortset] in
21  let equations = eqs ∪ eqs' in
22  let sortmap' = eval-equations(sortmap, equations)(dict) in
23  let dict'' = dict' + sortd + [(tid, TYPE) ↦ mk-TypeDD(sortmap', equations)] in
24  if (∃{mk-Ground-term1(t), mk-Error-term1()} ⊂ union rng sortmap')(is-Identifier1(t)) then
25    exit(“§5.4.1.7：字面值等于错误项”)
26   else
27    if is-wf-values(psmap, sortmap')(dict'') then
28      dict''
29     else
30      exit(“Z.100§5.2.1：包围作用域单位的等价类的生成或减少”))
```

type: *Data-type-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

目的 更新实体字典 *entitydict* 以包含数据类型定义 *Data-type-definition*$_1$ 的描述符以及它所包含的类别、运算符与字面值的描述符

参数

typedef 一个数据类型定义 *Data-type-definition*$_1$

l 代表它被定义所在的层次

结果 更新过的实体字典 *entitydict*

算法

第 2 行 构造数据类型的标识符 *Identifier*$_1$。

第 3—9 行 抽取父辈的（包围的）数据类型定义的类别映射表 *Sortmap* 和等式 *Equations*$_1$。如果在 *Type-union*$_1$ 中没有出现类型标识符 *Type-identifier*$_1$，则它是系统层次，否则在父辈的描述符中可以得到父辈的类别映射表 *Sortmap* 和等式 *Equations*$_1$。

第 10—12 行　构造在数据类型定义中定义的所有字面值的描述符。它们被看作是没有任何变元的运算符。

第 13—16 行　构造在数据类型定义中定义的所有运算符的描述符，把它们加入到现存的实体字典 *Entity-dict* 并给予它们实体类 **VALUE**。

第 18 行　构造在数据类型定义中定义的所有类别的描述符（*sortmap*）。

第 19—20 行　构造由每个类别的等价类组成的初始的类别映射表 *Sortmap*；其中，每个类别的等价类数目与该类别的项一样多，即，每个等价类包含一个并且仅包含一个项。*sortmap* 的域是局部定义的类别（*sorts*）和包围的作用域单位（**dom** *psmap*）的类别。

第 22 行　按照这些等式修改类别映射表 *Sortmap*。

第 24—25 行　字面值标识符的基项和错误项一定不能同属一个等价类。

第 23 行　用局部类别的描述符和数据类型定义的描述符来更新实体字典 *entitydict*。

第 27 行　如果数据类型与包围的作用域单位的数据类型一致（即，没有改变值），则返回更新过的实体字典 *entitydict*。

$$\textit{is-equivalent}(\textit{lterm}, \textit{rterm}, \textit{level})(\textit{dict}) \triangleq \qquad \textbf{(5.2.2)}$$

```
1  (let (id, TYPE) ∈ dom dict be s.t. s-Qualifier1(id) = level in
2   let mk-TypeDD(sortmap, ) = dict((id, TYPE)) in
3   let termsets = union rng sortmap in
4   let ltermset ∈ termsets be s.t. lterm ∈ ltermset,
5       rtermset ∈ termsets be s.t. rterm ∈ rtermset in
6   if mk-Error-term1() ∈ (ltermset ∪ rtermset)
7     then exit ("§5.4.1.7: 运算符应用等于错误项")
8     else ltermset = rtermset)
```

type: *Ground-term*$_1$ *Ground-term*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Bool*

目的　检测两个项是否属于同一个等价类

参数

lterm，*rterm*　两个项，其等价性需要测试

level　表示作用域单位的限定符，将在此范围中完成测试

结果　如果它们是等价的，则结果为真。

算法

第 1 行　抽取由 *level* 表示的作用域单位的类型标识符 *Type-identifier*$_1$。

第 2—3 行　构造作用域单位中所有可见类别的全部等价类的集合（在整个系统中，所有等价类是不相交的）。

第 4—5 行　抽取 *lterm* 的等价类和 *rterm* 的等价类。

第 6 行　这些等价类中都不包括错误项。

第 8 行　如果 *lterm* 和 *rterm* 属于同一个等价类，就返回真。

reduce-term(*sortid*, *term*, *level*)(*dict*) ≜ (5.2.3)

```
 1  if term = nil then
 2    nil
 3   else
 4    (let sortid' = if is-SortDD(dict((sortid, SORT))) then
 5                     sortid
 6                   else
 7                     s-Parent-sort-identifier1(dict((sortid, SORT))) in
 8     let mk-SortDD(tpid) = dict((sortid', SORT)) in
 9     let mk-TypeDD(sortmap, ) = dict((tpid, TYPE)) in
10     let (tpid', q) ∈ dom dict be s.t. q = TYPE ∧ s-Qualifier1(tpid') = level in
11     let mk-TypeDD(sortmap', ) = dict((tpid', q)) in
12     let vset ∈ sortmap'(sortid') be s.t. term ∈ vset in
13     let vset' ∈ sortmap(sortid') be s.t. vset' ⊆ vset in
14     let term' ∈ vset' in
15     term')
```

type: *Sort-reference-identifier*$_1$ [*Ground-term*$_1$] *Qualifier*$_1$ → *Entity-dict* → [*Ground-term*$_1$]

目的 把一个项转换到相同等价类的另一个项，使得所选的项仅包含定义 *sortid* 的作用域单位中所定义的字面值和运算符。每次当一个值被传送到一个没有被围住的作用域单位时都要求进行这种转换。(为了简化处理，每次当一个值（可能）被传送到另一个作用域单位时就进行这种转换，即，在赋值、实在参数的计算中，等等)

参数

sortid	类别引用标识符 *Sort-reference-identifier*$_1$，它表示要转换的项的类别
term	要被转换的项 *Term*$_1$
level	作用域单位，在其中 *term* 被使用

结果 是新的项。如果 *term* 是 ***nil***，结果就是 ***nil***（如果简化项函数 *reduce-term* 被用在一个求实在参数值的函数中，而其中实在参数未被指定，则 *term* 为 nil)。

算法

第 1 行	如果没有规定 *Term*$_1$，则返回 ***nil***。
第 4—7 行	如果 *sortid* 表示一个同义类型标识符 *Syntype-identifier*$_1$ (*SyntypeDD*)，则抽取父辈的类别标识符 *Sort-identifier*$_1$。
第 8 行	抽取定义此类别的类型标识符 *Type-identifier*$_1$ (*tpid*)。
第 9 行	抽取此类型的类别映射表 *Sortmap* (*sortmap*)。
第 10—11 行	抽取在使用 *term* 的作用域单位中定义的类型的类别映射表 *Sortmap* (*sortmap'*)。
第 12 行	在使用 *term* 的作用域单位中定义了类别。在属于该类别的等价类中间，抽取包含 *term* 的等价类。
第 13 行	抽取定义此类别 (*sortid'*) 的作用域单位的等价类，*sortid'* 包含在其它等价类中。
第 14 行	返回来自于这个等价类的任意一个元素。

is-wf-values(*survmap*, *vmap*)(*dict*) ≜ **(5.2.4)**

```
1  if survmap = [] then
2    is-wf-bool-and-pid(dict, vmap(dict(PIDSORT)))
3   else
4    (∀id ∈ dom survmap)
5     ((let survset = survmap(id),
6           vset = vmap(id) in
7       (∀class ∈ vset)((∃!class' ∈ survset)(class' ⊆ class))))
```

type: *Sortmap Sortmap* → *Entity-dict* → *Bool*

目的 检测包围作用域单位的等价类的集合是否与被包围的作用域单位的集合相同

参数

survmap 表示包围作用域单位的值的类别映射表 *Sortmap*

Vmap 表示被包围的作用域单位的类别映射表 *Sortmap*

结果 如果成功，则结果为真。

算法

第 1—2 行 如果包围作用域单位的 *sortmap* 为空，则它就是系统层，并且必须满足：布尔 True 和 False 属于不同的等价类，而且存在有无穷多个 PiD 值。在嵌套的作用域单位中，这些检查只是由函数 *is-wf-values* 来检查的更一般条件的特殊情况。

第 4—7 行 对所有属于包围作用域单位的类别标识符 *Sort-identifier*$_1$，该条件必须成立。

第 5 行 抽取应用在包围作用域单位中的现有类别的等价类。

第 6 行 抽取应用在被包围的作用域单位中的现有类别的等价类。

第 7 行 对于被包围的作用域单位中的每一个等价类，必须满足：它包括包围作用域单位的一个唯一的等价类的所有的项。

make-equivalent-classes(*sort*)(*dict*) ≜ **(5.2.5)**

```
1  (let termset = {term ∈ Ground-term1 | is-of-this-sort(sort, term)(dict)} in
2   let classes = {{term} | term ∈ termset} ∪ {{mk-Error-term1()}} in
3   [sort ↦ classes])
```

type: *Sort-identifier*$_1$ → *Entity-dict* → *Sortmap*

目的 构造由 *sort* 表示的一个类别标识符 *Sort-identifier*$_1$ 的所有可能的项 *Term*$_1$，并构造 *sort* 在类别映射表 *Sortmap* 中的项目，在映射表中，多个 *Term*$_1$ 被放入不同的等价类中。

结果 类别 *sort* 对类别映射表 *Sortmap* 的贡献

算法

第 1 行 构造由基项 *Ground-Term*$_1$ 组成的集合，包括该类别的所有可能的项。

第 2 行 把该集合内所有的项放入不同的等价类中，并把错误项放入它自己的等价类中。

is-of-this-sort(*sort*, *t*)(*dict*) ≜ **(5.2.6)**

```
 1  (let mk-Ground-term1(term) = t in
 2   if is-Identifier1(term) then
 3     (let entry = (term, VALUE) in
 4      entry ∈ dom dict ∧
 5      is-OperatorDD(dict(entry)) ∧
 6      s-Argument-list(dict(entry)) = ⟨⟩ ∧
 7      s-Result(dict(entry)) = sort)
 8    else
 9     if is-Conditional-term1(term) then
10       false
11      else
12       (let (id, arglist) = term in
13        let entry = (id, VALUE) in
14        if entry ∈ dom dict ∧ is-OperatorDD(dict(entry)) then
15          (let mk-OperatorDD(sortlist, result) = dict(entry) in
16           len arglist = len sortlist ∧
17           result = sort ∧
18           (∀i ∈ ind arglist)(is-of-this-sort(sortlist[i], arglist[i])(dict)))
19         else
20          false))
```

type : *Sort-identifier*$_1$ *Ground-term*$_1$ → *Entity-dict* → *Bool*

目的 检测一个给定的项 *Term*$_1$ *t* 是否是一个由 *sort* 表示的类别的项

算法

第 2 行 如果该项是一个标识符，则

第 4 行 此标识符必须能在实体字典 *entitydict* 中找到

第 5--6 行 作为一个字面值（即，作为一个没有变元的运算符）

第 7 行 并且结果类别必须等于 *sort*

第 9 行 如果该项是一个条件项，则它不表示一个值（但是条件项内的后继和替换则表示值）

第 12 行 如果该项是一个运算符项，则

第 14 行 如果在 *entitydict* 中能够找到该项内的标识符，并且它表示一个运算符，则

第 16 行 描述符内的变元数必须与该项内出现的变元数相等

第 17 行 并且在该描述符中的结果类别必须等于 *sort*

第 18 行 按照该描述符中的变元类别表，在 *term* 内的每个变元项都必须具有适当的类别。

eval-equations(*sortmap*, *equations*)(*dict*) ≜ **(5.2.7)**

```
1  (let trueterm = dict(TRUEVALUE),
2       falseterm = dict(FALSEVALUE) in
3   let quanteq = {eq ∈ equations | is-Quantified-equations1(eq)} in
4   let rest = equations \ quanteq in
5   let unquant = union {eval-quantified-equation(sortmap, eq) | eq ∈ quanteq} in
6   let rest' = expand-conditional-term-in-equations(rest ∪ unquant, trueterm, falseterm) in
7   let rest'' =
8       union {if is-Conditional-equation1(eq)
9                 then expand-conditional-term-in-conditions({eq}, trueterm, falseterm)
10                else {eq} | eq ∈ rest'} in
11  let unquanteqs = {eq ∈ rest'' | is-Unquantified-equation1(eq)},
12      condeqs = {eq ∈ rest'' | is-Conditional-equation1(eq)} in
13  let sortmap' = eval-unquantified-equations(sortmap, unquanteqs) in
14  eval-conditional-equations(sortmap', condeqs))
```

type: *Sortmap Equations*$_1$ → *Entity-dict* → *Sortmap*

目的 根据一组等式，减少在给定的作用域单位中可见类别的等价类数

参数

sortmap 一个类别映射表，它包含的等价类数目将被减少

equations 一组等式

结果 修改过的类别映射表 *Sortmap*

算法

第 1—2 行 从实体字典 *Entity-dict* 中抽取布尔字面值 True 和 False 的 AS_1 表示法。

第 3 行 抽取被量化的等式。

第 5 行 把这组量化的等式变成一组非量化的等式。

第 6 行 把所有出现在修改过的那组等式内的条件项（除了那些出现在条件等式的条件中的条件项以外）变成一组条件等式。

第 7—10 行 把在条件集合中包含条件项的所有的条件等式变成一个在条件中没有任何条件项的条件等式的集合（见跟在函数 *expand-conditional-term-in-conditions* 后面的正文中的例子）。

第 11—12 行 把等式的结果集合（*rest''*）分裂成一个非量化等式的集合和一个条件等式的集合。

第 13 行 修改 *sortmap* 使之与未量化的等式集合一致。

第 14 行 返回类别映射表 *Sortmap*，该表是与条件等式集合一致的修改过的 *sortmap*。

eval-unquantified-equations(*sortmap*, *equations*) $\triangleq$ **(5.2.8)**

```
 1  (if equations = {} then
 2     sortmap
 3   else
 4    (let eq ∈ equations in
 5     let mk-Unquantified-equation1(lterm, rterm) = eq in
 6     let sort ∈ dom sortmap be s.t. (∃termset ∈ sortmap(sort))(lterm ∈ termset) in
 7     let termset1 be s.t. termset1 ∈ sortmap(sort) ∧ lterm ∈ termset1 in
 8     let termset2 be s.t. termset2 ∈ sortmap(sort) ∧ rterm ∈ termset2 in
 9     if termset1 = termset2 then
10       eval-unquantified-equations(sortmap, equations \ {eq})
11      else
12       (let newset = sortmap(sort) \ {termset1, termset2} ∪ {termset1 ∪ termset2} in
13        let sortmap' = sortmap + [sort ↦ newset] in
14        let sortmap'' = eval-deduced-equivalence(sortmap') in
15        eval-unquantified-equations(sortmap'', equations \ {eq}))))
```

type: *Sortmap Equations*$_1$ $\rightarrow$ *Sortmap*

目的 修改 *sortmap*（等价类），使之与等式 *equations* 一致

参数

sortmap	要被修改的类别映射表 *Sortmap*
equations	一组非量化的等式

算法

第 1 行	当全部处理完时，返回修改过的 *sortmap*
第 4—5 行	从（剩余的）等式之一中抽取两个项 *Terms*。
第 6 行	抽取 *lterm* 的类别（它与 *rterm* 的类别一样）。
第 7 行	抽取包含 *lterm* 的等价类。
第 8 行	抽取包含 *rterm* 的等价类。
第 9 行	如果这两个项表示相同的等价类，则不更新 *sortmap*，否则
第 12 行	定义一个新的等价类集合，在这个集合中，这两个等价类已经被合并。
第 13 行	修改 *sortmap*，以包含新的等价类的集合。
第 14 行	通过使用由等式得到的信息来减少等价类数。
第 15 行	对其余的等式重复该运算。

eval-deduced-equivalence(*sortmap*) ≙ (5.2.9)

```
 1  if (∃class1, class2, class3 ∈ union rng sortmap)
 2     (class1 ≠ class2 ∧
 3      (∃term1, term2 ∈ class3)((∃term ∈ class1)(replace-term(term, term1, term2) ∈ class2))) then
 4     (let (class1, class2, class3) be s.t. {class1, class2, class3} ⊂ union rng sortmap ∧
 5           class1 ≠ class2 ∧
 6           (∃term1, term2 ∈ class3)((∃term ∈ class1)(replace-term(term, term1, term2) ∈ class2)) in
 7      let sort be s.t. {class1, class2} ⊂ rng sortmap(sort) in
 8      let classes = sortmap(sort) in
 9      let classes' = classes \ {class1, class2} ∪ {class1 ∪ class2} in
10      let sortmap' = sortmap + [sort ↦ classes'] in
11      eval-deduced-equivalence(sortmap'))
12    else
13     sortmap
```

type: *Sortmap* → *Sortmap*

目的 如果一个类别的两个项是在同一个等价类中，则利用此信息来减少类别的等价类数。

参数

sortmap 包含要修改的那些等价类的类别映射表 *Sortmap*

结果 某些类别的等价类的数目已经减少了的类别映射表 *Sortmap*

算法

第 1 行 如果在类别映射表 *Sortmap* 中存在 *class*1、*class*2、*class*3 三个等价类，使得 *class*1 和 *class*2 不相交（*class*3 可以等于 *class*1 或 *class*2 或者表示另一个等价类，甚至于是属于另一个类别），并且在 *class*3 中存在两项（*term*1 和 *term*2），使得在取自于 *class*1 的一个项（*term*）中，用 *term*2 替换 *term*1 时，获得 *class*2 中的一个项，则

第 4—13 行 把 *class*1 和 *class*2 合并为一个等价类。

第 4—6 行 设 *class*1、*class*2、*class*3 表示三个这样的等价类。

第 7 行 设 *sort* 表示 *class*1 和 *class*2 的类别。*class*1 和 *class*2 不能是不同的类别，就象 1—3 行那样。在那个情况下，将不能满足。

第 8—10 行 形成一个新的 *sortmap*，其中该类别的两个等价类已经被合并。

第 11 行 重复运算（用修改过的 *sortmap*）直到不再有等价类可以被合并为止。

replace-term(*term*, *oldterm*, *newterm*) ≙ (5.2.10)

```
 1  if term = oldterm then
 2    newterm
 3   else
 4    if is-Identifier1(term) then
 5      term
 6     else
 7      (let (opid, arglist) = term in
 8       if (∃i ∈ ind arglist)(replace-term(arglist[i], oldterm, newterm) ≠ arglist[i]) then
 9         (let i ∈ ind arglist be s.t. replace-term(arglist[i], oldterm, newterm) ≠ arglist[i] in
10          let arglist' = ⟨arglist[n] | 1 ≤ n < i⟩ ⌢
11              ⟨replace-term(arglist[i], oldterm, newterm)⟩ ⌢
12              ⟨arglist[n] | i < n ≤ len arglist⟩ in
13          (opid, arglist'))
14        else
15         term)
```

type: *Ground-term*$_1$ *Ground-term*$_1$ *Ground-term*$_1$ → *Ground-term*$_1$

目的 用 *newterm* 替换在 *term* 中的 *oldterm* 的一次出现并返回修改过的 term

算法

第 1 行 如果整个项等于 *oldterm*，则返回新的 *term*。

第 4 行 如果此项是一个标识符（并且与 *oldterm* 不同），则不作替换，否则

第 7 行 该项是一个运算符项（由于 *term* 取自于一个等价类，所以不会出现条件项）。设 *op* 表示此运算符标识符，并设 *arglist* 表示变元表。

第 8 行 如果存在一个包含 *oldterm* 的变元，则

第 9 行 设 *i* 表示包含 *oldterm* 的变元的标引号。

第 10—12 行 构造变元表，在该表的元素 *i* 中，一个 *oldterm* 已经由 *newterm* 替换。

第 13 行 返回修改过的项。

第 15 行 如果在变元表中没有出现 *oldterm*，则不改变该项。

eval-quantified-equation(*sortmap*, *quanteqs*) ≙ (5.2.11)

```
 1  (let mk-Quantified-equations1(nmset, sortid, equations) = quanteqs in
 2   let nm ∈ nmset in
 3   let mk-Identifier1(level, snm) = sortid in
 4   let valueid = mk-Identifier1(level ⌢ ⟨mk-Sort-qualifier1(snm)⟩, nm) in
 5   let allterms = union sortmap(sortid) \ {mk-Error-term1()} in
 6   let equations' = union {union {insert-term(sortmap, eq, valueid, term) | term ∈ allterms} |
 7                           eq ∈ equations} in
 8   if nmset = {nm} then
 9     equations'
10    else
11     (let quanteq = mk-Quantified-equations1(nmset \ {nm}, sortid, equations') in
12      eval-quantified-equation(sortmap, quanteq)))
```

type: *Sortmap Quantified-equations*$_1$ → *Equations*$_1$

目的 把一个量化的等式扩展成一组非量化的等式

参数

sortmap 是包围的数据类型定义的类别映射表 *Sortmap*，在该 *Sortmap* 中，各项（仍然）在不同的等价类中。

quanteqs 量化了的等式

结果 一个非量化的等式结果的集合

算法

第 2 行 在量化的等式中取出一个值名。

第 4 行 构造对应于该值名的值标识符。

第 5 行 构造一个集合（*allterms*），它是由量化的类别的所有可能项（除了 *Error-term*$_1$ 之外）组成的。

第 6—7 行 从包含在被量化的等式中的等式集合构造一个非量化等式的集合，办法是用 *allterms* 中的每个项来替换等式集合中的值标识符。

第 8 行 如果在等式中已经替换了每一个值名，则返回这些等式（*equations'*），否则

第 11—12 行 对于量化等式中的其他值名也进行同样的处理。

insert-term(*sortmap*, *equation*, *vid*, *term*) ≙ **(5.2.12)**

```
1   cases equation:
2    (mk-Unquantified-equation1(term1, term2)
3       → {mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
4                                    insert-term-in-term(term2, vid, term))},
5    mk-Quantified-equations1(,,)
6       → (let equations = eval-quantified-equation(sortmap, equation) in
7          union{insert-term(sortmap, eq, vid, term) | eq ∈ equations}),
8    mk-Conditional-equation1(eqs, eq)
9       → (let mk-Unquantified-equation1(term1, term2) = eq,
10            eqs' = union{insert-term(sortmap, e, vid, term) | e ∈ eqs} in
11         let eq' = mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
12                                             insert-term-in-term(term2, vid, term)) in
13         {mk-Conditional-equation1(eqs', eq')}),
14   T → {equation})
```

type: *Sortmap Equation*$_1$ *Value-identifier*$_1$ *Ground-term*$_1$ → *Equations*$_1$

目的 在被一个量化的等式包围的等式中，用一个基项 *Ground-term*$_1$ 来替换一个值名

参数

Sortmap 是一个类别映射表 *Sortmap*。如果（轮到的）等式含有量化了的等式，就要用此映射表。

equation 要修改的等式

vid 应该被替换的值标识符

term 是 *Term*$_1$，应该由它来替换 *vid*。

结果 是一个包含修改过的等式的等式集合。如果该等式是一个量化了的等式，该集合可包含不止一个等式。

算法

第 2—4 行 如果这是一个非量化的等式，则在两个所包含的项中（*term*$_1$，*term*$_2$）用 *term* 来替换 *vid*。

第 5—7 行　　如果它是一个量化的等式，则先把它扩展成一组非量化等式，然后替换该组中每一个等式的值标识符。

第 8—13 行　　如果它是一个条件等式，则由限制条件中每个等式中的项和被限制的等式中的项来替换值标识符，并构造和返回一个集合，此集合包含有修改过的条件等式。

第 14 行　　如果它是非形式的正文，则别管它。

insert-term-in-term(*term*, *vid*, *vterm*) $\triangleq$ **(5.2.13)**

```
 1  (if is-Ground-term1(term) ∨ is-Error-term1(term) then
 2     term
 3    else
 4     (let mk-Composite-term1(term′) = term in
 5      if is-Identifier1(term′) then
 6        if term′ = vid then vterm else term
 7       else
 8        if is-Conditional-term1(term′) then
 9          (let mk-Conditional-term1(cond, t1, t2) = term′ in
10           let cond′ = insert-term-in-term(cond, vid, vterm),
11               t1′ = insert-term-in-term(t1, vid, vterm),
12               t2′ = insert-term-in-term(t2, vid, vterm) in
13           let term″ = mk-Conditional-term1(cond′, t1′, t2′) in
14           if is-Ground-term1(cond′) ∧ is-Ground-term1(t1′) ∧ is-Ground-term1(t2′) then
15             mk-Ground-term1(term″)
16            else
17             mk-Composite-term1(term″))
18         else
19          (let (opid, arglist) = term′ in
20           let arglist′ =
21               ⟨insert-term-in-term(arglist[i], vid, vterm) | 1 ≤ i ≤ len arglist⟩ in
22           if (∃arg ∈ elems arglist)(is-Composite-term1(arg)) then
23             mk-Composite-term1((opid, arglist′))
24            else
25             mk-Ground-term1((opid, arglist′)))))
```

type:　$Term_1$ *Value-identifier*$_1$ *Ground-term*$_1$ $\rightarrow$ $Term_1$

目的　　在一个项（*term*）中，用一个（基）项（*vterm*）来替换一个值标识符（*vid*）。

参数

term　　其值标识符应被替换的 $Term_1$

vid　　要被替换的值标识符

vterm　　应该插入的 $Term_1$，用以取代值标识符

结果　　修改过的项

算法

第 1 行　　如果它是一个基项或是一个错误项，则不要修改它。

第 5—6 行　　如果它是一个标识符并等于 *vid*，则返回新的项，否则不要修改它。

第 8—13 行　　如果它是一个条件项，则构造这个条件项，使得所包含的三个项中出现的 *vid* 都用 *vterm* 来替换。

第 14－17 行　如果三个所包含的项全变成了基项，则把新的条件项作为一个基项返回，否则把它作为一个合成的项返回。

第 19－25 行　否则 *term* 必须是一个运算符项，在这个情况下，变元项中的 *vid* 由 *vterm* 替换；并且如果所有修改过的变元项已经变成了基项，则把这个新的运算符项作为一个基项返回，否则把它作为一个合成的项返回。

expand-conditional-term-in-equations(*equations*, *trueterm*, *falseterm*) ≙ **(5.2.14)**

```
1  (if equations = {} then
2     {}
3   else
4    (let eq ∈ equations in
5     let (condset, eq') =
6          cases eq:
7           (mk-Unquantified-equation1(,)
8              → ({}, eq),
9            mk-Conditional-equation1(condeq, eqs)
10             → (condeq, eqs)) in
11    let mk-Unquantified-equation1(t1, t2) = eq' in
12    let (t1', t1'', cond1) = expand-conditional-in-terms(t1),
13        (t2', t2'', cond2) = expand-conditional-in-terms(t2) in
14    if cond1 = nil ∧ cond2 = nil then
15      {eq} ∪ expand-conditional-term-in-equations(equations \ {eq}, trueterm, falseterm)
16     else
17      (let (cond, term, nterm1, nterm2) be s.t. (cond, term, nterm1, nterm2) ∈
18           {(cond2, t1, t2', t2''), (cond1, t2, t1', t1'')} ∧ cond ≠ nil in
19       let eq1 = mk-Unquantified-equation1(cond, trueterm),
20           eq2 = mk-Unquantified-equation1(cond, falseterm) in
21       let condeq1 =
22           mk-Conditional-equation1(condset ∪ {eq1}, mk-Unquantified-equation1(term, nterm1)),
23           condeq2 =
24           mk-Conditional-equation1(condset ∪ {eq2}, mk-Unquantified-equation1(term, nterm2)) in
25       let equations' = equations ∪ {condeq1, condeq2} \ {eq} in
26        expand-conditional-term-in-equations(equations', trueterm, falseterm))))
```

type: *Equations*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ → *Equations*$_1$

目的　用两个条件等式 *Conditional-equation*$_1$ 来替换每一个条件项 *Conditional-term*$_1$。

例如：

等式

```
if a then b else c==d
```

被扩展为

```
a==True==>b==d;
a==False==>c==d;
```

参数

equations　要被替换的等式的集合

trueterm, *falseterm*　表示布尔 True 和 False 的两个基项

结果　　修改过的等式集合，这些等式不包含任何条件项 *Conditional-terms*

算法

第 1 行　　当等式的集合为空时，什么都不返回。

第 4—9 行　　从该集合中取一个等式并抽取限制条件集合（*condset*）和被限制的等式的集合（*eq′*）。如果它是一个非量化的等式，则限制条件的集合为空。

第 12—13 行　　修改被限制等式中的项。*cond*1 和 *cond*2 是要被检测的条件。如果该项不包含任何条件项，则条件为 ***nil***。在原来的项（*t*1，*t*2）中，当一个条件项已经被该条件项的“then”部分替换时，则得到 *t*1′，*t*2′。当一个条件项已经被该条件项的“else”部分所替换时，则得到 *t*1″，*t*2″。

第 14—15 行　　如果这两项都不包含条件项，则不改变该等式，并继续处理 *equations* 中的另一个等式。

第 17 行　　选择两项之一去处理。在这个调用中将不改变另一个项。

第 19—20 行　　构造两个非量化的等式，它们必须对两个修改过的等式成立。

第 21—23 行　　构造两个条件等式，其中已经分别增加了 *eq*1 和 *eq*2 作为一个附加条件。（*condeq*1）包含一个等式，其中原始项之一（*t*1 或 *t*2）在这个等式中已经被一个包含“then”部分的项替换。（*Condeq*2）包含一个等式，其中原始项之一在这个等式中已经被一个包含“else”部分的项替换。

第 26 行　　在要被考虑的剩余等式的集合中加入这两个新的条件等式（因为 *eq* 中的一个项还未被扩展，还因为已扩展的项又可以包含条件项）。

expand-conditional-in-terms(t) $\triangleq$ (5.2.15)

```
1  (if is-Error-term1(t) then
2     (t, t, nil)
3   else
4    (let mk-Ground-term1(term) = t in
5     cases term:
6      (mk-Identifier1(,)
7         → (t, t, nil),
8       mk-Conditional-term1(cond, t1, t2)
9         → (t1, t2, cond),
10      (id, arglist)
11        → if (∃arg ∈ elems arglist)
12             ((let (,,cond) =
13                  expand-conditional-in-terms(arg) in
14               cond ≠ nil)) then
15            (let (i, t1, t2, cond) be s.t. i ∈ ind arglist ∧
16                 cond ≠ nil ∧
17                 expand-conditional-in-terms(arglist[i]) = (t1, t2, cond) in
18             let arglist' =
19                 ⟨arglist[n] | 1 ≤ n < i⟩ ⌢ ⟨t1⟩ ⌢ ⟨arglist[n] | i < n ≤ len arglist⟩,
20                 arglist'' =
21                 ⟨arglist[n] | 1 ≤ n < i⟩ ⌢ ⟨t2⟩ ⌢ ⟨arglist[n] | i < n ≤ len arglist⟩ in
22             (mk-Ground-term1((id, arglist')), mk-Ground-term1((id, arglist'')), cond))
23           else
24            (t, t, nil))))
```

type: $Term_1 \rightarrow Term_1\ Term_1\ [Ground\text{-}term_1]$

目的 把一个项（t）分成三个项。如果 t 不包含一个条件项，则前两个项是不相关的，且第三个项为 *nil*。否则结果是修改过以包含“then”部分的 t、修改过以包含“else”部分的 t 和布尔条件项。

结果 三个新的项

算法

第 1—6 行 如果它是一个错误项，则不修改它，并且返回 **nil** 作为条件项以表明它不包含一个条件项。

第 8 行 如果这是一个条件项，则返回它的三个部分。

第 10—14 行 如果它是一个运算符项并且它的变元之一包含一个条件项，则

第 15—17 行 取一个包含一条件项的变元项并把它拆开。i 是变元表中的位置。

第 18—20 行 构造对应于“then”部分的变元表（*arglist'*）和对应于“else”部分的变元表（*arglist''*），并且

第 22 行 返回对应于“then”部分、“else”部分的两个运算符项和该变元中条件项内的布尔条件。

expand-conditional-term-in-conditions(*equations*, *trueterm*, *falseterm*) ≙ **(5.2.16)**

```
1  (if equations = {} then
2     {}
3    else
4     (let eq ∈ equations in
5      let mk-Conditional-equation1(condset, eq') = eq in
6      if (∃cond ∈ condset)
7         ((let mk-Unquantified-equation1(t1, t2) = cond in
8           let (,,cond1) =
9               expand-conditional-in-terms(t1),
10              (,,cond2) =
11              expand-conditional-in-terms(t2) in
12          cond1 ≠ nil ∨ cond2 ≠ nil)) then
13        (let (condeq, cond, term, nterm1, nterm2) be s.t. condeq ∈ condset ∧
14            (let mk-Unquantified-equation1(t1, t2) =
15                condeq in
16             let (t1', t1'', cond1) =
17                expand-conditional-in-terms(t1),
18                (t2', t2'', cond2) =
19                expand-conditional-in-terms(t2) in
20             (cond, term, nterm1, nterm2) = (if cond1 = nil
21                                               then (cond2, t1, t2', t2'')
22                                               else (cond1, t2, t1', t1''))) in
23        let eq1 = mk-Unquantified-equation1(cond, trueterm),
24            eq2 = mk-Unquantified-equation1(cond, falseterm) in
25        let condset' = condset \ {condeq} ∪ {eq1, mk-Unquantified-equation1(term, nterm1)},
26            condset'' = condset \ {condeq} ∪ {eq2, mk-Unquantified-equation1(term, nterm2)} in
27        let equations' = equations \ {eq} ∪ {mk-Conditional-equation1(condset', eq'),
28                                            mk-Conditional-equation1(condset'', eq')} in
29        expand-conditional-term-in-conditions(equations', trueterm, falseterm))
30       else
31        {eq} ∪ expand-conditional-term-in-conditions(equations \ {eq}, trueterm, falseterm)))
```

type : *Conditional-equation*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ → *Equations*$_1$

目的

把 *equations* 中的条件等式分裂为两个条件等式，如果在 *Restriction*$_1$ 中它们包含任何条件项的话。

举例：

等式

if b then c else d==e==>f==g

被扩展为

b==True, c==e==>f==g;

b==False, d==e==>f==g

参数

equations	条件等式的集合
trueterm, *falseterm*	表示布尔 True 和 False 的两个基项

结果

扩展了的等式的集合

算法

第 1 行　　当全部处理完后，返回空集合。

第 4—12 行　　从该集合中取一个条件等式，如果在限制部分中不包含一个条件项，则继续处理该集合中其余的等式（第 31 行）。

第 13—21 行　　从限制的集合中抽取非量化的等式，该集合包含条件项（*condeq*）、条件项中的条件（*cond*）、包含条件项的非量化等式中的该项的“then”形式（*nterm*1）、包含条件项的非量化等式中的该项的“else”形式（*nterm*2）以及非量化等式的其它项（*term*）。

第 23—24 行　　构造两个附加限制，并把它们包含在对应的限制集合中。

第 25—26 行　　构造两个修改过的限制集合。

第 27 行　　用等式集合中的两个新的条件等式替换旧的条件等式。

第 29 行　　用修改过的等式集合重复上述运算。

***eval-conditional-equations*(*sortmap*, *condequations*)** ≙ **(5.2.17)**

```
1  if (∃condeq ∈ condequations)(restriction-holds(condeq, sortmap)) then
2    (let condeq ∈ condequations be s.t. restriction-holds(condeq, sortmap) in
3     let mk-Conditional-equation1(, eq) = condeq in
4     let sortmap' = eval-unquantified-equations(sortmap, {eq}) in
5     eval-conditional-equations(sortmap', condequations \ {condeq}))
6   else
7    sortmap
```

type: *Sortmap Conditional-equation*$_1$**-set** → *Sortmap*

目的　　根据一个作用域单位的条件等式减少类别映射表 *Sortmap* 中等价类的数目

参数

sortmap　　一个类别映射表 *Sortmap*

Condequations　　一个条件等式的集合

结果　　修改过的 *Sortmap*

算法

第 1 行　　如果存在一个成立的条件等式，则

第 2 行　　设 *condeq* 表示该成立的条件等式

第 3—4 行　　用被限制的等式（*eq*）的性质来更新 *Sortmap*。

第 5 行　　重复运算，直到剩余集合中没有成立的条件等式为止。

***restriction-holds*(mk-*Conditional-equation*$_1$(*eqs*,), *sortmap*)** ≙ **(5.2.18)**

```
1  (let termpairs = {{term1, term2} | (∃eq ∈ eqs)(mk-Unquantified-equation1(term1, term2) = eq)} in
2   (∀pairs ∈ termpairs)((∃class ∈ union rng sortmap)(pairs ⊂ class)))
```

type: *Conditional-equation*$_1$ *Sortmap* → *Bool*

目的　　检测一个条件等式的限制集合是否成立

参数

eqs	限制的集合
sortmap	用于检查限制是否成立的类别映射表 *Sortmap*

结果 如果成功，则结果为真。

算法

第 1 行 构造一个以项的偶对为元素的集合，每个偶对中包含限制集合中的一个限制的左边项和右边项。

第 2 行 如果各个限制保证右边项与左边项都同处于一个等价类中，则这些限制成立。

$$\textit{is-wf-bool-and-pid}(\textit{dict}, \textit{pidvalueset}) \triangleq \qquad \textbf{(5.2.19)}$$

```
1  (let trueterm = dict(TRUEVALUE),
2       falseterm = dict(FALSEVALUE),
3       mk-Identifier1(level, ) = trueterm in
4   (∀s ∈ Sortmap)(s ⊂ pidvalueset ⊃ (∃n ∈ N1)(n > card s)) ∧
5   ¬is-equivalent(trueterm, falseterm, level)(dict))
```

type: *Entity-dict Term-class-set* → *Bool*

目的 检测布尔 True 是否与布尔 False 属于同一个等价类，并检测是否存在无穷多个 PiD 值

结果 如果满足了这些条件，则结果为真。

算法

第 1—2 行 构造对应于字面值 True 和 False 的基项。

第 3 行 设 *level* 表示它的限定词。

第 4 行 PiD 值的集合必须是无穷大的，即对每个（有限的）*pidvalueset* 的子集合 s，必须存在一个 N_1 的值 n，使得 n 大于该子集合的基数。

第 5 行 布尔字面值 True 一定不能与布尔字面值 False 属于同一个等价类。

5.3 一致性子集合的选择

select-consistent-subset(*bset*, *subset*, *level*) ≙ (5.3.1)

```
 1  if bset = {} then
 2    {}
 3   else
 4    (let block ∈ bset in
 5     let rest = select-consistent-subset(bset \ {block}, subset, level) in
 6     let mk-Block-definition1(bnm, pdefs, , , , , sub) = block in
 7     let pset = {mk-Identifier1(level ⌢ ⟨mk-Block-qualifier1(bnm)⟩, s-Process-name1(pdef)) |
 8                 pdef ∈ pdefs} in
 9     let bid = mk-Identifier1(level, bnm) in
10     if bid ∉ subset then
11       exit("§3.2.1.: 子功能块不在一致性子集中")
12      else
13       if sub = nil then
14         rest ∪ pset
15        else
16         (let mk-Block-substructure-definition1(subnm, bset', , , , ) = sub in
17          let level' =
18              level ⌢ ⟨mk-Block-qualifier1(bnm), mk-Block-substructure-qualifier1(subnm)⟩ in
19          if mk-Identifier1(level, subnm) ∈ subset then
20            rest ∪ select-consistent-subset(bset', subset, level')
21           else
22            if pset = {} then
23              exit ("§3.2.1: 叶功能块不包含进程")
24             else
25              rest ∪ pset))
```

type: *Block-definition*$_1$-**set** *Block-identifier*$_1$-**set** *Qualifier*$_1$ → *Process-identifier*$_1$-**set**

目的

检查所给的功能块标识符的集合与功能块子结构标识符的集合是否表示一个一致性的子集合，并返回该一致性子集合中所包含的进程标识符。函数递归地遍历系统定义。

参数

bset 对于系统定义或功能块子结构定义的功能块定义的集合

subset （假定的）一致性子集合，用一组功能块标识符和功能块子结构标识符组成的集合来表示

level 包含功能块集合 *bset* 的作用域单位的限定符 *Qualifier*$_1$。

算法

第 1 行 当全部处理完时，返回进程标识符的空集合。

第 4 行 设 *block* 表示下一个要考虑的功能块的定义。

第 5 行 为剩下的功能块定义选择一致性子集合。

第 6 行 设 *bnm* 表示功能块名，设 *pdefs* 表示进程定义的集合，设 *sub* 表示任选的功能块子结构定义。

第 7 行 设 *pset* 表示对应于 *pdefs* 的进程标识符 *Process-Identifier*$_1$ 的集合。

第 9—11 行 此功能块（或子功能块）必须在一致性子集合内。

第 13 行 如果功能块中没有子结构，则

第 14 行 功能块中的进程在一致性子集合中。

第 16 行 如果规定了一个子结构，则设 *subnm* 表示它的名字，并设 *bset'* 表示它的功能块定义集合。

第 17—20 行　如果此子结构在一致性子集合中，则考虑子结构中的功能块，否则
第 22—23 行　功能块中至少有一个进程定义。

5.4 通信路径的构造

这一节包含一些函数，用来更新每个进程描述符（*ProcessDD*），以包括一个可达者 *Reachability* 的集合。

对于每个作用域单位，如果它包含两个功能块之间的信道，则构造信道的进来路径和出去路径，连接这些路径并最后更新进程描述符。这些进程描述符来自于出去路径，与功能块中所包含的进程相联系。进来路径和出去路径（部分路径），在把它们连接起来之前，在一个端点上包含一个信道，而在另一个端点上包含一个信号路由。中间的标识符都是子信道标识符。函数 *make-structure-paths* 是 *extract-dict* 中要用到的项目函数。

make-structure-paths(*bset*, *cset*, *level*)(*dict*) ≜ **(5.4.1)**

```
 1  (if cset = {} then
 2     dict
 3    else
 4     (let ch ∈ cset in
 5      let mk-Channel-definition1(nm, mk-Channel-path1(b1, b2, ), ) = ch in
 6      if (b1 = ENVIRONMENT ∨ b2 = ENVIRONMENT) ∧ ¬is-System-qualifier1(level[len level]) then
 7        make-structure-paths(bset, cset \ {ch}, level)(dict)
 8       else
 9        (let chid = mk-Identifier1(level, nm) in
10         let (reachset1, dict') = out-going-paths(chid, b1, bset, ⟨⟩)(dict) in
11         let (reachset1', dict'') = out-going-paths(chid, b2, bset, ⟨⟩)(dict') in
12         let reachset2 = in-coming-paths(chid, b2, bset, ⟨⟩)(dict) in
13         let reachset2' = in-coming-paths(chid, b1, bset, ⟨⟩)(dict) in
14         if is-consistent-refinement(reachset1, reachset2) ∧
15           is-consistent-refinement(reachset2, reachset2') then
16            (let d = update-processd(reachset1, reachset2)(dict'') in
17             let d' = update-processd(reachset1', reachset2')(d) in
18              make-structure-paths(bset, cset \ {ch}, level)(d'))
19           else
20            exit ("Z.100 § 3.3: 信道的非法具体化"))))
```

type: *Block-definition*$_1$-**set** *Channel-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

目的　对于和两个功能块或系统环境相连接的一个作用域单位中的全部信道，更新能够经过信道发送信号的那些进程的可达集 *Reachabilities*

参数

bset　功能块定义
cset　作用域单位的信道定义
level　表示作用域单位的限定符

结果　*entitydict*，其中需要更新的 *ProcessDD* 描述符已经被更新

算法

第 1 行　当全部处理完时，返回更新过的实体字典 *entitydict*。

第 4—5 行　从剩下来的定义集合中取出一个信道定义。

第 6—7 行　如果信道是一个子信道，则什么都不做，因为子信道是由函数 *incoming-path* 和函数 *outgoing-path* 处理的。

第 10—11 行　抽取可达集 *Reachabilities*，它包含那些能经过信道发送的进程，还包含适当的路径 *Path* 和被更新的实体字典 *entitydict*。该 *entitydict* 分别用在 *b*1（第 10 行）和 *b*2（第 11 行）中的局部通信路径的可达集 *Reachabilities* 的信息来更新。

第 12—13 行　抽取可达集 *Reachabilities*，它包含分别在功能块 *b*2 和 *b*1 中能经过信道来接收信号的那些进程，还包含适当的路径 *Path*。

第 14 行　对于两个方向，任何具体化子集合的选择都必须是一致的。

第 16 行　用可达集 *Reachabilities* 来更新分别在 *reachset*1 和 *reachset*1′ 中的进程描述符。此可达集包含分别从 *reachset*2 和 *reachset*2′ 中推出的可能的接收者。

第 18 行　对其余信道定义的处理与上面相同。

$$\textit{is-consistent-refinement}(\textit{reachset1}, \textit{reachset2}) \triangleq \qquad \textbf{(5.4.2)}$$

```
1  (let sigset1 = {sig | (∃(, sset, ) ∈ reachset1)(sig ∈ sset)},
2       sigset2 = {sig | (∃(, sset, ) ∈ reachset2)(sig ∈ sset)} in
3   let env1 = card reachset1 = 1 ∧ (∃(p, , ) ∈ reachset1)(p = ENVIRONMENT),
4       env2 = card reachset2 = 1 ∧ (∃(p, , ) ∈ reachset2)(p = ENVIRONMENT) in
5   ¬(∃mk-Identifier1(qual1, ), mk-Identifier1(qual2, nm2) ∈ sigset1)
6     (len qual1 > len qual2 ∧ qual2 ⌢ ⟨mk-Signal-qualifier1(nm2)⟩ = ⟨qual1[i] | 1 ≤ i ≤ len qual2 + 1⟩) ∧
7   ¬(∃mk-Identifier1(qual1, ), mk-Identifier1(qual2, nm2) ∈ sigset2)
8     (len qual1 > len qual2 ∧ qual2 ⌢ ⟨mk-Signal-qualifier1(nm2)⟩ = ⟨qual1[i] | 1 ≤ i ≤ len qual2 + 1⟩) ∧
9   (env1 ∨ env2 ∨ sigset1 = sigset2))
```

type：　*Reachabilities Reachabilities* → *Bool*

目的　检查一个信道的每个端点的信号路由中的信号是否不包括相同信号在不同的具体化层次上的信号，并检查来自于信道的出去端点的信号集合是否与进入端点上的信号集合相同。

参数

*reachset*1　信道的出去端点的可达集 *Reachabilities*

*reachset*2　信道的进入端点的可达集 *Reachabilities*

结果　如果满足上面所描述的条件，则结果为 **true**。

算法

第 1 行　设 *sigset*1 表示信道的出去端点处的信号集合。

第 2 行　设 *sigset*2 表示信道的进入端点处的信号集合。

第 3 行　如果信道的出去端点是系统环境，则设 *env*1 为 **true**。

第 4 行　如果信道的进入端点是系统环境，则设 *env*2 为 **true**。

第 5—6 行　对每两个出去信号，必须满足：它们一定不能互为子信号。

第 7—8 行　对每两个进入信号，必须满足：它们一定不能互为子信号。

第 9 行　必须满足：出去信号的集合等于进来信号的集合，除非有一个端点是系统环境。

out-going-paths(*chid*, *b*, *bset*, *path*)(*dict*) ≜ **(5.4.3)**

```
1  if b = ENVIRONMENT then
2    ({(ENVIRONMENT, ⟨chid⟩)}, dict)
3   else
4    (let mk-Identifier1(level, bnm) = b in
5     let bdef ∈ bset be s.t. s-Block-name1(bdef) = bnm in
6     let mk-Block-definition1(,,, connects, srdefs,,, sub) = bdef in
7     let path' = path ⌢ ⟨chid⟩ in
8     let bqual = level ⌢ ⟨mk-Block-qualifier1(bnm)⟩ in
9     if (∃(mk-Identifier1(qual,), PROCESS) ∈ dom dict)(qual = bqual) then
10      (let mk-Channel-to-route-connection1(ch, routeset) ∈ connects be s.t. ch = chid in
11       let (rset, dict') = make-out-reaches(routeset, srdefs, path')(dict) in
12       (rset, dict'))
13      else
14       (let mk-Block-substructure-definition1(, bset', connects', cset,,,) = sub in
15        let mk-Channel-connection1(cid, cidset) ∈ connects' be s.t. cid = chid in
16        let dict' = make-structure-paths(bset', cset, level)(dict) in
17        make-out-connect-paths(cidset, cset, bset', path')(dict')))
```

type: *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT)
Block-identifier$_1$-**set** *Path* →
Entity-dict → *Reachability*-**set** *Entity-dict*

目的

构造可达集 *Reachabilities*，它对应于从一功能块流出经过一个给定信道（*chid*）的信号。该信道是可达集 *Reachabilities* 中路径 *Path* 的一部分。所构造的（暂时的）可达集 *Reachabilities* 与进程描述符中的可达集 *Reachabilities* 不同，因为 *Path* 仅是通信路径的一个部分（缺少目的部分），并且因为可达集 *Reachabilities* 中的进程标识符 *Process-identifier*$_1$ 是发送进程。补函数 *incoming-path* 构造“相反的”可达集 *Reachability*，在此可达集中，路径 *Path* 是起始部分。在函数 *update-processd* 中，两个可达集 *Reachabilities* 被合并，形成插入到发送进程的描述符中的可达集 *Reachabilities*。函数 *outgoing-path* 也更新进程描述符，但仅用局部于信道从之起始的功能块的可达集 *Reachabilities* 去更新。由于几个信道可以起始于同一个功能块，因此进程描述符可能用相同的局部的可达集 *Reachabilities* 来更新几次，但是这无关紧要，因为可达集 *Reachabilities* 是一个集合。

参数

chid	信道标识符，函数从它构造路径 *Path*
b	功能块标识符，信道起始于该功能块
bset	功能块定义的集合，在这些定义中可以找到该功能块
path	到现在为止所构造的路径，它离开包围的功能块（如果信道不是一个子信道，该路径仅包含 *chid*）

结果

（暂时的）可达集 *Reachabilities* 和用局部于功能块的通信路径更新过的实体字典 *entitydict*

算法

第 1—2 行　如果起始端点是系统环境，则返回可达者 *Reachability*，它包含环境 **ENVIRONMENT** 作为起始端点，并返回未改变的 *dict*。

第 4—6 行　从 *bset* 中抽取对应于功能块标识符 *b* 的功能块定义。

第 7 行　把信道标识符（*chid*）加入到 *Path* 中（*Path* 表示从一个与子信道不同的信道到 *chid* 的路径）。

第 8 行　　设 *bqual* 表示功能块 *bnm* 中定义的实体的限定符。

第 9—12 行　　如果选择该功能块中的进程，则返回包含这些进程（*rset*）的可达集 *Rechabilities* 并返回用一些进程可达集 *Reachabilities* 更新过的实体字典 *Entity-dict*，这些进程能把信号发送给功能块中其它的进程。

第 14—15 行　　分解功能块子结构定义和连接此信道（*chid*）的连接点。

第 16 行　　用包含路径 *Path* 的可达集 *Reachabilities* 来更新实体字典 *Entity-dict*，这些 *Path* 是局部于功能块子结构的。注意，如果有几个信道被连接到该功能块上，就用相同的可达集 *Reachabilities*（无害地）更新该进程描述符几次。

第 17 行　　通过进入子功能块继续建立可达集 *Reachabilities*，这些子功能块与子信道 *cidset* 相连接。

***make-out-connect-paths*(*cidset*, *cset*, *bset*, *path*)(*dict*)** ≙ **(5.4.4)**

```
1  (if cidset = {} then
2     ({}, dict)
3   else
4    (let cid ∈ cidset in
5     let (reachsetrest, dictrest) = make-out-connect-paths(cidset \ {cid}, cset, bset, path)(dict) in
6     let cdef ∈ cset be s.t. s-Channel-name1(cdef) = s-Name1(cid) in
7     let mk-Channel-definition1(, mk-Channel-path1(b1, b2, ), ) = cdef in
8     let block = if b2 = ENVIRONMENT then b1 else b2 in
9     let (rset, dict') = out-going-paths(cid, block, bset, path)(dictrest) in
10    (reachsetrest ∪ rset, dict')))
```

type: *Channel-identifier*$_1$-**set** *Channel-definition*$_1$-**set** *Block-definition*$_1$-**set** *Path* → *Entity-dict* → *Reachability*-**set** *Entity-dict*

目的　　构造临时的可达集 *Reachabilities*，它对应于由一个功能块子结构的子信道传递的那些信号。它的补函数是 *make-in-connect-paths*。

参数

cidset　　子信道的集合

cset　　功能块子结构的信道定义的集合

bset　　功能块子结构的功能块定义的集合

path　　从功能块子结构引向不是子信道的第一个信道的（部分的）*Path*

结果　　同函数 *out-going-paths* 的结果

算法

第 1 行　　当全部处理完时，不返回可达集 *Reachabilities* 和信号，但返回未改变的实体字典 *Entity-dict*（结果是递归地建立的）。

第 4—5 行　　从子信道的集合中取一个信道（*cid*）并（递归地）构造其余子信道的可达集 *Reachabilities*。

第 6—7 行　　抽取对应于当前子信道的信道定义。

第 8 行　　抽取子信道的起始功能块。

第 9 行　　构造临时的可达集 *Reachabilities*（*rset*），其中的进程是经过信道（*cid*）发送的，而且是包含在功能块（*block*）之中的那些进程。并且路径 *Path* 是用余下的路径来更新的 *path*，这些路径是从信道通向连接到那些进程的信号路由。而且，构造用局部于功能块（*block*）的可达集 *Reachabilities* 来更新的实体字典 *Entity-dict*。

第 10 行　　返回当前的子信道的可达集 *Reachabilities* 和实体字典 *Entity-dict*（与上面所描述的相同），加入到所有其它子信道的可达集 *Reachabilities* 和实体字典 *Entity-dict* 中去。

***in-coming-paths*(*cid*, *block*, *bset*, *path*)(*dict*)** $\triangleq$ **(5.4.5)**

```
1  if block = ENVIRONMENT then
2    {(ENVIRONMENT, ⟨cid⟩)}
3   else
4    (let mk-Identifier1(qual, bnm) = block in
5     let bdef ∈ bset be s.t. s-Block-name1(bdef) = bnm in
6     let mk-Block-definition1(,,, connects, srdefs,,, sub) = bdef in
7     let path' = path ⌢ ⟨cid⟩ in
8     let bqual = qual ⌢ ⟨mk-Block-qualifier1(bnm)⟩ in
9     if (∃(mk-Identifier1(qual,), PROCESS) ∈ dom dict)(qual = bqual) then
10      (let mk-Channel-to-route-connection1(ch, routeset) ∈ connects be s.t. ch = cid in
11       make-in-reaches(routeset, srdefs, path'))
12     else
13      (let mk-Block-substructure-definition1(, bset', connects', cdefs,,,) = sub in
14       let mk-Channel-connection1(ch, cset) ∈ connects' be s.t. ch = cid in
15       make-in-connect-paths(cset, cdefs, bset', path')(dict)))
```

type: *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT)
Block-identifier$_1$-set *Path* → *Entity-dict* → *Reachability*-set

目的　　构造并返回可达集 *Reachabilities*，其中进程是包含在给定功能块中的那些进程。与函数 *out-going-paths* 相反，函数 *in-coming-paths* 构造“真正的”可达集 *Reachabilities*，因为它们包含接收进程。可达集 *Reachabilities* 中的路径 *Path* 是部分的，并且表示从第一个不是子信道的信道到接收进程的路径。路径的“另一端”将在补函数 *out-going-paths* 中构成。

参数

cid	信道，要为它构造可达集 *Reachabilities*
block	信道的目的功能块
bset	功能块定义的集合，其中包含 *block*
path	可能的路径集合

算法

第 1—2 行　　如果目的端点是系统环境，则返回仅包含一个可达者 *Reachability* 的集合，在该可达者 *Reachability* 中，接收者是环境。

第 4—6 行　　抽取并分解对应于功能块标识符 *Block* 的功能块定义。

第 7 行　　把信道标识符加入到将在功能块内部使用的路径中。

第 8 行　　设 *bqual* 表示在功能块 *bnm* 中定义的实体的限定符。

第 9 行　　如果存在一个功能块中定义的一个进程的描述符，则不选择该子结构

第 10—11 行　　抽取并返回可达集 *Reachabilities*，它对应于连接到信道（*ch*）的信号路由（*routeset*）。

第 13—14 行　分解功能块子结构定义并且分解把信道（*chid*）连接到功能块子结构的信道连接。

第 15 行　进入连接到子信道 *cset* 的子功能块（*bset'*）来继续建造可达集 *Reachabilities*。

***make-in-connect-paths*(*cidset*, *cset*, *bset*, *path*)(*dict*)** ≙ **(5.4.6)**

```
1  if cidset = {} then
2    {}
3   else
4   (let cid ∈ cidset in
5    let reachsetrest = make-in-connect-paths(cidset \ {cid}, cset, bset, path)(dict) in
6    let cdef ∈ cset be s.t. s-Channel-name1(cdef) = s-Name1(cid) in
7    let mk-Channel-definition1(, mk-Channel-path1(b1, b2, ), ) = cdef in
8    let block = if b2 = ENVIRONMENT then b1 else b2 in
9    let inblockreach = in-coming-paths(cid, block, bset, path)(dict) in
10   reachsetrest ∪ inblockreach)
```

type: *Channel-identifier*$_1$-**set** *Channel-definition*$_1$-**set**
Block-identifier$_1$-**set** *Path* → *Entity-dict* → *Reachability*-**set**

目的　构造可达集 *Reachabilities*，它对应于由一个功能块子结构的那些子信道传送的信号。其补函数是 *make-out-connect-paths*。

参数

cidset	子信道的集合
cset	功能块子结构的信道定义的集合
bset	该功能块子结构的功能块定义的集合
path	（部分的）*Path*，它从不是子信道的第一个信道通到该功能块子结构

算法

第 1 行　当全部处理完时，没有可达集 *Reachabilities* 可返回（结果是递归地建立的）。

第 4—5 行　从子信道的集合中取出一个子信道标识符，并为余下的子信道（递归地）建造可达集 *Reachabilities*。

第 6—7 行　从对应于当前的子信道标识符的 *cset* 中取出信道定义。请注意，关于信道传递哪些信号的信息未被使用，因为是信号路由决定哪些信号是**实际地**由信道传送的。

第 8 行　抽取目的功能块

第 9 行　构造可达集 *Reachabilities*（*inblockreach*），其中进程是经过信道（*cid*）接收的并且是功能块（*block*）中所包含的那些进程；信号是经过信道（*cid*）和适当的信号路由由进程接收的那些信号；*Path* 是由其余的路径更新过的 *path*，路径是从信道到连接到这些进程的信号路由。

第 10 行　返回当前的子信道的可达集 *Reachabilities*（同上所述），加入到所有其它子信道的可达集 *Reachabilities* 中。

make-in-reaches(*routeset*, *srdefs*, *path*) ≙ **(5.4.7)**

```
1  if routeset = {} then
2    ({}, {})
3   else
4    (let route ∈ routeset in
5     let reachrest = make-in-reaches(routeset \ {route}, srdefs, path) in
6     let mk-Identifier1(, rnm) = route in
7     let mk-Signal-route-definition1(nm, path1, path2) ∈ srdefs be s.t. nm = rnm in
8     let mk-Signal-route-path1(e1, e2, sigset) = path1 in
9     let (signalset, dest) =
10        if e1 = ENVIRONMENT then
11          (sigset, e2)
12         else
13          if path2 = nil then
14            ({}, e1)
15           else
16            (let mk-Signal-route-path1(, , sigset') = path2 in
17             (sigset', e1)) in
18    reachrest ∪ {(dest, signalset, path ⌢ ⟨route⟩)})
```

type : *Signal-route-identifier*$_1$-**set** *Signal-route-definition*$_1$-**set** *Path* → *Reachabilities*

目的 为一个指向一个信号路由连接点的部分路径 *Path* 构造可达集 *Reachabilities*。在该信号路由连接点上有多少个信号路由标识符就要构造同样多的可达集 *Reachabilities*。处理出去信号的补函数是 *make-out-reaches*。

参数

routeset	一个信号路由连接的信号路由标识符集合
srdefs	与它们相应的信号路由定义
path	增加了这些信号路由的路径 *Path*

结果 构造好的可达集 *Reachabilities*

算法

第 1 行	当全部处理完时，没有东西可返回（结果是递归地建立的）。
第 4—5 行	从路由集合 *routeset* 中取出一个信号路由标识符 *Signal-route-identifier*$_1$，并且构造其余信号路由标识符的可达集 *Reachabilities*。
第 6—8 行	抽取并分解对应于此信号路由标识符的信号路由定义。
第 9—17 行	从信号路由定义中抽取进入的信号集合（*signalset*）和目的进程。
第 18 行	返回对应于当前信号路由标识符的可达者 *Reachability*，并把它加入到对应于其余的信号路由标识符的可达集 *Reachabilities* 中。

make-out-reaches(*routeset*, *routedefs*, *path*)(*dict*) ≙ (5.4.8)

```
1  (if routedefs = {} then
2     ({}, dict)
3   else
4    (let route ∈ routedefs in
5     let (restr, restd) = make-out-reaches(routeset, routedefs \ {route}, path)(dict) in
6     let mk-Signal-route-definition1(rnm, mk-Signal-route-path1(p1, p2, sset), path2) = route in
7     if p1 = ENVIRONMENT ∨ p2 = ENVIRONMENT then
8       if (∃id ∈ routeset)(s-Name1(id) = rnm) then
9        (let id ∈ routeset be s.t. s-Name1(id) = rnm in
10        if path2 = nil then
11          if p1 = ENVIRONMENT then
12            (restr, restd)
13           else
14            (restr ∪ {(p1, sset, ⟨id⟩ ⌢ path)}, restd)
15         else
16          (let mk-Signal-route-path1(, , sset') = path2 in
17           let (originp, sset'') =
18              if p1 = ENVIRONMENT then
19                (p2, sset')
20               else
21                (p1, sset) in
22            (restr ∪ {(originp, sset'', ⟨id⟩ ⌢ path)}, restd)))
23       else
24        (restr, restd)
25     else
26      (let mk-Identifier1(level, ) = p1 in
27       (restr, make-local-reach(mk-Identifier1(level, rnm), route)(restd)))))
```

type: *Signal-route-identifier*$_1$-**set** *Signal-route-definition*$_1$-**set** *Path* → *Entity-dict* → *Reachability*-**set** *Entity-dict*

目的

为一个起始于一个信号路由连接点的部分路径 *Path* 构造可达集 *Reachabilities*。所构造的可达集 *Reachabilities* 的数目和信号路由连接点上的信号路由标识符的数目一样多。处理进入信号的补函数是 *make-in-reaches*。此外，更新实体字典 *Entity-dict* 中的那些进程描述符，用对应于进程之间的信号路由的可达集 *Reachabilities* 来更新。如果功能块有几个信号路由连接点，这些进程描述符就要被更新几次。

参数

routeset	一个连接点的信号路由标识符集合
routedefs	此功能块的信号路由定义集合
path	起始于连接点的部分路径 *Path*

结果

构造好的可达集 *Reachabilities* 和用对应于进程之间的信号路由的可达集 *Reachabilities* 更新过的实体字典 *Entity-dict*

算法

第 1 行　考虑功能块中的每一个信号路由定义。当全部处理完时，不返回可达集 *Reachabilities*，而返回未改变的实体字典 *Entity-dict*。

第 4 行　从路由集合 *routeset* 中取出一个信号路由定义，并为其余信号路由定义构造可达集 *Reachabilities* (*restr*) 和更新了的实体字典 *Entity-dict* (*restd*)。

第 7—24 行　把信号路由连接到一个信道的情况包括在内。

第 8 行和第 24 行　如果在该连接点（由 *routeset* 表示的）中没有提到此信号路由，则返回其余信号路由定义的信息（即，对当前的信号路由定义不做任何事）。

第 9 行　从路由集合 *routeset* 中抽取该信号路由的标识符。

第 10—14 行　如果信号路由是单向的，那么如果信号路由是“进入的”（第 11—12 行），就对当前的信号路由定义不做任何事，否则从其余信号路由定义中返回可达集 *Reachabilities*，与一个包含发送进程（*p*1）的可达者 *Reachability* 相汇合，返回由信号路由（*sset*）传送的信号和已经增加了信号路由标识符（*id*）的路径 *Path*。也返回可能更新了的实体字典 *Entity-dict*。

第 16—22 行　如果信号路由是双向的，则从适当的信号路由路径 $signal\text{-}route\text{-}path_1$ 中抽取起始进程（*orig-inp*），然后执行和第 14 行相同的处理。

第 26—27 行　如果信号路由正在连接进程，则不建造新的可达者 *Reachability*，但实体字典 *Entity-dict* 中的进程描述符要更新，用两个进程之间的通信路径的可达集 *reachbilities* 来更新，用函数 *make-local-reach* 来处理。

$$\textit{make-local-reach}(\textit{id}, \textbf{mk-}\textit{Signal-route-definition}_1(\textit{rnm}, \textit{path1}, \textit{path2}))(\textit{dict}) \triangleq \qquad \textbf{(5.4.9)}$$

```
1  (let mk-Signal-route-path1(p1, p2, sset) = path1 in
2   let mk-ProcessDD(parm, init, maxi, graph, inrset) = dict((p1, PROCESS)) in
3   let reach = (p2, sset, ⟨id⟩) in
4   let dict' = dict + [(p1, PROCESS) ↦ mk-ProcessDD(parm, init, maxi, graph, inrset ∪ {reach})] in
5   if path2 = nil then
6     dict'
7    else
8     make-local-reach(id, mk-Signal-route-definition1(rnm, path2, nil))(dict'))
```

type：　$\textit{Signal-route-identifier}_1\ \textit{Signal-route-definition}_1 \rightarrow \textit{Entity-dict} \rightarrow \textit{Entity-dict}$

目的　用另一个进程端点的可达集 *Reachabilities* 来更新实体字典 *Entity-dict* 中的一个或两个进程描述符。仅当信号路由是双向时才更新两个进程描述符。

参数

id	信号路由的标识符
$signal\text{-}route\text{-}definition_1$	信号路由定义包含
rnm	信号路由的名字
*path*1	第一条 $signal\text{-}route\text{-}path_1$
*path*2	第二条（任选的）$Signal\text{-}route\text{-}path_1$

结果　更新了的实体字典 *Entity-dict*

算法

第 1—4 行　用一个方向上的可达集 *Reachabilities* 更新实体字典 *Entity-dict*。增加到发送进程（*p*1）的进程描述符上的可达者 *Reachability* 包含接收进程（*p*2）、信号集（*sset*）和一个仅包含信号路由标识符（*id*）的路径 *Path*。

第 5—8 行　如果信号路由是单向的，则返回更新了的实体字典 *Entity-dict*，否则就要作同样的处理。这里把信号路由当作单向的，它所包含的路径 $Signal\text{-}route\text{-}path_1$ 是至今尚未处理的那一个。

update-processd(*outrset*, *inrset*)(*dict*) ≜ **(5.4.10)**

```
1  if outrset = {} then
2    dict
3   else
4    (let outreach ∈ outrset in
5     let (pid, sigset, path) = outreach in
6     let inrset' =
7         {inr ∈ inrset | (let (, , path') = inr in
8                         path'[len path'] = hd path)} in
9     let mk-ProcessDD(parmd, init, maxi, graph, rset) = dict((pid, PROCESS)) in
10    let reachabilityset = extract-reachabilities(sigset, path, inrset') in
11    let dict' = dict +
12        [(pid, PROCESS) ↦ mk-ProcessDD(parmd, init, maxi, graph, rset ∪ reachabilityset)] in
13    update-processd(outrset \ {outreach}, inrset)(dict'))
```

type: *Reachability*-**set** *Reachability*-**set** → *Entity-dict* → *Entity-dict*

目的 用可达集 *reachabilities* 来更新实体字典 *Entity-dict* 中的进程描述符。可达集 *reachabilities* 可能来自包含走出的（部分）*Path* 的可达集，也可能来自包含进入的（部分）*Path* 的可达集。

参数

outrset 包含走出的路径 *Path* 的可达集 *Reachabilities*

inrset 包含进入的路径 *Path* 的可达集 *Reachabilities*

结果 更新过的实体字典 *Entity-dict*

算法

第 1 行 检查包含走出路径 *Path* 的每一个可达者 *Reachability*。当处理完该集合时，则返回（更新过的）实体字典 *Entity-dict*。

第 4—5 行 从 *outrset* 集合中取出一个可达者 *reachability*。

第 6 行 抽取那些进入的可达集 *Reachabilities*，它们包含当前可达者 *Reachability* 中的路径 *Path* 的延续部分，也就是说，抽取那些可达集 *Reachabilities*，它们在其路径 *Path* 末端的信道标识符和当前的路径 *Path* 的始端的信道标识符相同。

第 9 行 分解发送进程的进程描述符。

第 10 行 为了构造可能的（完整的）可达集 *Reachabilities*，遍历所有的进入的可达集 *Reachabilities*。

第 11—13 行 用新的可达集 *Reachabilities* 来更新实体字典 *Entity-dict*。在处理其余的走出的可达集 *Reachabilities* 时，要使用此更新了的实体字典 *Entity-dict*。

extract-reachabilities(*sigset*, *path*, *inrset*) ≜ **(5.4.11)**

```
1  if inrset = {} then
2    {}
3   else
4    (let inr ∈ inrset in
5     let (pid, sigset', path') = inr in
6     let reach = if sigset ∩ sigset' = {} then
7                   {}
8                 else
9                   {(pid, sigset ∩ sigset', path ⌢ tl path')} in
10    {reach} ∪ extract-reachabilities(sigset, path, inrset \ {inr}))
```

type: *Signal-identifier*$_1$-**set** *Path* *Reachability*-**set** → *Reachability*-**set**

目的　　从一条走出的（部分的）路径 *Path* 和进入的可达集 *Reachabilities* 构造可达集 *Reacchabilities*

参数

sigset	在走出的路径 *Path* 上的信号集合
path	走出的路径 *Path*
inrset	进入的可达集 *Reachabilities*

结果　　构造好的可达集 *Reachabilities*

算法

第 1 行　　当处理完可达集 *Reachabilities* 时，什么都不返回。

第 4—5 行　　从 *inrset* 中取出一个进入的可达者 *Reachabilitity*。

第 5—6 行　　如果进入的可达者 *Reachability* 和走出的路径 *Path* 没有公共的信号，就构造一个空的可达者 *Reachability*；否则，它包含接收者（*pid*）、进入的可达者 *Reachability* 中和走出的路径 *Path* 中可能的信号的交集、和通过拼接走出的路径（*path*）和除第一个元素外（即，该元素是信道标识符，也作为走出路径中的最后一个元素）的进入路径（*path*）而构造出来的完整路径 *Path*。

第 10 行　　返回构造好的可达者 *Reachability*，同时返回从其余的进入的可达集 *Reachabilities* 构成的可达集 *Reachabilities*。

域索引

函数索引

处理器索引

变量索引

错误信息

§2.7.4：　发现了多个接收者 13
§2.7.4：　没有发现接收者 13
§2.7.5：　判定动作中的回答不是互斥的 47，48
§2.7.5：　无匹配的回答 30

§3.2.1：　叶功能块不包含进程 69
§3.2.1：　子功能块不在一致性子集中 69
§3.3：　信道的非法具体化 70

§5.2.1：　包围作用域单位的等价类的生成或减少 52
§5.4.1.7：　字面值等于错误项 52
§5.4.1.7：　运算符应用等于错误项 53
§5.4.1.9：　值不在同义类型范围内 33，34，38，39，42
§5.5.2.2：　被视见的值未定义 39
§5.5.2.2：　所访问变量的值未定义 38
§5.5.2.3：　条件必须计算出 TRUE 或 FALSE 40
§5.5.4.4：　透露进程未激活 17

中国印刷 ISBN 92-61-03795-X