国 际 电 信 联 盟

ITU-T

国际电信联盟
电信标准化部门

Z.143

(03/2006)

Z系列：电信系统使用的语言和一般性软件情况

正式描述技巧（FDT）— 测试和测试控制记法（TTCN）

测试和测试控制记法第三版（TTCN-3）：操作语义

ITU-T Z.143建议书

ITU-T

ITU-T Z系列建议书

电信系统使用的语言和一般性软件情况

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

欲了解更详细信息，请查阅*ITU-T*建议书目录。

ITU-T Z.143建议书

测试和测试控制记法第三版（TTCN-3）：操作语义

摘　要

本建议书定义了TTCN-3（测试和测试控制记法第三版）的操作语义。对清楚解释TTCN-3制定的规范而言，操作语义是必要的。本建议书基于ITU-T Z.140建议书定义的TTCN-3核心语言。

来　源

ITU-T第17研究组（2005-2008）按照ITU-T A.8建议书规定的程序，于2006年3月16 日批准了ITU-T Z.143建议书。

前　言

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

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

WTSA 第 1 号决议规定了批准建议书须遵循的程序。

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

注

本建议书为简明扼要起见而使用的“主管部门”一词，既指电信主管部门，又指经认可的运营机构。

遵守本建议书的规定是以自愿为基础的，但建议书可能包含某些强制性条款（以确保例如互操作性或适用性等），只有满足所有强制性条款的规定，才能达到遵守建议书的目的。“应该”或“必须”等其它一些强制性用语及其否定形式被用于表达特定要求。使用此类用语不表示要求任何一方遵守本建议书。

知识产权

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

至本建议书批准之日止，国际电联尚未收到实施本建议书可能需要的受专利保护的知识产权的通知。但需要提醒实施者注意的是，这可能并非最新信息，因此特大力提倡他们通过下列网址查询电信标准化局（TSB）的专利数据库：http://www.itu.int/ITU-T/ipr/。

© 国际电联 2006

版权所有。未经国际电联事先书面许可，不得以任何手段复制本出版物的任何部分。

目 录

页码

1 范围 ... 1

2 参考文献 ... 1

3 定义和缩写 ... 1

3.1 定义 ... 1

3.2 缩写 ... 1

4 引言 ... 1

5 本建议书的结构 ... 2

6 限制 ... 2

7 简短格式的替换 ... 2

7.1 替换步骤的次序 ... 3

7.2 全局常量和模块参数的替换 ... 3

7.3 在 alt 语句中嵌入单个接收操作 ... 3

7.4 在 alt 语句中嵌入独立的可选步骤 ... 4

7.5 interleave 语句的替换 ... 4

7.6 trigger 操作的替换 ... 16

8 TTCN-3 的流程图语义 ... 17

8.1 流程图 ... 17

8.2 TTCN-3 行为的流程图表示 ... 22

8.3 TTCN-3 模块的状态定义 ... 27

8.4 消息、程序调用、应答和异常 ... 35

8.5 函数、可选步骤和测试用例的调用记录 ... 37

8.6 TTCN-3 模块的计算程序 ... 38

9 TTCN-3 构件的流程图片段 ... 40

9.1 Action 语句 ... 40

9.2 Activate 语句 ... 40

9.3 Alt 语句 ... 41

9.4 可选步骤调用 ... 47

9.5 Assignment 语句 ... 47

9.6 Call 操作 ... 47

9.7 Catch 操作 ... 53

9.8 Check 操作 ... 54

9.9 Clear 端口操作 ... 57

9.10 Connect 操作 ... 57

9.11 Constant 定义 ... 58

9.12 Create 操作 ... 59

9.13 Deactivate 语句 ... 59

9.14 Disconnect 操作 ... 61

9.15 Do-while 语句 ... 62

9.16 Done 部件操作 ... 63

9.17 Execute 语句 ... 64

9.18 表达式 ... 67

9.18b 流程图片段 <dynamic-error> ... 69

9.19 流程图片段 <finalize-component-init> ... 70

9.20 流程图片段 <init-component-scope> ... 70

9.21 流程图片段 <parameter-handling> ... 71

9.22 流程图片段 <statement-block> ... 71

9.23 For 语句 ... 72

9.24 函数调用 ... 73

9.25 Getcall 操作 ... 77

9.26 Getreply 操作 ... 78

9.27 Getverdict 操作 ... 78

9.28 Goto 语句 ... 79

9.29 If-else 语句 79
9.30 Label 语句 80
9.31 Log 语句 80
9.32 Map 操作 81
9.33 Mtc 操作 81
9.34 Port 声明 82
9.35 Raise 操作 82
9.36 Read 定时器操作 84
9.37 Receive 操作 85
9.38 Repeat 语句 88
9.39 Reply 操作 88
9.40 Return 语句 90
9.41 Running 部件操作 93
9.42 Running 定时器操作 96
9.43 Self 操作 97
9.44 Send 操作 97
9.45 Setverdict 操作 100
9.46 Start 部件操作 100
9.47 Start 端口操作 102
9.48 Start 定时器操作 102
9.49 Stop 部件操作 104
9.50 Stop 执行语句 108
9.51 Stop 端口操作 110
9.52 Stop 定时器操作 110
9.53 System 操作 111
9.54 Timer 声明 111
9.55 Timeout 定时器操作 113
9.56 Unmap 操作 113
9.57 Variable 声明 114
9.58 While 语句 116

10 操作语义部件列表 116
10.1 功能和状态 116
10.2 特殊关键字 118
10.3 TTCN-3 行为描述的流程图 119
10.4 流程图片段 119

ITU-T Z.143建议书

测试和测试控制记法第三版（TTCN-3）：操作语义

1 范围

本建议书定义了 TTCN-3（测试和测试控制记法第三版）的操作语义。本建议书基于 ITU-T Z.140 建议书[1]中定义的 TTCN-3 核心语言。

2 参考文献

下列 ITU-T 建议书和其他参考文献的条款，在本建议书中的引用而构成本建议书的条款。在出版时，所指出的版本是有效的。所有的建议书和其他参考文献均会得到修订，本建议书的使用者应查证是否有可能使用下列建议书或其他参考文献的最新版本。当前有效的 ITU-T 建议书清单定期出版。本建议书引用的文件自成一体时不具备建议书的地位。

[1] ITU-T Recommendation Z.140 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Core language.*

3 定义和缩写

3.1 定义

就本建议书而言，ITU-T Z.140 建议书[1]中给出的术语和定义适用。

3.2 缩写

本建议书采用下列缩写：

ASN.1	抽象语法记法1
BNF	巴科斯范式
IDL	接口描述语言
MTC	主测试部件
SUT	接受测试的系统
TTCN	测试和测试控制记法

4 引言

本节以直观和明确的方式来定义 TTCN-3 行为的含义。操作语义并不意味着是标准的，因此，基于本语义的数学证明能力是非常有限的。

本操作语义提供了关于 TTCN 模块执行的、面向状态的观点。文中介绍了不同的状态，对不同 TTCN-3 结构的含义，通过以下方法来描述：

1） 使用状态信息来定义执行结构的前提条件；以及

2） 定义执行结构将如何改变一个状态。

操作语义受限于 TTCN-3 行为的含义，即函数、可选步骤、测试用例、模块控制以及定义测试行为的语言结构，如 **send** 和 **receive** 操作、**if-else**- 或 **while**-语句。通过用其他语言结构来替换它们，可以解释某些 TTCN-3 结构的含义。例如，**interleave** 语句是一系列内嵌 **alt** 语句的简短格式，而通过用相应系列的内嵌 alt 语句对其进行替换，可以解释各 **interleave** 语句的含义。

在大多数情况下，一种语言的语义定义基于所描述代码的抽象语法树。本语义并不基于抽象语法树，但要求对 TTCN-3 行为的描述以流程图的形式来做图形表示。流程图用于描述函数、可选步骤、测试用例或模块控制中的控制流。TTCN-3 行为描述至流程图的映射是直观的。

注 — TTCN-3 语句至流程图的映射是一个非正式的步骤，不通过使用ITU-T Z.140建议书 [1]中的BNF规则来定义。其原因是BNF规则对直观映射并非是最佳的，因为某些静态语义规则被编码为BNF规则，以便允许在语法检查期间进行静态语义检查。

5 本建议书的结构

本建议书结构上分为四部分：

1） 第一部分（参见第6节）描述操作语义的限制，即与语义相关的问题，这超出了本建议书的讨论范围。

2） 第二部分（参见第7节）定义TTCN-3 简短格式的含义以及用其它TTCＮ-3语言结构对其进行替换的宏记法。TTCN-3模块中的这些替换可以视为在依据以下操作语义描述对模块进行解释之前的预处理步骤。

3） 第三部分（参见第8节）通过解释流程图和修改状态方式来描述TTCN-3的操作语义。

4） 第四部分（参见第9节）规定不同TTCN-3 语句至流程图片段的映射，这为代表函数、可选步骤、测试用例和模块控制的流程图提供了构件块。

6 限制

操作语义仅包括 TTCN-3 的行为方面，即它描述语句和操作的含义。它不提供：

a） TTCN-3数据方面的语义。这包括如编码、解码和从非TTCN-3规范中输入数据的用法等方面的问题。

b） 分组机制的语义。分组与TTCN-3 模块的定义部分相关，不存在任何行为方面的问题。

c） **`import`** 语句的语义。在TTCN-3 模块的定义部分，必须完成定义的引入。操作语义对引入的定义进行处理，就好像它们是在引入模块中定义的那样。

d） 端口参数化的语义。

7 简短格式的替换

在该操作语义可以用来解释 TTCN-3 行为之前，在文本层面上，必须通过相应的完整定义来扩展简短格式。

TTCN-3 简短格式为：

- 模块参数的列表、相同类型的常量与变量的声明，以及定时器声明的列表；
- 单独的接收操作；
- 单独的可选步骤调用；
- **`trigger`** 操作；
- 在函数和测试用例定义结尾处缺少**`return`** 和 **`stop`** 语句；
- 缺少**`stop`** 执行语句；以及
- **`interleave`** 语句。

除了处理简短格式，操作语义要求对模块参数和全局常量进行特别处理，即在模块定义部分中定义的常量。对模块参数和全局常量的所有引用将用具体的值来替换。这意味着，假设模块参数和全局常量的值可以在操作语义变得相关之前予以确定。

注 1 — 操作语义中模块参数和全局常量的处理将有别于TTCN-3 编译器中对它们的处理。操作语义描述TTCN-3 行为的含义，不是用于执行TTCN-3 编译器的指南。

注 2 — 操作语义象对变量那样来处理测试部件、测试用例、函数和模块控制中的参数和局部常量。局部常量或**in**、**out** 和 **inout** 等参数的错误用法必须经过静态检查。

7.1 替换步骤的次序

必须按以下次序来替换简短格式、全局常量和模块参数的原文：

1） 替换模块参数、常量、变量以及带单个声明的定时器声明的列表；

2） 用具体的值来替换全局常量和模块参数；

3） 在**alt**语句中嵌入单独的接收操作；

4） 在**alt**语句中嵌入单独的可选步骤调用；

5） 扩展**interleave**语句；

6） 用等同的**receive**操作和**repeat**语句来替换所有的**trigger**操作。

7） 没有**return**语句时，在函数结尾处增加**return**，没有stop语句时，在测试用例定义结尾处增加**self.stop**操作；

8） 没有**stop**语句时，在模块控制部分结尾处增加stop。

注 — 如果未遵照这一替换步骤次序执行，那么替换的结果将不代表定义的行为。

7.2 全局常量和模块参数的替换

在模块定义部分中声明的常量，对模块控制以及在 TTCN-3 模块执行期间创建的所有测试部件而言是全局的。模块参数在运行时间意为全局常量。

全局常量和模块参数的所有引用应在操作语义开始解释模块之前用实际的值予以替换。如果常量或模块参数的值以表达式的形式给出，那么必须对表达式进行计算。然后，用计算结果替换常量或模块参数的所有引用。

7.3 在alt语句中嵌入单个接收操作

TTCN-3 的接收操作是：**receive**、**trigger**、**getcall**、**getreply**、**catch**、**check**、**timeout** 和 **done** 。

注 — **receive**、**trigger**、**getcall**、**getreply**、**catch** 和**check**等操作工作于端口，归因于消息、程序调用、应答和异常的接收，它们允许分岔。**timeout** 和 **done** 并非真正的接收操作，但它们可以以相同的方式用作接收操作，即作为**alt**语句中的选项。因此，操作语义将像对待接收操作那样来处理**timeout** 和 **done**。

接收操作可以用作函数、可选步骤或测试用例中的单独语句。**timeout** 操作也可用作模块控制中的单独语句。在这种情况下，可以认为接收操作是 **alt** 语句的一种简短形式，它只有一种由接收操作定义的选择。对操作语义而言，嵌入了接收语句的 **alt** 语句将替换接收操作的所有单独事件。

例如：

```
// 单独发生……
   :
  MyCL.trigger(MyType:?);
   :

// 将由……替换
   :
  alt {
      []   MyCL.trigger (MyType:?) { }
  }
   :
// 或
   :
```

```
    MyPTC.done;
     :

  // 将由……替换
     :
    alt {
       []   MyPTC.done { }
    }
     :
```

7.4 在alt语句中嵌入独立的可选步骤

TTCN-3 允许像函数、可选步骤、测试用例和模块控制中的函数那样来调用可选步骤。只带有一个调用可选步骤分支的 **alt** 语句给出了单独调用可选步骤的含义。**alt** 语句负责快照，快照在可选步骤内计算，如果可选步骤中没有选项可供选择，那么 **alt** 语句还负责调用缺省机制。

注 — 在模块控制中使用的可选步骤只能包括一些带**timeout** 操作和一个**else** 分支的选项。

例如：

```
  // 单独发生……
     :
    myAltstep(MyPar1Val);
     :

  // 将由……替换
     :
    alt {
       []   myAltstep(MyPar1Val) { }
    }
     :
```

7.5 interleave语句的替换

interleave 语句的含义通过用一系列具有相同含义的内嵌 **alt** 语句对它的替换来定义。本节描述了 **interleave** 语句的替换结构算法。替换将在语法层面上进行。

在 **interleave** 语句内，不允许：

1） 使用控制转移语句**for**、**while**、**do-while**、**goto**、**activate**、**deactivate**、**stop**、**repeat** 和 **return**；

2） 调用可选步骤；

3） 调用包括通信操作在内的用户定义的函数；

4） 用布尔表达式来防卫**interleave** 语句的分支；以及

5） 指定**else** 分支。

由于这些限制，所有未提及的单独语句（如赋值语句、**log**、**send** 或 **reply**）、*阻塞的* **call** 操作以及 **interleave**、**if-else** 和 **alt** 复合语句，都可以在 **interleave** 语句内使用。

注 1 — 如果它们不存在嵌入的**alt** 语句，那么阻塞的**call**操作和**if-else** 语句可视为单独语句来处理。对嵌入**alt** 语句，选项对**interleave** 语句起作用，并需要做特别处理。为简化起见，以下算法不对这两种情况做区分。

注 2 — interleave 语句之中也允许非阻塞状态的**call** 操作，它们被视为独立语句。

本节中描述的算法只对不带嵌入 **interleave** 语句的 **interleave** 语句起作用。如果 **interleave** 语句中已经嵌入了 **interleave** 语句，那么必须在能够运用算法之前替换嵌入的 **interleave** 语句。

注 3 — 由于1）～5）的限制，为**interleave** 语句的内嵌寻找有限的替换总是可能的。

替换算法对 **interleave** 语句的图形表示起作用，并将其转换为语义上等同的树结构，用它来描述一系列内嵌的 **alt** 语句。对这一点，需要单独语句、**if-else**，阻塞 **call**，**alt** 和 **interleave** 复合语句的图形表示。

用一个带语句的节点来描述单独语句，作为标注。单独语句序列通过一组用流程线相连的节点来描述。如图1所示：

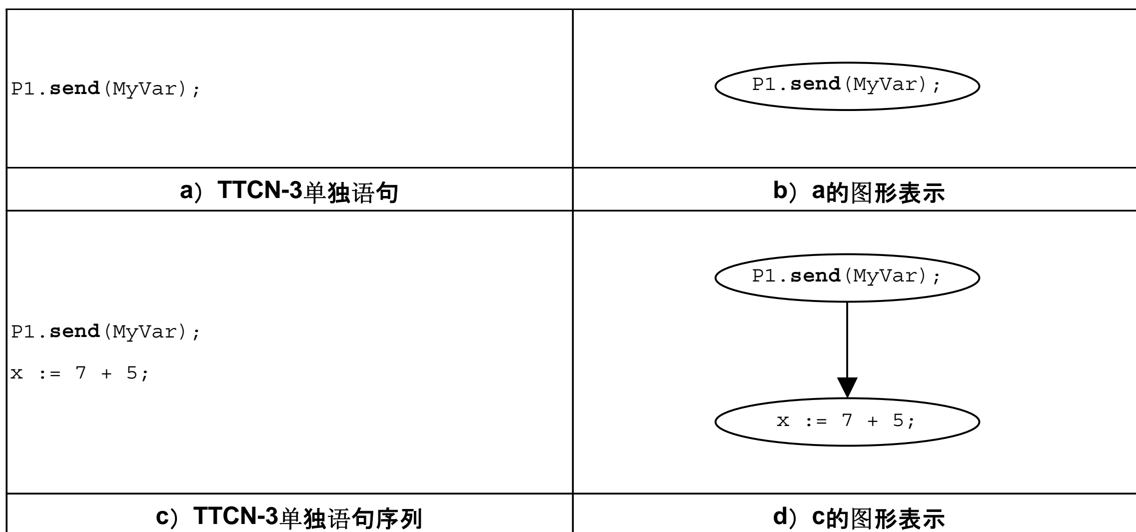


图 1/Z.143－TTCN-3单独语句的图形表示

图 2 所示为 **if-else** 语句的图形表示。**if-else** 语句由 IF 节点表示，它带两条连接于两种选择中第一个语句的流程线。如果在 **if-else** 语句之后还有一些语句，那么不带 ELSE 分支的 **if-else** 语句以相同的方式来表示。这种情况下，表示 *else* 分支的流程线连接至 **if-else** 语句后的第一个语句上。不带 ELSE 分支和不带下列语句的 **if-else** 语句，用只带有一条流程线的 IF 节点来表示。

注 4 — 仅出于可读性目的而引入图1中流程线上的标注。算法只使用流程线而非标注所表达的关系。

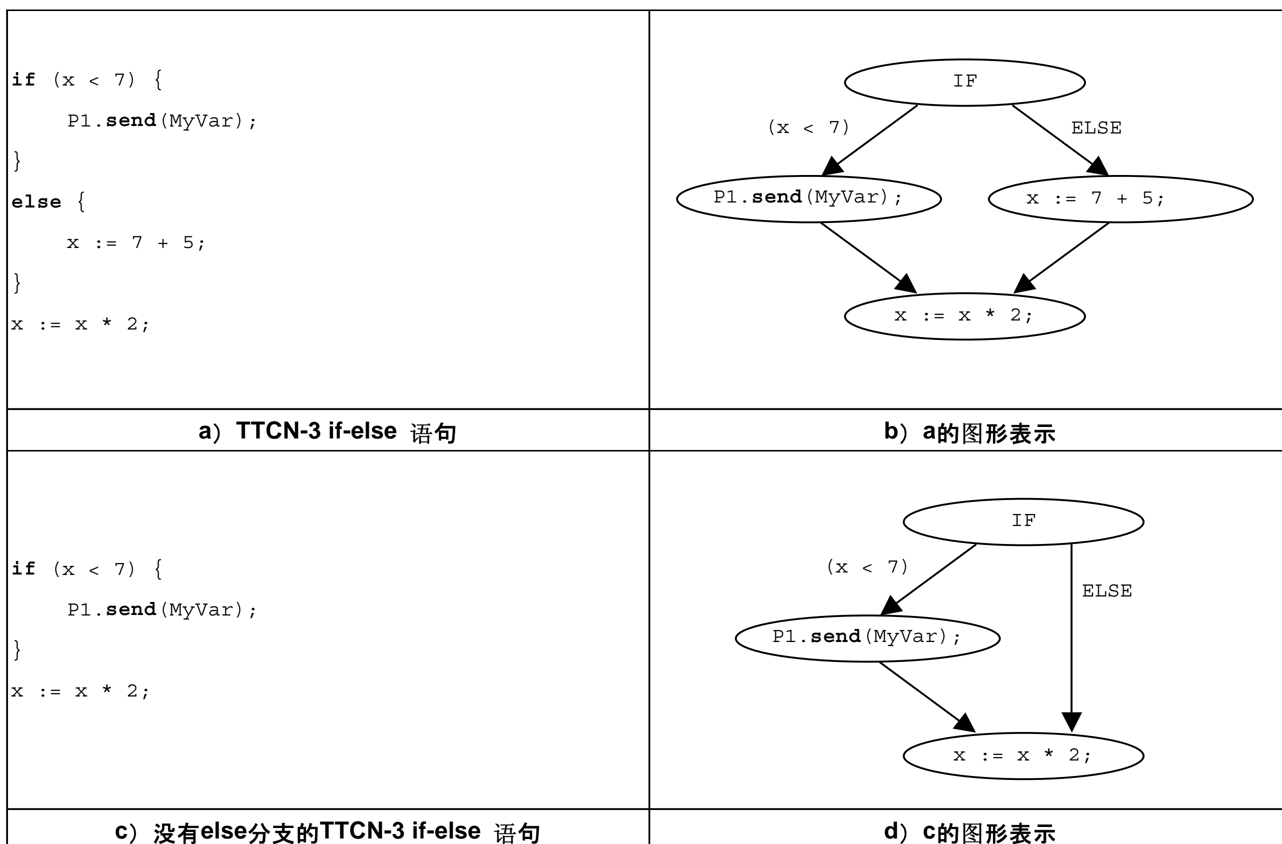
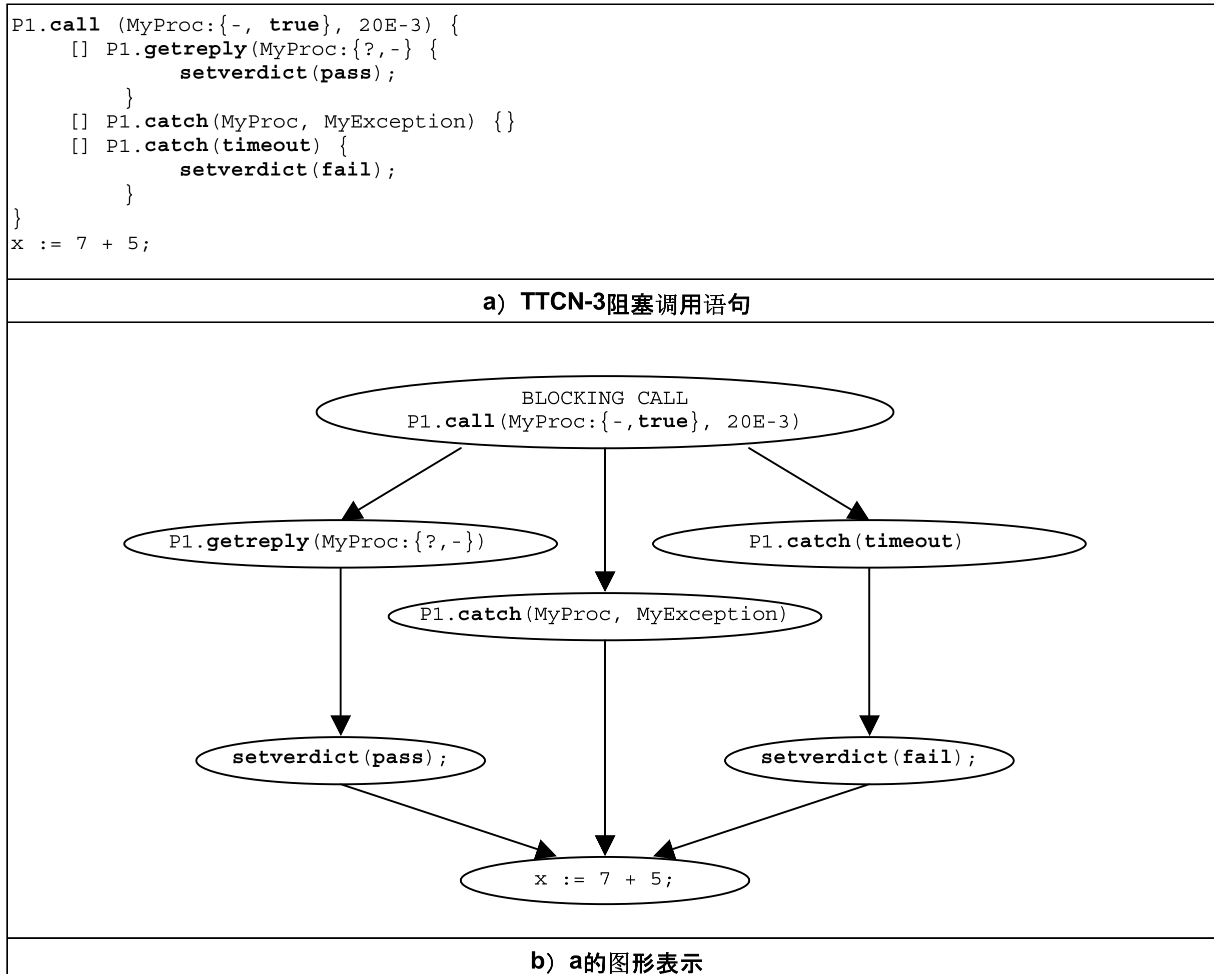


图 2/Z.143－TTCN-3 if-else 语句的图形表示

图 3 所示为阻塞 **call** 语句的图形表示。阻塞 **call** 语句用 BLOCKING-CALL 节点来表示，它带有若干条连接至不同选择的 **getreply** 和 **catch** 语句的流程线。



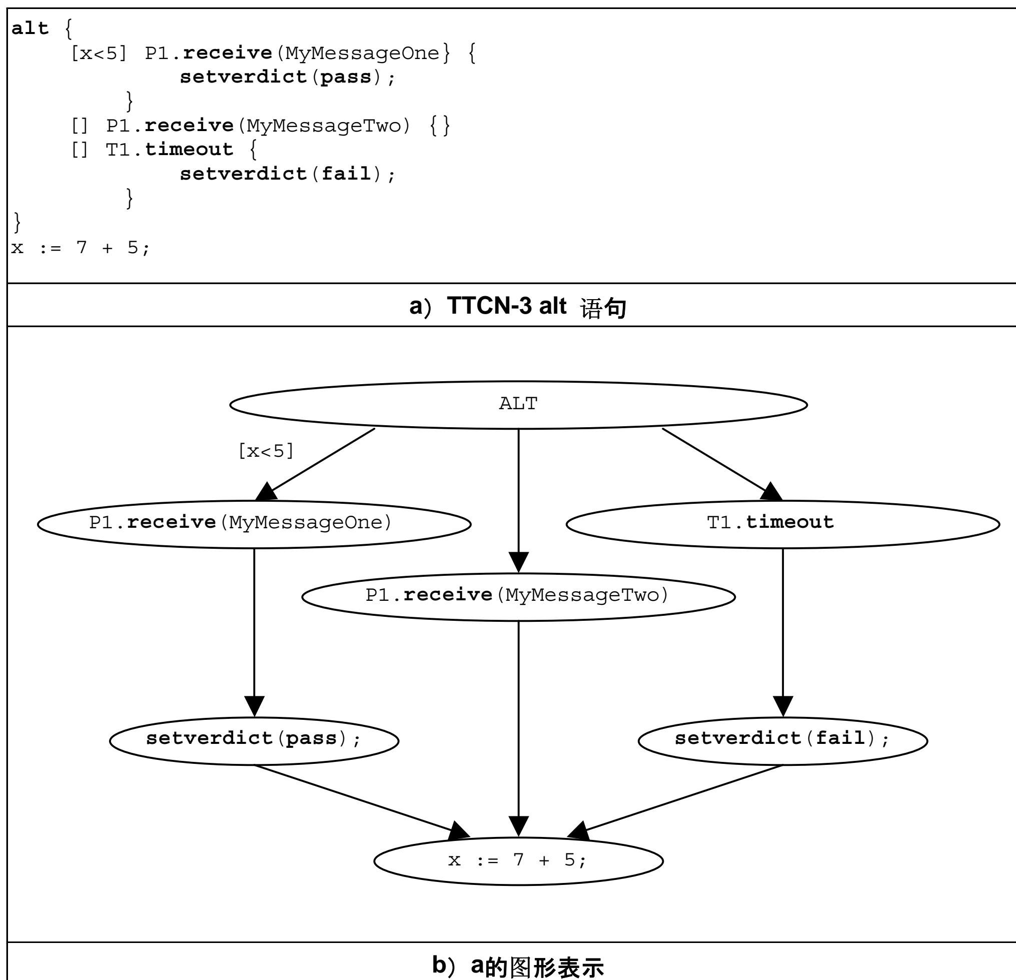
```
P1.call (MyProc:{-, true}, 20E-3) {
    [] P1.getreply(MyProc:{?,-} {
           setverdict(pass);
       }
    [] P1.catch(MyProc, MyException) {}
    [] P1.catch(timeout) {
           setverdict(fail);
       }
}
x := 7 + 5;
```

a）TTCN-3阻塞调用语句

b）a的图形表示

图 3/Z.143－TTCN-3阻塞调用语句的图形表示

图 4 所示为 **alt** 语句的图形表示。**alt** 语句用 **alt** 节点来表示，它带有若干条连接至不同选择的流程线。

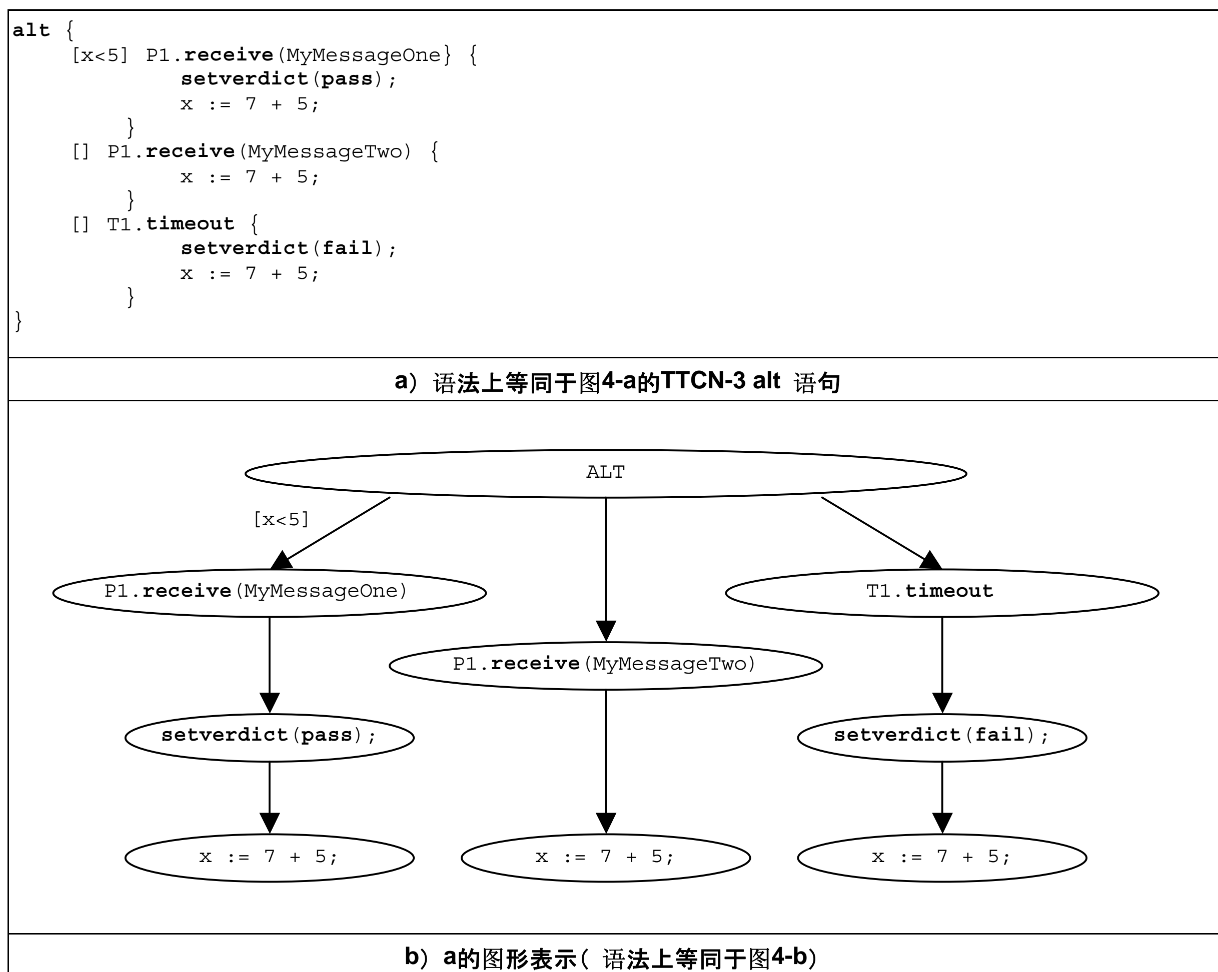


a）TTCN-3 alt 语句

b）a的图形表示

图 4/Z.143－TTCN-3 alt 语句的图形表示

通常，**if-else**、阻塞 **call** 和 **alt** 语句的图形表示都是不带回路的有向图，其中，当离开该语句时，不同选择的流程线实现连接。通过复制，将此类有向图转移至语义上等同的树表示上是可能的。如图 5 中所示，它针对的是图 4 中的 alt 语句。以下描述的算法将构造此类树表示。



a）语法上等同于图4-a的TTCN-3 alt 语句

b）a的图形表示（ 语法上等同于图4-b）

图 5/Z.143－TTCN-3 alt 语句的图形表示

interleave 语句可用图形来描述，它由一组有向子图组成，每个有向子图通过单独的语句和复合语句 **if-else**、阻塞 **call** 和 **alt** 来构造。有向子图描述交错的控制流。图 6 是一个示例。图 6-b 中的节点标注指的是图 6-a 中 TTCN-3 语句的标签。

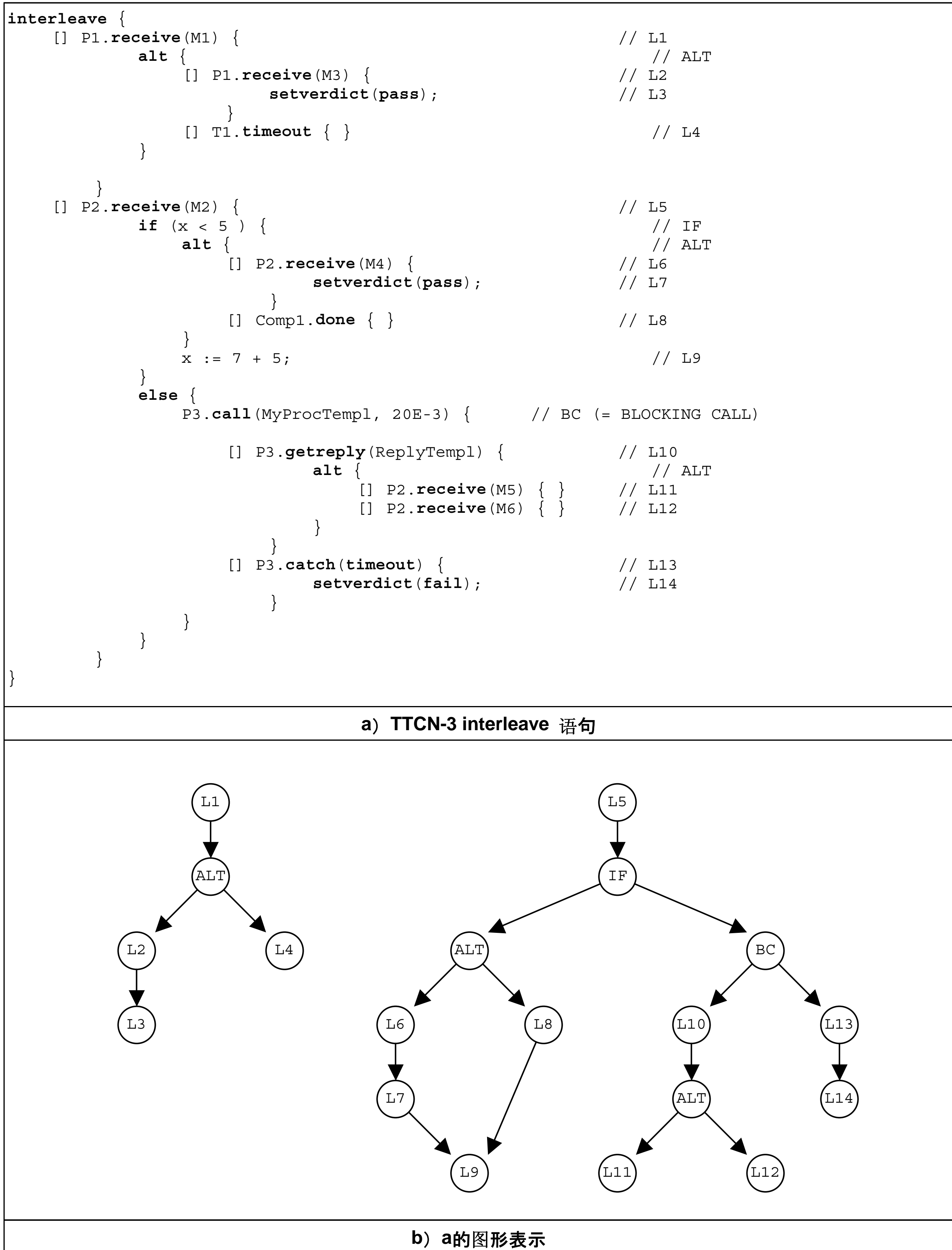


图 6/Z.143－TTCN-3 interleave 语句的图形表示

正式地，`interleave` 语句可用 $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ 图形来描述，其中：

$\boldsymbol{St}$ 是一组允许的TTCN-3 语句；以及

$\boldsymbol{F} \subseteq (\boldsymbol{St} \times \boldsymbol{St})$ 描述流的关系。

*允许的 TTCN-3 语句*这一术语指的是以上静态限制 1～5。

对构造算法，需要定义以下函数：

- <u>*REACHABLE*</u> 函数返回图$\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$中可从语句$\boldsymbol{s}$达到的所有语句：

  <u>*REACHABLE*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{s} \} \cup$
  $\{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge \exists(\boldsymbol{s} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n} = \boldsymbol{stmt})$ 其中： $\boldsymbol{x_i} \in \boldsymbol{St}$,
  $\mathrm{i} \in \{1 \ldots \mathrm{n}\} \wedge (\boldsymbol{x_i}, \boldsymbol{x_{i+1}}) \in \boldsymbol{F}\}$

- <u>*SUCCESSORS*</u> 函数返回图$\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$中语句$\boldsymbol{s}$的所有后续语句：

  <u>*SUCCESSORS*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{s}, \boldsymbol{stmt}) \in \boldsymbol{F}\}$

- <u>*ENABLED*</u> 函数返回图$\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$中没有父辈的所有语句：

  <u>*ENABLED*</u> $(\boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{F} \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \emptyset)\}$

- <u>*KIND*</u> 函数返回图形中代表`interleave`语句的TTCN-3语句类别或类型。

- <u>*DISCARD*</u> 函数删除语句$\boldsymbol{s}$ 或来自图$\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ 中的一组语句，并返回结果图$\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$:

  对单个节点：

  <u>*DISCARD*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \boldsymbol{GI'}$ 其中：$\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$，且$\boldsymbol{St'} = \boldsymbol{St} \backslash \{\boldsymbol{s}\}$ 和

  $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \{\boldsymbol{s}\} \times \boldsymbol{St} \backslash \{\boldsymbol{s}\})$.

  对节点组：

  <u>*DISCARD*</u> $(\boldsymbol{S}, \boldsymbol{GI}) = \boldsymbol{GI'}$ 其中：$\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$，且$\boldsymbol{St'} = \boldsymbol{St} \backslash \boldsymbol{S}$ 和 $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \boldsymbol{S} \times \boldsymbol{St} \backslash \boldsymbol{S})$.

- <u>*RECEIVING*</u> 函数获取图$\boldsymbol{GI}$中的一组语句，并返回所有接收语句：

  <u>*RECEIVING*</u> $(S) = \{ stmt \mid stmt \in St \wedge$ <u>*KIND*</u>$(stmt) \in$ {receive, trigger, getcall, getreply, catch, check, done, timeout}}

- <u>*RANDOM*</u> 函数随机地从一个给定的集合$\boldsymbol{S}$中选择一个元素并返回$\boldsymbol{s}$。

  <u>*RANDOM*</u> $(\boldsymbol{S}) = \boldsymbol{s}$ 其中：$\boldsymbol{s} \in \boldsymbol{S}$

树的构造算法（参见图 7）是一个递归过程，在该过程的每一次递归调用中，为特定节点构造后续节点。该过程在一个类似于 C 的伪代码记法中进行提供，它使用以上定义的函数和一些额外的数学符号。

```
CONSTRUCT-SUCCESSORS (statementType *predecessor, graphType GI) {
    // - statementType指的是创建的树的节点类型。
    // - *predecessor指的是创建的最后一个节点。
    // - graphType表示TTCN-3语句图形类型。
    // - GI通过值来调用，指的是由需要考虑到的、所有剩余TTCN-3语句组成的子图。

    var graphType myGraph;
    var statementType i, myStmt;
    var statementType *newStmt, *firstInBranch; // 有关递归创建之新语句树的指针。

    // 检索在" GI" 中没有父辈的TTCN-3语句集
    var statementSet enabStmts := ENABLED(GI);          // 所有没有父辈的语句
    var statementSet enabRecStmts := RECEIVING(enabStmts);  // 'enabStmts'中的接收语句
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                                                       // 'enabStmts'中的非接收语句

    if (GI.St == ∅) {   // 假设GI.St指的是GI中的语句集。
        return;      // 不剩余任何语句，递归的终止准则。
    }
    elseif (enabNonRecStmts != ∅) { // 处理'enabStmts'中的非接收语句

        myStmt := RANDOM(enabNonRecStmts);
                // 'enabNonRec'中只能有一个语句，原因是算法继续创建过程，
                // 直至有一个分支生成interleave语句。
        newStmt := create(myStmt, predecessor);
                // 创建一个新的树节点，它表示树中的'myStmt'，
                // 更新'newStmt'和'predecessor'中的指针。

        if (KIND(myStmt) == IF || KIND(myStmt) == BLOCKING_CALL) {
            for each i in SUCCESSORS(myStmt, GI) {

                firstInBranch := create(i, newStmt);
                        // 因if-else语句而为分支中的第一个语句创建第二个节点。
                        // 注意，该创建语句将用于创建表示阻塞调用操作中接收语句的树节点。

                myGraph := DISCARD({i, myStmt} ∪ REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                        // 移去i、myStmt以及所有可从myStmt达到但从i达不到的语句。
                        // 后者考虑GI子图中控制流的分岔情况。

                CONSTRUCT-SUCCESSORS(firstInBranch, myGraph);       // 下一递归步骤。
            }
        }
        elseif (KIND(myStmt) == ALT) {
            for each (i in SUCCESSORS(myStmt, GI) {

            CONSTRUCT-SUCCESSORS(mystmt, DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI)));
                    // NEXT RECURSION STEP, the DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                        // 变元因不同接收事件而考虑控制流的分岔情况。
            }
        }
        else {       // mystmt是一个单独的语句。
            CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, GI));
                    // 下一递归步骤。
        }
    }
    else {  // 处理交错的接收事件。
        if (KIND(predecessor) != ALT) { // 缺少一个alt节点，必须创建，
                                        // 假如交错不受嵌入的alt语句影响。
            predecessor := create(ALT, predecessor);
        }

        for each i in enabRecStmts) {
            newStmt := create(i, predecessor);              // 新的树节点。
            CONSTRUCT-SUCCESSORS(newStmt, DISCARD(i, GI));  // 下面的递归步骤。
        }
    }
}
```

图 7/Z.143－TTCN-3 interleave 语句的替换算法

最初，CONSTRUCT-SUCCESSORS 函数（参见图 7）将用一棵空树的根节点以及应替换的、用于描述 **`interleave`** 语句的 TTCN-3 语句图来调用。终止后，根节点可用来访问构建的树。

将 CONSTRUCT-SUCCESSORS 函数应用于图 6 中所示的 **`interleave`** 语句，将导致图 8 中所示的树。标签指的是图 6-a 中的各语句。多个标签是代码复制的结果。与图 8 中的树相对应的 TTCN-3 代码在图 9 中显示。

注 5 — 图7中的算法应用例子（参见图6、图8和图9）非常全面。提供这一例子的目的是为了表明大多数的特殊情况，即流程线的分岔和连接、嵌入的**`alt`**语句、阻塞的**`call`**语句和**`if-else`**语句。

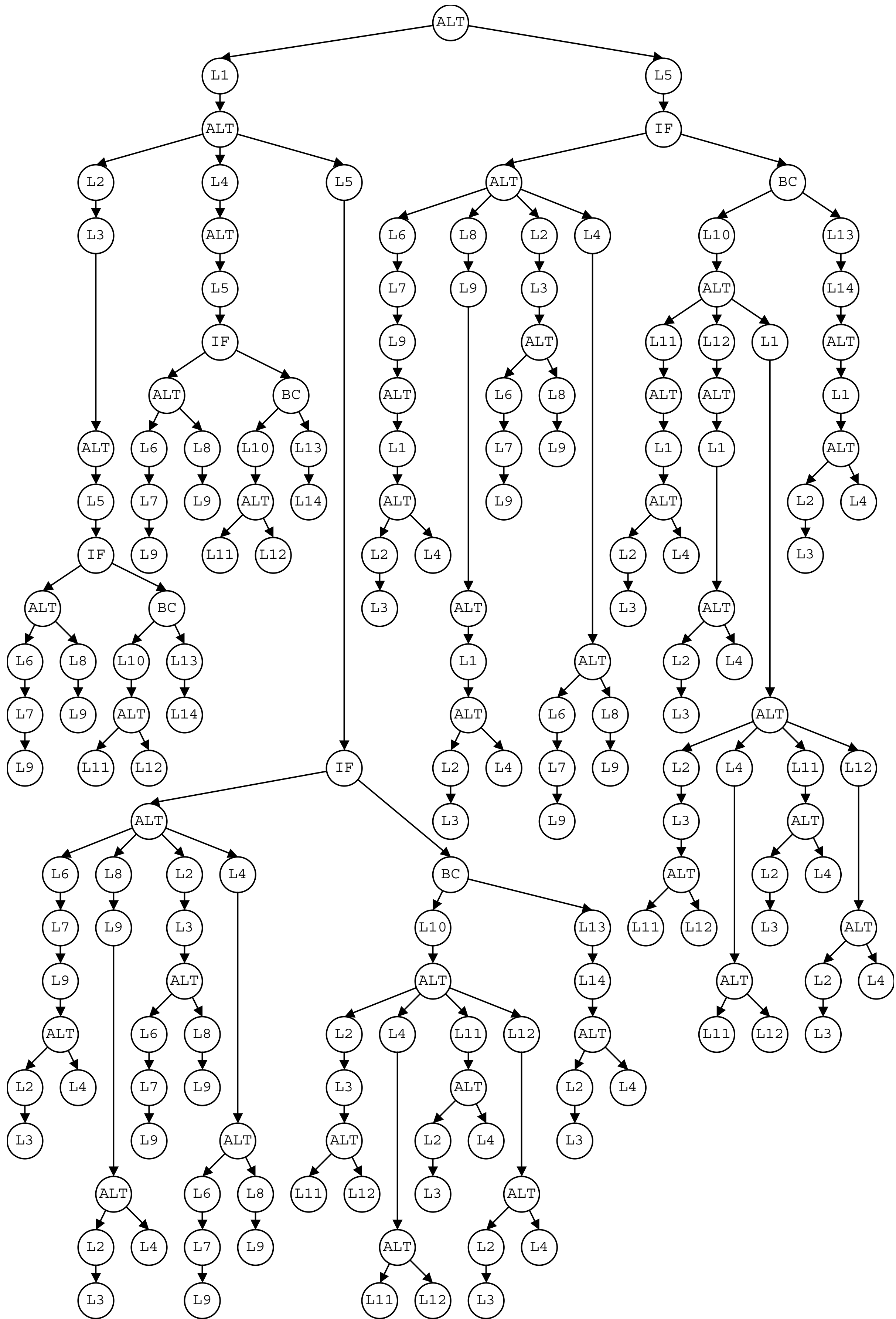


图 8/Z.143－将图7中的算法用于图6中的interleave语句的结果

```
alt {                                                                   // ALT
    []  P1.receive(M1} {                                                // L1
            alt {                                                       // ALT
                []  P1.receive(M3) {                                    // L2
                        setverdict(pass);                               // L3
                        alt {                                           // ALT
                            []  P2.receive(M2) {                        // L5
                                    if (x < 5 ) {                       // IF
                                        alt {                           // ALT
                                                []  P2.receive(M4) {    // L6
                                                        setverdict(pass); // L7
                                                        x := 7 + 5;     // L9
                                                    }
                                                []  Comp1.done {        // L8
                                                        x := 7 + 5;     // L9
                                    }   }               }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {       // L10
                                                    alt {                       // ALT
                                                        []  P2.receive(M5) { }  // L11
                                                        []  P2.receive(M6) { }  // L12
                                                }   }
                                            []  P3.catch(timeout) {             // L13
                                                    setverdict(fail);           // L14
                    }   }       }   }   }       }
                []  T1.timeout {                                        // L4
                        alt {                                           // ALT
                            []  P2.receive(M2) {                        // L5
                                    if (x < 5 ) {                       // IF
                                        alt {                           // ALT
                                                []  P2.receive(M4) {    // L6
                                                        setverdict(pass); // L7
                                                        x := 7 + 5;     // L9
                                                    }
                                                []  Comp1.done {        // L8
                                                        x := 7 + 5;     // L9
                                    }   }               }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {       // L10
                                                    alt {                       // ALT
                                                        []  P2.receive(M5) { }  // L11
                                                        []  P2.receive(M6) { }  // L12
                                                }   }
                                            []  P3.catch(timeout) {             // L13
                                                    setverdict(fail);           // L14
                    }   }       }   }   }       }
                []  P2.receive(M2) {                                    // L5
                        if (x < 5 ) {                                   // IF
                            alt {                                       // ALT
                                []  P2.receive(M4) {                    // L6
                                        setverdict(pass);               // L7
                                        x := 7 + 5;                     // L9
                                        alt {                           // ALT
                                            []  P1.receive(M3) {        // L2
                                                    setverdict(pass);   // L3
                                                }
                                            []  T1.timeout { }          // L4
                                    }   }
                                []  Comp1.done {                        // L8
                                        x := 7 + 5;                     // L9
                                        alt {                           // ALT
                                            []  P1.receive(M3) {        // L2
                                                    setverdict(pass);   // L3
                                                }
                                            []  T1.timeout { }          // L4
                                    }   }
                                []  P1.receive(M3) {                    // L2
                                        setverdict(pass);               // L3
                                        alt {                           // ALT
                                            []  P2.receive(M4) {        // L6
                                                    setverdict(pass);   // L7
                                                    x := 7 + 5;         // L9
                                                }
                                            []  Comp1.done {            // L8
                                                    x := 7 + 5;         // L9
                                    }   }       }
                                []  T1.timeout {                        // L4
                                        alt {                           // ALT
```

```
                                        []  P2.receive(M4) {                        // L6
                                                setverdict(pass);                   // L7
                                                x := 7 + 5;                         // L9
                                            }
                                        []  Comp1.done {                            // L8
                                                x := 7 + 5;                         // L9
                    }   }       }   }       }
                    else {
                        P3.call(MyProcTempl, 20E-3) {           // BC (= BLOCKING CALL)
                            []  P3.getreply(ReplyTempl) {                           // L10
                                    alt {                                           // ALT
                                        []  P1.receive(M3) {                        // L2
                                                setverdict(pass);                   // L3
                                                alt {                               // ALT
                                                    []  P2.receive(M5) { }          // L11
                                                    []  P2.receive(M6) { }          // L12
                                            }   }
                                        []  T1.timeout {                            // L4
                                                alt {                               // ALT
                                                    []  P2.receive(M5) { }          // L11
                                                    []  P2.receive(M6) { }          // L12
                                            }   }
                                        []  P2.receive(M5) {                        // L11
                                                alt {                               // ALT
                                                    []  P1.receive(M3) {            // L2
                                                            setverdict(pass);       // L3
                                                        }
                                                    []  T1.timeout { }              // L4
                                            }   }
                                        []  P2.receive(M6) {                        // L12
                                                alt {                               // ALT
                                                    []  P1.receive(M3) {            // L2
                                                            setverdict(pass);       // L3
                                                        }
                                                    []  T1.timeout { }              // L4
                                }   }       }   }
                            []  P3.catch(timeout) {                                 // L13
                                    setverdict(fail);                               // L14
                                    alt {                                           // ALT
                                        []  P1.receive(M3) {                        // L2
                                                setverdict(pass);                   // L3
                                            }
                                        []  T1.timeout { }                          // L4
    }   }       }   }   }       }   }
[]  P2.receive(M2) {                                                                // L5
        if (x < 5 ) {                                                               // IF
            alt {                                                                   // ALT
                []  P2.receive(M4) {                                                // L6
                        setverdict(pass);                                           // L7
                        x := 7 + 5;                                                 // L9
                        alt {                                                       // ALT
                            []  P1.receive(M1} {                                    // L1
                                    alt {                                           // ALT
                                        []  P1.receive(M3) {                        // L2
                                                setverdict(pass);                   // L3
                                            }
                                        []  T1.timeout { }                          // L4
                    }   }       }   }
                []  Comp1.done {                                                    // L8
                        x := 7 + 5;                                                 // L9
                        alt {                                                       // ALT
                            []  P1.receive(M1} {                                    // L1
                                    alt {                                           // ALT
                                        []  P1.receive(M3) {                        // L2
                                                setverdict(pass);                   // L3
                                            }
                                        []  T1.timeout { }                          // L4
                    }   }       }   }
                []  P1.receive(M3) {                                                // L2
                        setverdict(pass);                                           // L3
                        alt {                                                       // ALT
                            []  P2.receive(M4) {                                    // L6
                                    setverdict(pass);                               // L7
                                    x := 7 + 5;                                     // L9
                                }
                            []  Comp1.done {                                        // L8
                                    x := 7 + 5;                                     // L9
                    }   }       }
                []  T1.timeout {                                                    // L4
                        alt {                                                       // ALT
```

```
                        []  P2.receive(M4) {                                     // L6
                                setverdict(pass);                                // L7
                                x := 7 + 5;                                      // L9
                            }
                        []  Comp1.done {                                         // L8
                                x := 7 + 5;                                      // L9
            }   }       }   }       }
            else {
                P3.call(MyProcTempl, 20E-3) {                   // BC (= BLOCKING CALL)
                    []  P3.getreply(ReplyTempl) {                                // L10
                            alt {                                                // ALT
                                []  P2.receive(M5) {                             // L11
                                        alt {                                    // ALT
                                            []  P1.receive(M1} {                 // L1
                                                    alt {                        // ALT
                                                        []  P1.receive(M3) {     // L2
                                                                setverdict(pass); // L3
                                                            }
                                                        []  T1.timeout { }       // L4
                                    }   }       }   }
                                []  P2.receive(M6) {                             // L12
                                        alt {                                    // ALT
                                            []  P1.receive(M1} {                 // L1
                                                    alt {                        // ALT
                                                        []  P1.receive(M3) {     // L2
                                                                setverdict(pass); // L3
                                                            }
                                                        []  T1.timeout { }       // L4
                                    }   }       }   }
                                []  P1.receive(M1} {                             // L1
                                        alt {                                    // ALT
                                            []  P1.receive(M3) {                 // L2
                                                    setverdict(pass);            // L3
                                                    alt {                        // ALT
                                                        []  P2.receive(M5) { }   // L11
                                                        []  P2.receive(M6) { }   // L12
                                                }   }
                                            []  T1.timeout {                     // L4
                                                    alt {                        // ALT
                                                        []  P2.receive(M5) { }   // L11
                                                        []  P2.receive(M6) { }   // L12
                                                }   }
                                            []  P2.receive(M5) {                 // L11
                                                    alt {                        // ALT
                                                        []  P1.receive(M3) {     // L2
                                                                setverdict(pass); // L3
                                                            }
                                                        []  T1.timeout { }       // L4
                                                }   }
                                            []  P2.receive(M6) {                 // L12
                                                    alt {                        // ALT
                                                        []  P1.receive(M3) {     // L2
                                                                setverdict(pass); // L3
                                                            }
                                                        []  T1.timeout { }       // L4
                        }   }       }   }       }   }
                    []  P3.catch(timeout) {                                      // L13
                            setverdict(fail);                                    // L14
                            alt {                                                // ALT
                                []  P1.receive(M1} {                             // L1
                                        alt {                                    // ALT
                                            []  P1.receive(M3) {                 // L2
                                                    setverdict(pass);            // L3
                                                }
                                             [] T1.timeout { }                   // L4
}       }   }   }       }   }       }   }
```

图 9/Z.143－语法上与图6中interleave 语句等同的TTCN-3代码

7.6 trigger操作的替换

trigger 操作从指定端口上的一个消息流中过滤掉带有特定匹配准则的消息。**trigger** 操作的语义可以通过用两个 **receive** 操作和一个 **goto** 语句对它的替换来描述。操作语义假设该替换在语法层面上进行。

例 1：

```
// 以下trigger操作...

      alt {
       []  MyCL.trigger (MyType:?) { }
      }

// 将由……替换

   alt {
      []   MyCL.receive (MyType:?) { }
      []   MyCL.receive {
           repeat
       }
   }
```

如果 **trigger** 语句用在更复杂的 **alt** 语句中，那么将以同样的方式进行替换。

例 2：

```
// 以下alt语句包括一个trigger语句...

   alt {
      []   PCO2.receive {
           stop;
       }
      []   MyCL.trigger (MyType:?) { }
      []   PCO3.catch {
           setverdict(fail);
           stop;
       }
   }

// 它将由……替换

   alt {
      []   PCO2.receive {
           stop;
       }
      []   MyCL.receive (MyType:?) { }
      []   MyCL.receive {
           repeat;
       }
      []   PCO3.catch {
           setverdict(fail);
           stop;
       }
   }
```

8 TTCN-3的流程图语义

TTCN-3 的操作语义基于对流程图的解释。本节介绍了流程图（见第 8.1 节）、解释了表示 TTCN-3 模块控制、测试用例、可选步骤、函数和部件类型定义的流程图结构（见第 8.2 节）、定义了用于描述 TTCN-3 模块执行状态的模块和部件状态（见第 8.3 节），描述了对消息、远端程序调用、远端程序调用应答和异常的处理（见第 8.4 节），并解释了模块控制和测试用例的计算过程（见第 8.6 节）。

8.1 流程图

流程图是一个有向图，它由带有标签的节点和带有标签的边界组成。遍历流程图描述在执行所表示的行为描述期间可能的控制流。

8.1.1 流程图框架

流程图将置于一个定义流程图边界的框架中。流程图的名称跟在关键字 **flow graph**（这些不是 TTCN-3 核心语言的关键字）后，并将置于流程图的左上角。按照惯例，假定流程图的名称指的是由流程图所表示的 TTCN 行为描述。一个简单的流程图如图 10 所示。

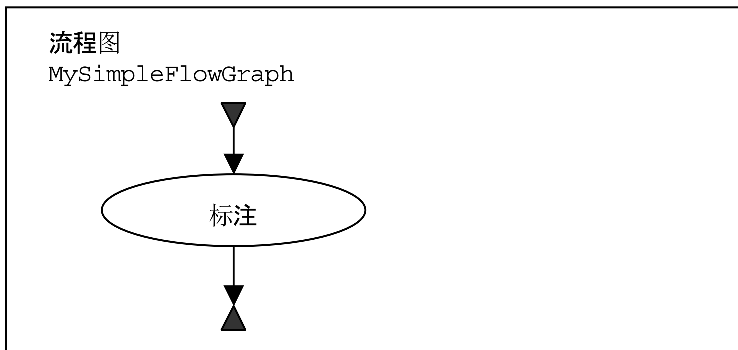


图 10/Z.143－一个简单的流程图

8.1.2 流程图节点

流程图由起始节点、结束节点、基本节点和引用节点组成。

8.1.2.1 起始节点

起始节点描述流程图的起始点。流程图将只有一个起始节点。起始节点如图 11-a 所示。

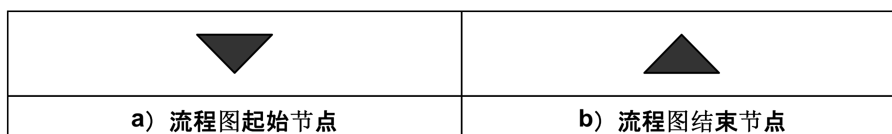


图 11/Z.143－起始节点和结束节点

8.1.2.2 结束节点

结束节点描述流程图的终点。一个流程图可以拥有几个结束节点，或者在闭环的情况下，没有结束节点。没有后继节点的基本节点（见第 8.1.2.3 节）和引用节点（见第 8.1.2.4 节），将与一个结束节点相连，以表示它们描述一个流程图路径的最后动作。结束节点如图 11-b 所示。

8.1.2.3 基本节点

基本节点描述一个执行单元，即它在一步内执行。基本节点具有一种类型，取决于其类型，它可以具有一个相关的属性列表。图 12 显示了两个基本节点。

在基本节点的标注中，节点的属性跟在节点类型之后，并置于圆括号中。类型和属性用于确定在执行所表示的语言结构期间要执行的动作。属性描述将从相应的 TTCN-3 结构中获取的信息。

属性拥有一些值，操作语义将通过引用属性名称来获取这些值。如果需要，允许通过使用赋值符号“:=”来为基本节点赋予一些明确的值。图 12-b 显示了一个例子。

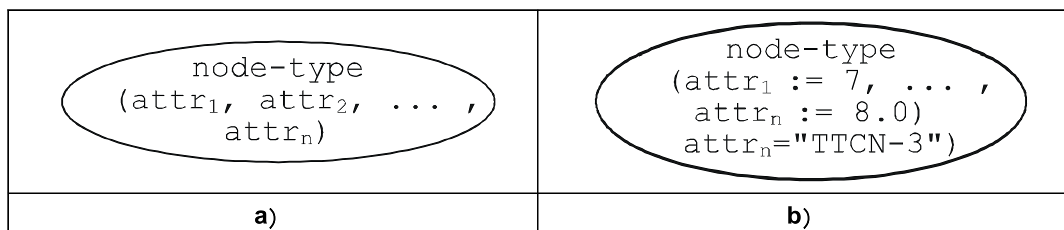


图 12/Z.143－带属性的基本节点

8.1.2.4 引用节点

引用节点指的是作为子流程图的流程图片段（见第 8.1.4 节）。引用节点的含义通过用流程图中所引用的流程图片段替换它来定义。引用节点的节点标注为流程图片段提供了引用。引用节点如图 13-a 所示。

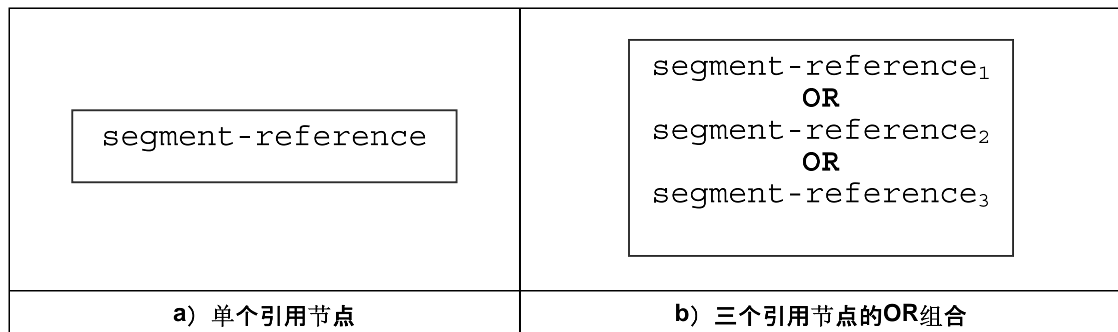


图 **13/Z.143**－引用节点

8.1.2.4.1 引用节点的OR合并

某些情况下，一些流程图片段可以替换引用节点。对这些情况，**OR** 运算符可用来表示一些流程图片段（见图 13-b）。在表示模块控制、测试用例或函数的实际流程图中，由所表示的结构来确定一个选择项。

8.1.2.4.2 引用节点的多次出现

某些情况下，相同种类的引用节点可以在流程图中出现零次、一次或多次。在常规表述中，通过使用运算符符号'+'（一次或多次重复）和'*'（零次或多次重复），来描述常规表述中各部分可能的重复。如图 14 所示，通过引入带相关运算符符号的、加双外框的引用节点，流程图采用了这些运算符。一旦出现零次（使用带'*'运算符、加双外框的引用节点），单流程线（见第 8.1.3 节）将替换引用节点。

图 **14/Z.143**－引用节点的重复

引用节点可能重复的最多次数可以以圆括号中的一个整数的形式来提供，该整数跟在加双外框的引用节点中'*'或'+'符号之后。图 15 所示的片段引用可以出现 0～5 次。

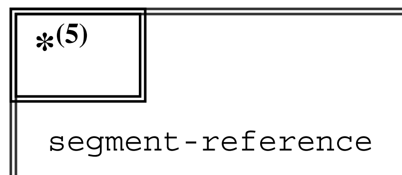
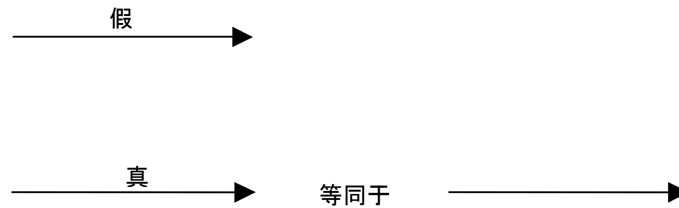


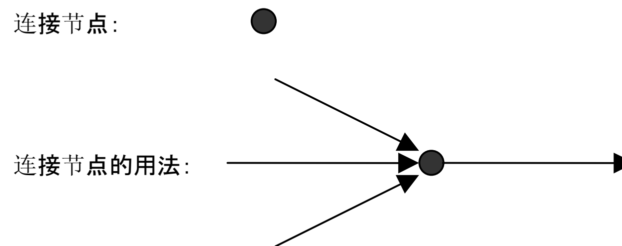
图 **15/Z.143**－引用节点受限的重复

8.1.3 流程线

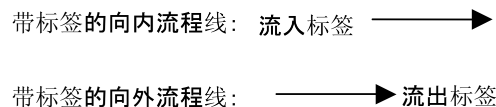
流程线以箭头方式来表示。流程线有一个真或假的标注，指明在解释流程图期间流程线的选择条件。作为一种简短记法，它允许省略真标注。流程线的例子如下所示：



为支持将若干流程线连接成图形层面上的一条流程线，引入了一个特殊的连接节点。下图表示了连接节点及其用法的例子。



例如，对 TTCN-3 结构 `goto` 和 `label` 建模而言，在大图中画长流程线是必要的，但是不易使用。出于这一目的，可以使用向外和向内的流程线标签。例子如下所示：



如果标签是相同的，那么带标签的向外流程线与带标签的向内流程线相连。向内流程线的流程线标签将是唯一的。如果存在若干条带相同标签的向外流程线，那么认为这是各条线连接至向内流程线，其标签相同。

8.1.4 流程图片段

流程图片段是子流程图。它们在引用节点中被引用，并定义该引用节点的含义。流程图片段可包括进一步的引用节点。

如图 16 所示，流程图片段拥有精确的接口，这些接口由向内和向外流程线组成。只有一条不带标签的向内流程线，以及一条或多条不带标签的向外流程线。此外，可能存在一些带标签的向内流程线和向外流程线。例如，需要用带标签的向内流程线和向外流程线来描述 TTCN-3 语句 `goto` 和 `alt` 的含义。

流程图片段置于一个框架内，流程图片段的名称将跟在关键字 `segment` 后，随后是在框架左上角的片段名称。描述流程图片段接口的流程线将横跨流程图片段框架。

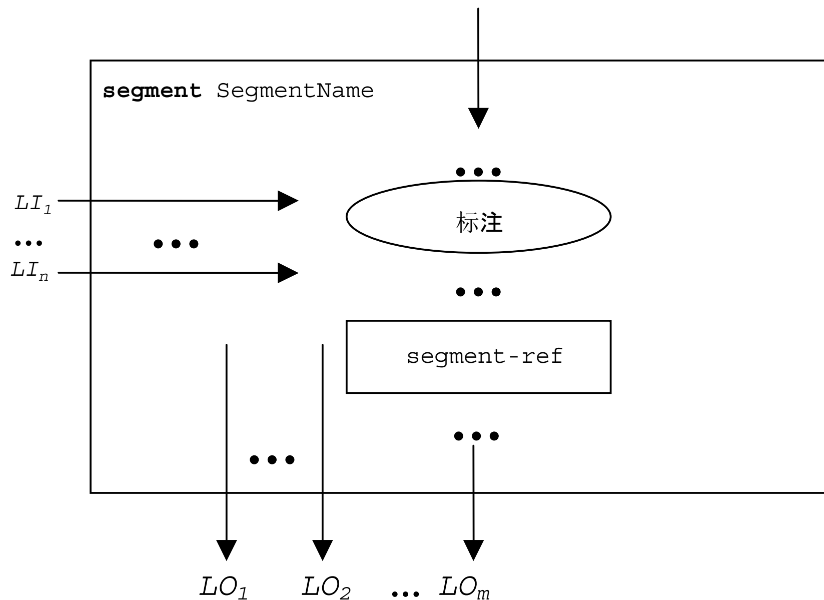


图 16/Z.143－流程图片段描述的结构

8.1.5 注释

为改进可读性和一致性，可使用一个特殊的注释符号来将流程图节点和流程线的注释关联起来。注释符号及其用法如图 17 所示。

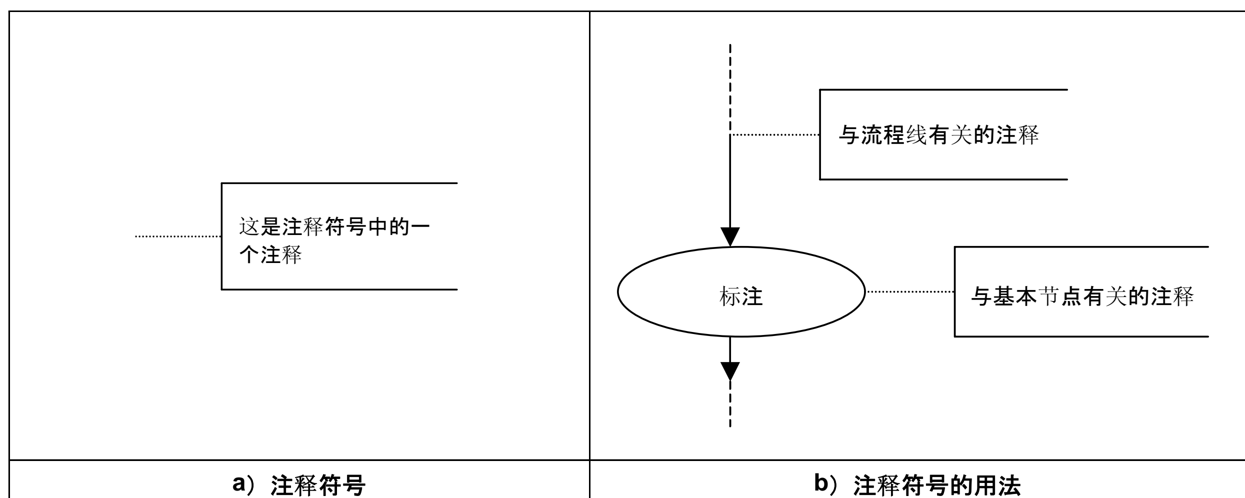


图 17/Z.143－注释的流程图表示

8.1.6 流程图描述的处理

操作语义的计算过程遍历只包括基本节点的流程图，也就是说，所有引用节点都必须通过相应的流程图片段定义加以扩展。要求用 NEXT 函数来支持这种遍历。NEXT 函数以如下方式进行定义：

actualNodeRef.NEXT(*bool*) := `successorNodeRef` 其中：

- *actualNodeRef* 是对基本流程图节点的引用；
- *successorNodeRef* 是对*actualNodeRef*所引用节点之后继节点的引用；
- *bool* 是一个布尔值，用于指定是否返回真或假后继节点（见第8.1.3节）。

8.2 TTCN-3行为的流程图表示

操作语义假设 TTCN-3 行为描述以一组流程图的形式提供，即对各 TTCN-3 行为描述，必须构建一个单独的流程图。

操作语义将下列种类的 TTCN-3 定义解释为行为描述：

a） 模块控制；

b） 测试用例定义；

c） 函数定义；

d） 可选步骤定义；

e） 部件类型定义。

模块控制规定测试活动，即实际测试用例的执行次序（可能是重复的）。测试用例定义 MTC 的行为。函数构成行为。它们通过模块控制或通过测试部件来执行。可选步骤用做缺省行为的定义或以一种类似于函数的方式来构成行为。假定部件类型定义就是行为描述，因为在创建一个部件类型示例期间，它们规定端口、常量、变量和定时器的创建、声明和初始化。

8.2.1 流程图构建程序

图 18 至图 22 介绍的流程图和第 8 节中介绍的流程图片段，只是一些模板。它们包括必须提供的信息占位符号，以便产生一个具体的流程图或流程图片段。占位符号用“<”和“>”括号来标记。

TTCN-3 模块的流程图表示分三个步骤来构造：

1） 为模块控制、测试用例、可选步骤、函数和部件类型定义中的各TTCN-3语句构造一个具体的流程图片段。

2） 为模块控制和各测试用例、可选步骤、函数和部件类型定义，构建一个具体的流程图（带引用节点）。

3） 具体流程图中的所有引用节点逐步地用相应流程图片段定义予以替换，直至所有的流程图都只包括一个起始节点、结束节点和基本流程图节点。

注 1 — 基本流程图节点描述基本的不可分割的执行单元。TTCN-3行为的操作语义基于对基本流程图节点的解释。第8.6节只介绍基本流程图节点的执行方法。

用相应的流程图片段定义替换引用节点可能导致流程图中不相连的部分，即不能通过沿流程线遍历流程图的方法从起始节点到达的部分。操作语义将忽略流程图的不相连部分。

注 2 — 流程图的不相连部分是机械替换过程的结果。为构造一个最佳的流程图表示，TTCN-3语句的不同结合也必须予以重视。不过，本建议书的目的是提供一个正确和完整的语义，而非最佳流程图表示。

8.2.2 模块控制的流程图表示

示意性地，TTCN-3 模块的语法结构为：

```
module <identifier> <module-definitions-part> control <statement-block>
```

对流程图行为表示，只有下列信息是相关的：

```
module <identifier> <statement-block>
```

这对于函数定义是可比的，因此模块控制的流程图表示类似于函数的流程图表示（见第 8.2.4 节）。语义将通过使用模块名称的方法来访问表示模块控制的流程图。

注 — 模块定义部分的含义超出了本操作语义的范围。模块参数定义为在运行时间时的全局常量。对模块参数的引用必须在语法层面上用其具体的值来替换（见第8.3节）。

图 18 显示了模块控制流程图表示的设计图。流程图名称 control 标识表示模块控制的流程图。流程图的节点具有相关的注释，描述不同节点的含义。引用节点<stop-entity-op> 包括了那些未规定明确 **stop** 操作的情况，即操作语义假定 **stop** 操作是隐性地添加的。

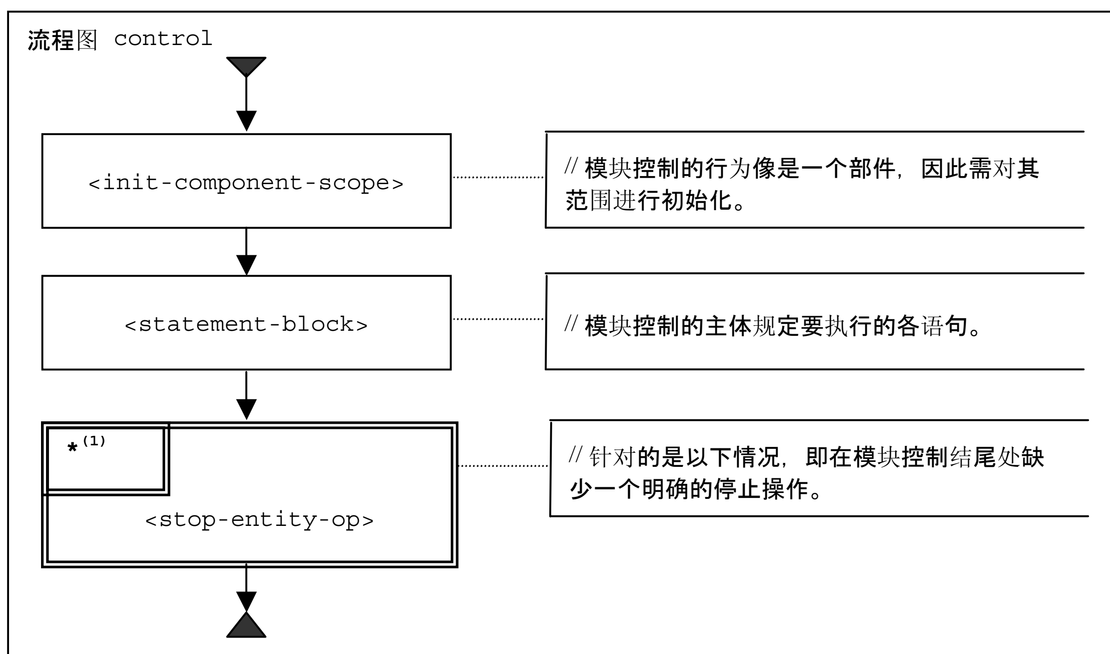


图 18/Z.143－模块控制的流程图表示

8.2.3 测试用例的流程图表示

示意性地，TTCN-3 测试用例定义的语法结构为：

```
testcase <identifier> (<parameter>) <testcase-interface> <statement-block>
```

以上的<testcase-interface>指的是测试用例定义中的（强制性的）**runs on** 和（可选的）**system** 子句。测试用例的流程图描述用于描述 MTC 的行为。由<testcase-interface>提供的信息与 MTC 不相关。它将由 **execute** 语句使用，但不必在测试用例的流程图表示中予以表示。因此，对流程图表示，只有下列信息是相关的：

```
testcase <identifier> (<parameter>) <statement-block>
```

图 19 显示了测试用例流程图表示的设计图。流程图名称`<identifier>` 指的是所表示测试用例的名称。流程图节点具有描述不同节点含义的相关注释。引用节点`<stop-entity-op>` 包括了那些对 MTC 未规定明确 **stop** 操作的情况，即操作语义假定 **stop** 是隐性地添加的。

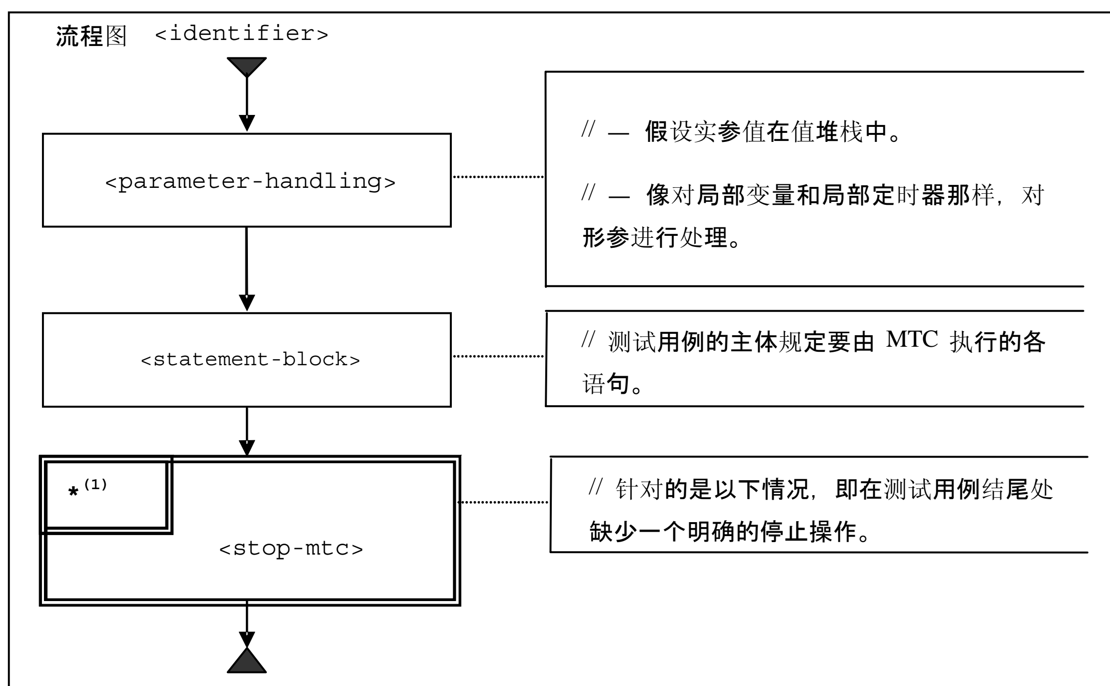


图 19/Z.143－测试用例的流程图表示

8.2.4 函数的流程图表示

示意性地，TTCN-3 函数的语法结构为：

```
function <identifier> (<parameter>) [<function-interface>] <statement-block>
```

以上可选的`<function-interface>`指的是在函数定义中的 **runs on** 和 **return** 子句。`<function-interface>`提供的信息与行为描述是不相关的。它将用于静态语义检查，但不必在流程图中表示。因此，对流程图表示，只有下列信息是相关的：

```
function <identifier> (<parameter>) <statement-block>
```

通过使用函数名称，语义将访问表示函数的流程图。

图 20 显示了函数流程图表示的设计图。流程图名称<identifier> 指的是所表示函数的名称。引用节点<return-without-value>包括了那些未规定明确 **return** 语句的情况，即操作语义假定 **return** 语句是隐性地添加的。

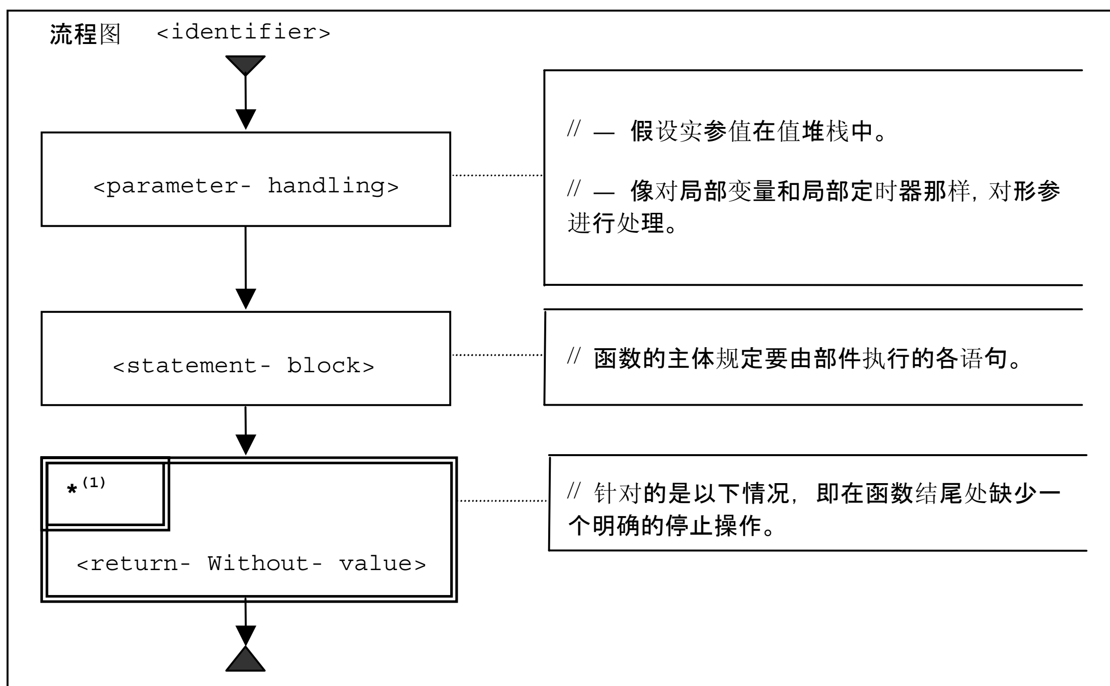


图 20/Z.143－函数的流程图表示

8.2.5 可选步骤的流程图表示

示意性地，TTCN-3 可选步骤的语法结构为：

```
altstep <identifier> (<parameter>) [<altstep-interface>]
            <constant-variable-timer-declarations>
             { <receiving-branch> | <else-branch> }*
```

以上可选的<altstep-interface>指的是在可选步骤定义中的 **runs on** 子句。<altstep-interface>提供的信息与行为描述是不相关的。它将用于静态语义检查，但不必在流程图中表示。因此，对于流程图表示，只有以下信息是相关的：

```
altstep <identifier> (<parameter>) [<altstep-interface>]
            <constant-variable-timer-declarations>
             { <receiving-branch> }*
             [ <else-branch> ]
```

注 — 只考虑第一个else分支的选择和第一个else分支。第一个else分支之后的各分支是无法达到的。

语义将通过使用可选步骤名称来访问表示可选步骤的流程图。

图 21 显示了可选步骤流程图表示的设计图。流程图名称<identifier> 指的是所表示可选步骤的名称。引用节点<successful-altstep-termination>包括了可选步骤在选择和执行一个选项后终止的情况。引用节点<unsuccessful-altstep-termination>规定了未执行任何可选步骤选择的情况。

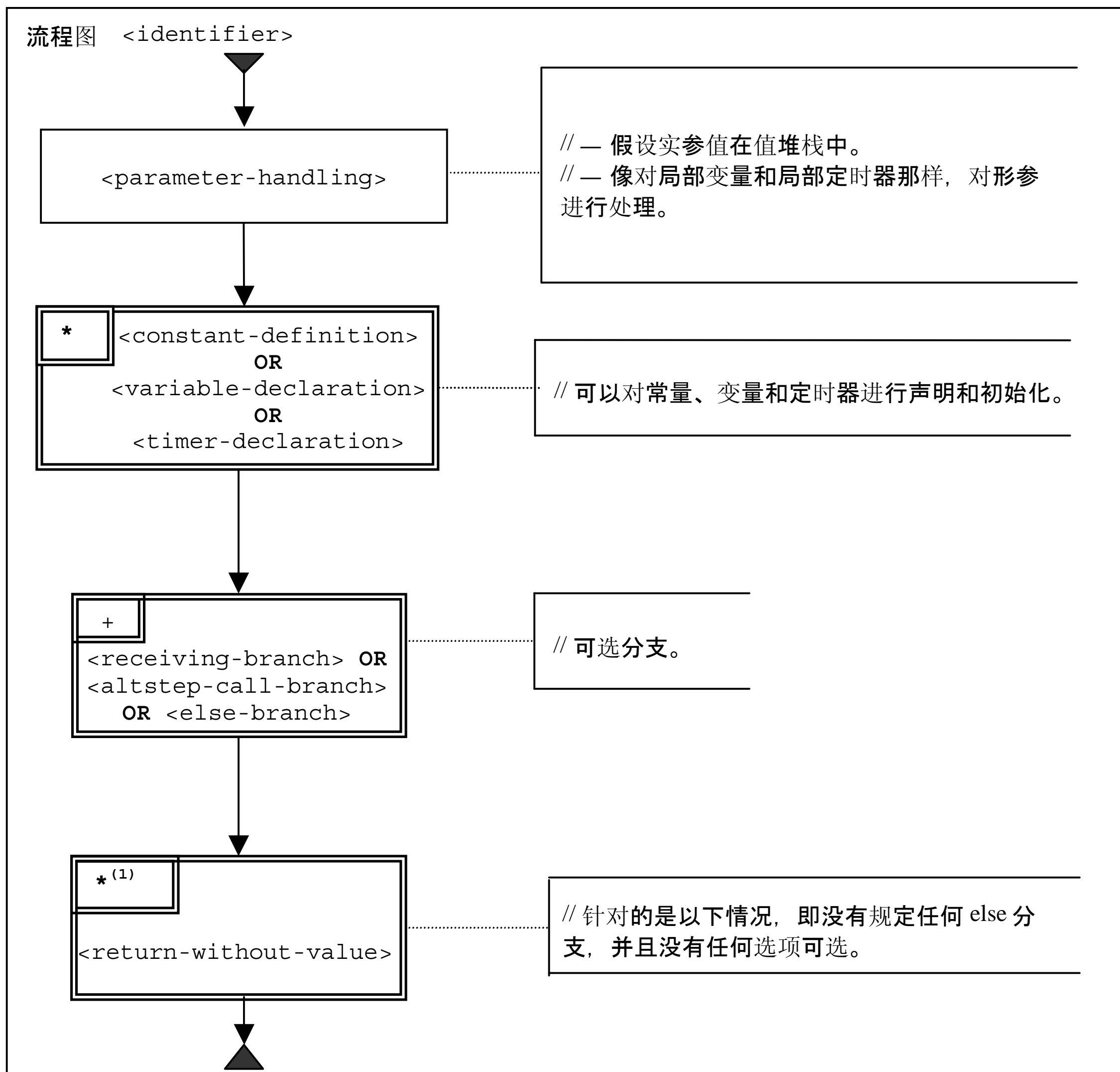


图 21/Z.143－可选步骤的流程图表示

8.2.6 部件类型定义的流程图表示

示意性地，TTCN-3 部件类型定义的语法结构为：

```
type component <identifier> <port-constant-variable-timer-declarations>
```

语义将通过使用部件类型名称来访问表示类型的流程图。

图 22 显示了部件类型定义流程图表示的设计图。流程图名称<identifier>指的是所表示部件类型的名称。

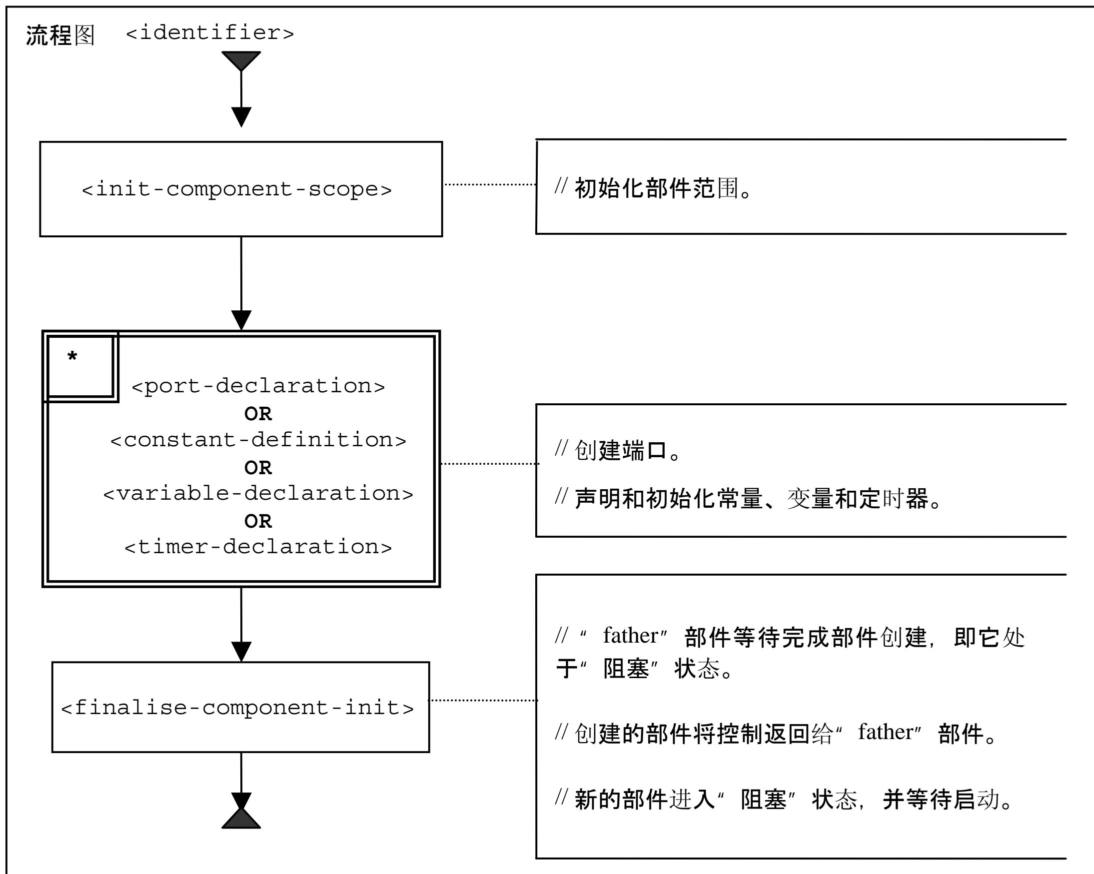


图 22/Z.143－部件类型定义的流程图表示

8.2.7 获取流程图的起始节点

为了获取流程图的起始节点引用，需要以下函数：

```
The GET-FLOW-GRAPH function:          GET-FLOW-GRAPH (flow-graph-identifier)
```

函数向流程图的起始节点返回一个引用，其名称为 *flow-graph-identifier*。*flow-graph-identifier* 指的是控制的模块名称、测试用例名称、函数名称和部件类型定义。

8.3 TTCN-3模块的状态定义

在解释表示 TTCN-3 行为的流程图过程中，要使用模块状态。模块状态是一个结构化的状态，由一些描述测试部件和端口状态的子状态组成。模块状态、部件状态和端口状态都在本节中进行介绍。此外，还定义了一些函数，以便从状态中获取信息或使用状态。

8.3.1 模块状态

如图 23 所示，模块状态可分解为 *ALL-ENTITY-STATES*、*ALL-PORT-STATES*、*MTC*、*TC-VERDICT*、*DONE* 和 *SNAP-ACTIVE*。*ALL-ENTITY-STATES* 描述模块控制状态以及执行测试用例期间实例化测试部件的状态。*ALL-PORT-STATES* 、*MTC* 引用和 *TC-VERDICT* 只有在测试用例执行期间是相关的。*ALL-PORT-STATES* 描述不同端口的状态。*MTC* 为主测试部件（MTC）提供一个引用，*TC-VERDICT* 保存测试用例实际的全局测试判定，*DONE* 是测试用例执行期间所有已停止测试部件的列表，*SNAP-ACTIVE* 用作 MTC 快照的一部分。当 MTC 照快照时，*SNAP-ACTIVE* 保存活动测试部件的数目。它用于 **all component.done** 和 **all component.running** 操作的计算。

注 1 — TC-VERDICT 更新的数量等同于已终止测试部件的数量。

对模块控制（图 23 中的 *M-CONTROL*）的行为当作正常的测试部件来处理，其状态是模块状态 *ALL-ENTITY-STATES* 中的第一个元素。

ALL-ENTITY-STATES						ALL-PORT-STATES					MTC	TC-VERDICT	DONE	SNAP-ACTIVE
	M-CONTROL	ES_1	...	ES_n			P_1	...	P_n					

图 23/Z.143－模块状态的结构

注 2 — 端口状态可认为是实体状态的一部分。通过`connect` 和 `map`，端口对其他部件变得可见，因此，该操作语义处理处于模块状态最顶层的端口。

8.3.1.1 访问模块状态

MTC、*TC-VERDICT* 和 *SNAP-ACTIVE* 都是模块状态的组成部分，当作全局变量来处理，即关键字 *MTC* 和 *TC-VERDICT* 可以用来获取和改变相应模块状态的值。

注 1 — 在解释TTCN-3模块期间，只存在一种模块状态。因此，可认为关键字*MTC* 和 *TC-VERDICT* 是计算程序全局唯一的标识符。

为处理 *ALL-ENTITY-STATES*、*ALL-PORT-STATES* 和 *DONE* 列表，可使用 *add*、*append*、*delete*、*member*、*first*、*length*、*next*、*random* 和 *change* 操作列表。它们具有如下含义：

- *myList.add(item)* 在列表*myList*中加入*item*，作为第一个元素；
- *myList.append(item)* 在列表*myList*中附加*item*，作为最后一个元素；
- *myList.delete(item)* 从列表*myList*中删除*item*；
- 如果*item* 是列表*myList*中的一个元素，那么*myList.member(item)*返回`true`，否则返回`false`；
- *myList.first()* 返回*myList* 的第一个元素；
- *myList.length()*返回*myList* 的长度；
- *myList.next(item)* 返回*myList* 中跟在*item*后的元素，或者如果*item*是*myList* 中的最后元素，那么返回`NULL`；
- *MyList.random(<condition>)* 随机返回*myList* 的一个元素，如果*myList*的元素不满足*<condition>*，那么它满足布尔条件*<condition>*或`NULL`；
- *MyList.change(<operation>)* 允许对*myList*的所有元素应用*<operation>*。

注 2 — 操作的随机和变化不是普通的列表操作。引入它们是为了解释TTCN-3 操作中关键字`all` 和`any`的含义。

8.3.2 实体状态

实体状态用于描述模块控制和测试部件的实际状态。在模块状态中，实体状态在 *ALL-ENTITY-STATES* 列表中进行处理。实体状态的结构如图 24 所示。

<标识符>	STATUS	CONTRL-STACK	DEFAULT-LIST	DEFAULT-POINTER	VALUE-STACK	E-VERDICT	TIMER-GUARD	DATA-STATE	TIMER-STATE	SNAP-DONE

图 24/Z.143－实体状态的结构

*<identifier>*是实体的唯一标识符，即测试系统中测试部件的模块控制。当模块开始执行或测试用例通过 `execute` 已经来执行时，便隐性地为模块控制、`mtc` 和测试 `system` 创建了这种唯一的标识符。标识符用于识别和寻址测试系统中的实体，如对带 `to` 子句的 `send` 操作或带 `from` 子句的 `receive` 操作的情况。

STATUS 描述模块控制或测试部件是否 `ACTIVE`、`SNAPSHOT`、`REPEAT` 或 `BLOCKED`。模块控制在测试用例执行期间是阻塞的。测试部件在其它测试部件创建期间可能是阻塞的，即在执行 `create` 操作期间。`SNAPSHOT` 状态表示部件是活动的，但是在快照的计算阶段。`REPEAT` 状态表示部件是活动的，因 `repeat` 语句，在 alt 指令中都应进行重新计算。

CONTROL-STACK 是一个流程图节点引用堆栈。*CONTROL-STACK* 中的顶层元素是下一个要解释的流程图节点。要求堆栈以适当的方式来构建函数调用的模型。

DEFAULT-LIST 是一个激活缺省值的列表，也就是说，它是一个指针列表，指向激活缺省值的起始节点。列表按反向顺序激活，也就是说，第一个被激活的缺省值是列表的最后一个元素。

在缺省机制执行期间，如果实际的缺省值不是成功终止，那么 *DEFAULT-POINTER* 指向下一个要计算的缺省值。

VALUE-STACK 是一个所有可能类型值的堆栈，它允许中途保存操作、函数和语句的最终或中间结果。例如，一个表达式的计算结果或 **mtc** 操作的结果将推入 *VALUE-STACK* 栈中。除了模块中已知的所有数据类型的值，我们还定义了特殊的值 **MARK** 作为栈字母表的一部分。当留出一个空间单元时，**MARK** 用来清空 *VALUE-STACK*。

E-VERDICT 保存测试部件实际的局部判定。当实体状态表示模块控制时，忽略 *E-VERDICT*。

TIMER-GUARD 表示特殊的定时器，它是确保测试用例执行时间和调用操作持续时间所必须的。*TIMER-GUARD* 作为一个定时器绑定来建模（见第 8.3.2.4 节和图 28）。

DATA-STATE 被认为是变量绑定列表的一个列表。列表结构的列表反映了因内嵌函数调用而出现的内嵌空间单元。变量绑定列表列表中的每个列表描述已知的变量及其在一个确定空间单元中的值。进入或离开一个空间单元对应在 *DATA-STATE* 中增加或删除一个变量绑定列表。实体状态 *DATA-STATE* 部分的描述请参见第 8.3.2.2 节。

TIMER-STATE 被认为是定时器绑定列表的一个列表。列表结构的列表反映了因内嵌函数调用而出现的内嵌空间单元。定时器绑定列表列表中的每个列表描述已知定时器及其在一个确定空间单元中的状态。进入或离开一个空间单元对应在定时器状态中增加或删除一个定时器状态列表。实体状态的定时器状态部分的描述请参见第 8.3.2.4 节。

SNAP-DONE 支持测试部件的快照语义。当照快照时，模块状态 *DONE* 列表的一个拷贝将被赋给 *SNAP-DONE*，即 *SNAP-DONE* 是已停止部件的一个部件标识符列表。

8.3.2.1 访问实体状态

<*identifier*>是实体状态的唯一标识符，它可用来访问由实体状态和实体状态不同部分表示的部件。

实体状态的 *STATUS*、*DEFAULT-POINTER*、*E-VERDICT* 和 *TIMER-GUARD* 部分当作全局可见的变量加以处理，也就是说，通过使用“点”记法，如 *myEntity*.*STATUS*、*myEntity*.*DEFAULT-POINTER* 和 *myEntity*.*E-VERDICT*，其中的 *myEntity* 指的是实体状态，可以获取或改变 *STATUS*、*DEFAULT-POINTER* 和 *E-VERDICT* 的值。

注 — 下面，我们假设能够通过使用引用和唯一的标识符来使用“点”记法。例如，在*myEntity*.*STATUS*中，*myEntityState* 可以是指向实体状态的指针，或者是<*identifier*>字段的值。

实体状态 *myEntity* 的 *CONTROL-STACK*、*DEFAULT-LIST* 和 *VALUE-STACK* 可通过使用“点”记法 *myEntity*.*CONTROL-STACK*、*myEntity*.*DEFAULT-LIST* 和 *myEntity*.*VALUE-STACK*.来访问。

通过使用 *push*、*pop*、*top*、*clear* and *clear-until* 等栈操作，可以访问和使用 *CONTROL-STACK* 和 *VALUE-STACK*。栈操作具有下列含义：

- myStack.push(item) 将条目推入myStack；
- myStack.pop() 从myStack弹出条目；
- 如果myStack 是空的，那么myStack.top() 返回myStack 的顶层元素或**NULL**；
- myStack.clear() 清除myStack，即从myStack 弹出所有条目；
- myStack.clear-until(item) 从myStack弹出条目，直至找到条目或myStack变空。

通过使用列表操作 *add*、*append*、*delete*、*member*、*first*、*length*、*next*、*random* 和 *change*，可以访问和使用 *DEFAULT-LIST*。这些列表操作的含义在第 8.3.1.1 节做了定义。

为了创建一个新的实体状态，假定 *NEW-ENTITY* 函数是可用的：

- *NEW-ENTITY* (*entityIdentifier, flow-graph-node-reference*)；

创建一个新的实体状态，并返回其引用。新的实体状态的部件拥有以下值：

- <标识符>设为*entityIdentifier*，将是一个全局唯一的标识符；
- *STATUS*设为**ACTIVE**；
- *flow-graph-node-reference* 是*CONTROL-STACK* 中唯一的（顶层）元素；
- *DEFAULT-LIST* 是一个空列表；
- *DEFAULT-POINTER* 拥有**NULL**值；
- *VALUE-STACK* 是一个空栈；
- *E-VERDICT* 设为**none**；
- *TIMER-GUARD* 是一个带有GUARD 名称、**IDLE**状态且没有缺省持续时间的新的定时器绑定（见第8.3.2.4节）；
- *DATA-STATE*是一个空列表；
- *TIMER-STATE*是一个空列表；
- *SNAP-DONE*是一个空列表。

在遍历流程图期间，*CONTROL-STACK* 通常以相同的方式来改变其值：顶层元素从 *CONTROL-STACK* 中弹出，并将弹出节点的后继节点推入 *CONTROL-STACK*。这一系列栈操作封装在 *NEXT-CONTROL* 函数中：

```
myEntity.NEXT-CONTROL(myBool) {
      successorNode := myEntity.CONTROL-STACK.NEXT(myBool).top();
      myEntity.CONTROL-STACK.pop();
      myEntity.CONTROL-STACK.push(successorNode);
}
```

8.3.2.2 数据状态和变量绑定

如图 25 所示，实体状态的数据状态 *DATA-STATE* 是一个变量绑定列表的列表。各变量绑定列表定义在特定范围单元内的变量绑定。增加一个新的变量绑定列表，对应输入一个新的范围单元，例如调用一个函数。删除一个变量绑定列表，对应留下一个范围单元，例如函数执行 **return** 语句。

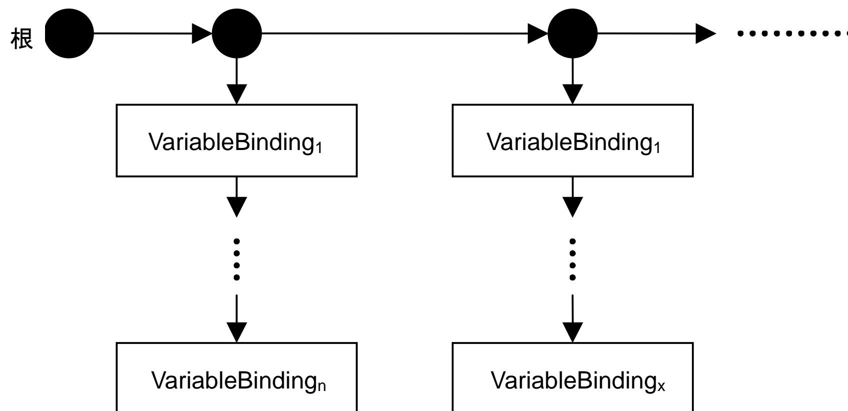


图 25/Z.143－实体状态*DATA-STATE*部分的结构

变量绑定的结构如图 26 所示。变量拥有一个名称、一个<*location*>和一个 *VALUE*。*VAR-NAME* 确定范围单元内的一个变量。<*location*>是变量值存储位置的唯一标识符。变量绑定的 *VALUE* 部分描述变量的实际值。

注 — 当变量被声明时，应自动提供唯一的位置标识符。

VAR-NAME	<位置>	VALUE

图 26/Z.143－变量绑定的结构

区分变量名称与位置之间的差别，以便构建函数调用的模型，以及构建以适当方式用值和引用参数化来执行测试用例的模型：

a） 对通过值来传入的参数，当作一个新变量的声明来处理，也就是说，将一个新的变量绑定附加至所调用函数或所执行测试用例范围的变量绑定列表中。新的变量绑定使用正式的参数名称作为*VAR-NAME*，接收一个新的位置，并获取传入函数或测试用例的值。

b） 通过引用来传入的参数也将导致在所调用函数或所执行测试用例范围中的新变量绑定。新变量绑定也使用正式的参数名称作为*VAR-NAME*，但不接收新的位置和新的值。新变量绑定获得<位置>的一个拷贝以及通过引用传入之变量的*VALUE*。

当更新一个变量值时，即在变量赋值的情况下，变量名称用于确定一个位置，并同时更新带有相同位置的所有变量绑定。因此，当留下范围单元时，属于该范围单元的变量列表可以在无需进一步更新的情况下予以删除。由于这一更新程序，通过引用传入的变量将自动具有更正值。

8.3.2.3 访问数据状态

通过使用“点”记法 *myEntity.myVar.VALUE*，可以获取一个变量的值，当中，*myEntity* 指的是实体状态，*myVar* 指的是变量名称。

为处理变量和变量范围，考虑定义以下函数：

a） *VAR-SET* 函数： *myEntity.VAR-SET* (*myVar*, *myValue*)

将实体*myEntity* 实际范围中*myVar* 变量的*VALUE* 部分设为*myVal*。此外，所有与变量*myVar*具有相同位置的变量的*VALUE*部分，都将设为*myVal*。

b） *INIT-VAR* 函数： *myEntity.INIT-VAR* (*myVar, myVal*)

为实体*myEntity* 实际范围单元中带有初始值*myVal* 的变量*myVar* 创建一个新的变量绑定。与*myVal* 一样使用关键字**NONE**意味着创建一个带有未定义初始值的变量。自动创建一个新的、唯一的<位置>值。

c） *GET-VAR-LOC* 函数： *myEntity.GET-VAR-LOC* (*myVar*)

获取*myEntity*所拥有的变量*myVar* 的位置。

d） *INIT-VAR-LOC* 函数： *myEntity.INIT-VAR-LOC* (*myVar,myLoc*)

为实体*myEntity* 实际范围单元中带有位置*myLoc*的变量*myVar* 创建一个新的变量绑定。变量将用带有位置*myLoc* 的另一个变量值来初始化。

注 — 带有相同位置的变量是用引用进行参数化的结果。由于如第8.3.2.2节所述那样对引用参数进行处理，因此所有带有相同位置的变量都将在其生命周期期间拥有相同的值。

e） *INIT-VAR-SCOPE*函数： *myEntity.INIT-VAR-SCOPE* ()

初始化一个处于实体*myEntity* 数据状态的新的变量范围，即在变量绑定列表列表中附加一个空列表，作为第一个列表。

f） *DEL-VAR-SCOPE* 函数： *myEntity.DEL-VAR-SCOPE* ()

删除*myEntity* 数据状态的变量范围，即删除变量绑定列表列表中的第一个列表。

8.3.2.4 定时器状态和定时器绑定

如图 27 和图 25 所示，实体状态的定时器状态 *TIMER-STATE* 和数据状态 *DATA-STATE* 是可比的。它们都是绑定列表中的一个列表，各绑定列表定义特定范围中的有效绑定。增加一个新的列表对应输入一个新的范围单元，删除一个绑定列表对应留下一个范围单元。

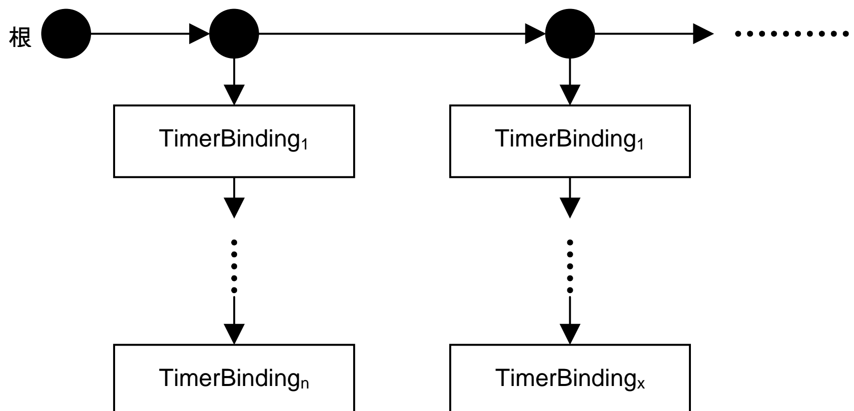


图 27/Z.143－实体状态*TIMER-STATE*部分的结构

定时器绑定的结构如图 28 所示。*TIMER-NAME* 和<位置>的含义类似于变量绑定的 *VAR-NAME* 和<位置>的意义（见图 26）。

TIMER-NAME	<位置>	STATUS	DEF-DURATION	ACT-DURATION	TIME-LEFT	SNAP-VALUE	SNAP-STATUS

图 28/Z.143－定时器绑定的结构

STATUS 表示定时器是活动的、非活动的或者已超时。对应的 *STATUS* 值为 **IDLE**、**RUNNING** 和 **TIMEOUT**。*DEF-DURATION* 描述定时器的缺省持续时间。*ACT-DURATION* 用一个已经启动的运行定时器来存储实际的持续时间。*TIME-LEFT* 描述在定时器超时之前运行定时器需要运行的实际持续时间。

注 — 如果定时器声明不带缺省持续时间，那么*DEF-DURATION* 是未定义的。如果定时器已停止或超时，那么 *ACT-DURATION* 和*TIME-LEFT* 设为0.0。如果不带持续时间的定时器已启动，那么*DEF-DURATION* 的值拷贝至 *ACT-DURATION*。如果不带定义的持续时间的定时器已启动，那么将发生一个动态错误。

需要 *SNAP-VALUE* 和 *SNAP-STATUS* 支持 TTCN-3 的快照语义。照快照时，*SNAP-VALUE* 获得 *ACT-DURATION* – *TIME-LEFT* 的实际值。*SNAP-STATUS* 获得与 *STATUS* 相同的值。快照计算将只依据 *SNAP-VALUE* 和 *SNAP-STATUS* 中的值。

定时器只能通过引用传入函数中，即该机制类似于第 8.3.2.2 节中所述的变量机制。这意味着创建一个新的定时器绑定（带形式参数名称），它从通过引用传入的定时器那里获得<位置>、*STATUS*、*DEF-DURATION*、*ACT-DURATION*、*TIME-LEFT*、*SNAP-VALUE* 和 *SNAP-STATUS* 的拷贝。当更新定时器时，所有带相同<位置>值的定时器绑定都将同时得到更新。

8.3.2.5 访问定时器状态

通过使用“点”记法，可以获取定时器 *myTimer* 的 *STATUS*、*DEF-DURATION*、*ACT-DURATION*、*TIME-LEFT*、*SNAP-VALUE* 和 *SNAP-STATUS* 值：

- myEntity.myTimer.STATUS；
- myEntity.myTimer.DEF-DURATION；
- myEntity.myTimer.ACT-DURATION；
- myEntity.myTimer.TIME-LEFT；
- myEntity.myTimer.SNAP-VALUE；

- myEntity.myTimer.SNAP-STATUS。

点记法中的 *myEntity* 指的是表示拥有定时器 *myTimer* 的测试部件或模块控制状态的实体状态。

为了改变定时器 *timer-name* 的 *STATUS*、*DEF-DURATION*、*ACT-DURATION*、*TIME-LEFT*、*SNAP-VALUE* 和 *SNAP-STATUS* 值，必须使用通用的 *TIMER-SET* 操作，例如：

- *myEntity.TIMER-SET(myTimer, STATUS, myVal).*

将 *myEntity* 实际范围中定时器 *myTimer* 的 *STATUS* 值设为 *myVal*。此外，带有与定时器 *myTimer* 相同位置的所有定时器，其 *STATUS* 值也设为 *myVal*。*TIMER-SET* 函数还可用来改变 *DEF-DURATION*、*ACT-DURATION*、*TIME-LEFT*、*SNAP-VALUE* 和 *SNAP-STATUS* 的值。

为了处理定时器、定时器范围和快照，必须定义以下函数：

a） *INIT-TIMER*函数： *myEntity.INIT-TIMER* (*myTimer, myDuration*)

为实体*myEntity*实际范围中带有缺省持续时间*myDuration*的定时器*myTimer*创建一个新的定时器绑定。与*myDuration*一样使用关键字**NONE**意味着创建一个不带缺省持续时间的定时器。

b） *GET-TIMER-LOC*函数： *myEntity.GET-TIMER-LOC* (*myTimer*)

获取*myEntity*所拥有的定时器*myTimer*的位置。

c） *INIT-TIMER-LOC*函数： *myEntity.INIT-TIMER-LOC* (*myTimer, myLocation*)

为*myEntity*实际范围单元中带有位置*myLocation*的定时器*myTimer*创建一个新的定时器绑定。定时器将用带位置<*位置*>的另一个定时器的*STATUS*、*DEF-DURATION*、*ACT-DURATION* 和 *TIME-LEFT* 值进行初始化。

注 — 带相同位置的定时器是用引用进行参数化的结果。由于按第8.3.2.3节中所述的那样来处理定时器引用参数，因此在其生命周期期间，带相同位置的所有定时器都将具有相同的*STATUS*、*DEF-DURATION*、*ACT-DURATION* 和 *TIME-LEFT* 值。

d） *INIT-TIMER-SCOPE*函数： *myEntity.INIT-TIMER-SCOPE* ()

初始化处于实体*myEntity*定时器状态的新定时器范围，即附加一个空列表，作为定时器绑定列表列表中的第一个列表。

e） *DEL-TIMER-SCOPE*函数： *myEntity.DEL-TIMER-SCOPE* ()

删除实体*myEntity*定时器状态的定时器范围，即删除定时器绑定列表列表中的第一个列表。

f） *SNAP-TIMER*函数： *myEntity.SNAP-TIMER* ()

更新*myEntity*所拥有的全部定时器中的*SNAP-VALUE* 和 *SNAP-STATUS*，即：

```
myEntity.SNAP-TIMERS () {
 for all myTimer in TIMER-STATE {
     myEntity.myTimer.SNAP-VALUE := myEntity.myTimer.ACT-DURATION –
     myEntity.myTimer.TIME-LEFT;
     myEntity.myTimer.SNAP-STATUS := myEntity.myTimer.STATUS;
 }
```

8.3.3 端口状态

端口状态用于描述端口的实际状态。在模块状态中，端口状态在 *ALL-PORT-STATES* 列表中进行处理（见图23）。端口状态的结构如图 29 所示。*PORT-NAME* 指的是端口名称，它由拥有该端口的测试部件 *OWNER* 用来标识端口。*STATUS* 提供端口的实际状态。一个端口或者是 **STARTED** 的，或者是 **STOPPED**。

注 — 测试系统中的端口，由拥有测试部件的<所有者>及对<所有者>而言是本地的端口名称<端口名称>来唯一地确定。

端口状态的 *CONNECTIONS-LIST* 明了测试系统中不同端口之间的连接。第 8.3.3.1 节解释了这种机制。

处于端口状态的 *VALUE-QUEUE* 用于存储消息、调用、应答和异常，它们在该端口上予以接收，但尚未消耗。

SNAP-VALUE 支持 TTCN-3 的快照机制。当照快照时，将 *VALUE-QUEUE* 中的第一个元素拷入 *SNAP-VALUE* 中。如果 *VALUE-QUEUE* 为空，或者 *STATUS* 为 **STOPPED**，那么 *SNAP-VALUE* 将获得值 **NULL**。

PORT-NAME	OWNER	STATUS	CONNECTIONS-LIST	VALUE-QUEUE	SNAP-VALUE

图 29/Z.143－端口状态的结构

8.3.3.1 端口间连接的处理

通过 **connect** 操作，连接它们当中的两个端口，来实现两个测试部件之间的连接。因此，一个部件之后可以使用其本地端口名称来解决远程排队问题。如图 30 所示，处于连接状态的两个队列，可以用一对 *REMOTE-ENTITY* 和 *REMOTE-PORT-NAME* 来表示连接。*REMOTE-ENTITY* 是拥有远程端口的测试部件的唯一标识符。*REMOTE-PORT-NAME* 指的是由 *REMOTE-ENTITY* 用于解决排队问题的本地名称。TTCN-3 支持端口的一对多连接，因此，在列表中对端口的所有连接进行组织。

注 1 — 由**map** 操作建立的连接也在连接列表中进行处理。**map** 操作：**map**(*PTC1*:*MyPort*, **system**.*PCO1*) 致使在由 *PTC1*拥有的*MyPort* 端口状态中建立一个新的连接（**system**、*PCO1*）。与*PCO1*相连的远方一侧，驻留在SUT内。其行为超出了本语义的范围。

注 2 — 操作语义将关键字**system** 作为一个符号地址来处理。端口连接列表中的连接（**system**、*myPort* ）表示端口映射至测试系统接口中的端口*myPort*上。

REMOTE-ENTITY	REMOTE-PORT-NAME

图 30/Z.143－连接的结构

8.3.3.2 端口状态的处理

通过使用已知的队列操作 *enqueue*、*dequeue*、*first* 和 *clear*，可以访问和处理端口状态中的值队列。使用 *GET-PORT* 或 *GET-REMOTE* 函数来引用要访问的队列。

注 1 — 队列操作*enqueue*、*dequeue*、*first* 和 *clear* 具有如下含义：

- *myQueue*.*enqueue*(条目) 将条目作为最后一个条目加入到*myQueue* 中；
- *myQueue*.*dequeue*() 从*myQueue* 中删除第一个条目；
- 如果*myQueue*为空，那么*myQueue*.*first*() 返回*myQueue* 中的第一个条目或**NULL**；
- *myQueue*.*clear*() 从*myQueue* 移去所有元素。

以下函数支持对端口状态的处理：

a） *NEW-PORT* 函数：*NEW-PORT* (*myEntity, myPort*)
建立一个新端口并返回其引用。新端口由*myEntity*拥有，并命名为*myPort*，指的是由测试部件*myEntity*和端口名称*myPort*所确定的端口。新端口的状态为**STARTED**。*CONNECTIONS-LIST* 和 *VALUE-QUEUE* 为空。*SNAP-VALUE* 的值为**NULL**（即新端口的输入队列为空）。

b） *GET-PORT* 函数：*GET-PORT* (*myEntity, myPort*)
向由测试部件*myEntity*确定的端口返回一个引用，该测试部件拥有端口和端口名称*myPort*。

c） *GET-REMOTE-PORT*函数：*GET-REMOTE-PORT* (*myEntity, myPort, myRemoteEntity*)
向由测试部件*myRemoteEntity*拥有的端口返回引用，并与由*myEntity*和*myPort*确定的端口相连接。如果远程端口映射至测试系统接口的一个端口，那么返回符号地址**SYSTEM**。

注 2 — 如果不存在远程端口，或者无法唯一确定远程端口，那么*GET-REMOTE-PORT*返回**NULL**。如果远程实体是未知的或不需要，即对该端口只存在一对一连接，那么特殊的值**NONE**可用作*myRemoteEntity*参数的值。

d） 端口的*STATUS* 当作一个变量来处理。它可通过带*GET-PORT* 调用的、符合条件的*STATUS*来解决：
GET-PORT(*myEntity*, *myPort*).*STATUS*

e） *ADD-CON*函数：*ADD-CON* (*myEntity, myPort, myRemoteEntity*, *myRemotePort*)
向由*myEntity* 拥有的端口*myPort* 连接列表增加一个连接（*myRemoteEntity*、*myRemotePort*）。

f） *DEL-CON* 函数：*DEL-CON* (*myEntity, myPort, myRemoteEntity*, *myRemotePort*)

从由*myEntity* 拥有的端口*myPort* 连接列表中移去一个连接（*myRemoteEntity*、*myRemotePort*）。

g） *SNAP-PORTS* 函数：*SNAP-PORTS* (*myEntity*)

为*myEntity* 拥有的所有端口更新*SNAP-VALUE* ，即：

```
SNAP-PORTS (myEntity) {
    for all ports p         /* 处于模块状态 */ {
        if (p.OWNER == myEntity) {
            if (p.STATUS == STOPPED) {
                p. SNAP-VALUE := NULL;
            }
            else {
                p. SNAP-VALUE := p.first()
            }
        }
    }
}
```

8.3.4 用于模块状态处理的通用函数

操作语义假定存在以下函数，用于处理模块状态。

注 1 — 在解释TTCN-3模块期间，只存在一种模块状态。假定模块状态的部件存储在全局变量中，并不是在一个复杂的数据对象中。因此，假定下列函数对全局变量起作用，并不解决特定的模块状态对象。

a） *DEL-ENTITY* 函数：*DEL-ENTITY*(*myEntity*)

删除带有唯一标识符*myEntity*的实体。删除操作包括：

— 删除*myEntity* 的实体状态；

— 删除由*myEntity* 拥有的所有端口；

— 删除涉及*myEntity*的所有连接。

b） *UPDATE-REMOTE-REFERENCES* 函数：

UPDATE-REMOTE-REFERENCES (*source, target*)

UPDATE-REMOTE-REFERENCES 更新两个实体中带相同位置的变量和定时器。用于更新的值是由*source* 拥有的变量和定时器的值。

注 2 — 在测试用例终止期间，使用*UPDATE-REMOTE-REFERENCES*。它允许对模块控制变量进行更新，该变量作为引用参数传给测试用例。

8.4 消息、程序调用、应答和异常

测试部件之间以及测试部件与 SUT 之间的信息交换与*消息*、*程序调用*、*程序调用应答*和*异常*有关。出于通信的目的，必须构建、编码和解码这些条目。具体的编码，即如何将 TTCN-3 数据类型映射至比特和字节，以及具体的解码，即如何将字节和比特映射至 TTCN-3 数据类型，均超出本操作语义的范围。在本建议书中，只在概念层面上处理*消息*、*程序调用*、*程序调用应答*和*异常*。

8.4.1 消息

消息与基于消息的通信有关。所有（预定义的和用户定义的）数据类型的值都可以在通信的实体之间进行交换。如图 31 所示，操作语义将消息作为由*发送方*、*类型*和*值*部分组成的结构化对象来处理。*sender* 部分用于确定消息的发送方实体，*type* 部分用于规定消息的类型，*value* 部分用于定义消息的值。

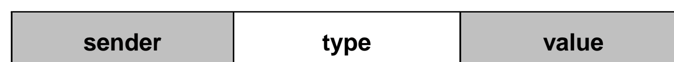


图 31/Z.143－消息的结构

注 — 操作语义只介绍TTCN-3概念的一种模型。*发送方*信息是否发送与/或接收以及如何发送与/或接收，取决于测试系统的实现，例如，在某些情况下，发送方信息可以是消息值部分的一部分，因此，它不是消息结构的独立部分。

8.4.2 程序调用与应答

程序调用和程序应答与基于程序的通信有关。像对带有表示参数的部件的记录值那样来定义它们。操作语义还像对结构化类型中的值那样来处理程序调用和程序调用应答。图 32 和图 33 表示了程序调用的结构和应答的结构。

sender	procedure-reference	parameter-part				
			in-or-inout-parameter$_1$	…	in-or-inout-parameter$_n$	

图 32/Z.143－程序调用的结构

sender	procedure-reference	parameter-part					value
			inout-or-out-parameter$_1$	…	inout-or-out-parameter$_n$		

图 33/Z.143－程序调用应答的结构

在以上两图中，*sender* 和 *procedure-reference* 部分具有相同的含义。*sender* 部分指的是调用或程序调用应答的发送方实体。*procedure-reference* 指的是调用和应答所属的程序。在图 32 中，程序调用的 *parameter-part* 指的是 **in** 参数和 **inout** 参数，在图 33 中，应答的 *parameter-part* 指的是调用与应答所属的 **inout** 参数和 **out** 参数。此外，应答还有一个有关程序应答返回值的 *value* 部分。

注 1 — 与之前注释所阐述的那样（见第8.4.1节），操作语义只介绍TTCN-3概念的一种模型。图32和图33中所述的信息是否发送与/或接收以及如何发送与/或接收，取决于测试系统的实现。

注 2 — 对程序调用，**out** 参数没有相关性，并在图32中被省略。对程序调用应答，**in**参数没有相关性，并在图33中被省略。

注 3 — 参数类型和返回值类型可以总是一致地来自相关的签名定义。

8.4.3 异常

异常也涉及基于程序的通信。异常的结构如图 34 所示。它由四个部分组成。*sender* 部分用于确定异常的发送方；*procedure-reference* 部分指的是异常所属的程序，*type* 部分用于确定异常的类型，*value* 部分提供异常的值。在程序引用部分中提到的程序签名用于定义允许的异常类型列表。收到的异常应符合所列的类型之一。通常，它可以是任何预定义或用户定义的 TTCN-3 数据类型。

sender	procedure-reference	type	value

图 34/Z.143－异常的结构

8.4.4 消息、程序调用、应答和异常的构建

用于发送消息、程序调用、程序调用应答或异常的操作是 **send**、**call**、**reply** 和 **raise**。所有这些发送操作都以相同的方式进行构建：

```
<port-name>.<sending-operation>(<send-specification>) [to <receiver>]
```

<port-name>和<sending-operation>定义用于发送一个条目的端口和操作。在一对多连接的情况下，需要规定<receiver>实体。通过使用<send-specification>，建立将要发送的条目。发送规范可使用具体的值、模板引用、可变值、常量、表达式、函数等，以建立和编码待发送的条目。

操作语义假设存在一个通用的 *CONSTRUCT-ITEM* 函数：

CONSTRUCT-ITEM (*myEntity*, <sending-operation>, <send-specification>)

依据`<sending-operation>`和`<send-specification>`（`<sending-operation>`和`<send-specification>`二者指的都是TTCN-3发送操作中的相应部分）返回消息、程序调用、程序调用应答或异常。实体引用*myEntity* 是待发送条目的发送方。还假设该*sender*信息是待发送条目的一部分（图31至图34）。

8.4.5 消息、程序调用、应答和异常的匹配

用于接收消息、程序调用、程序调用应答或异常的操作是 **receive**、**getcall**、**getreply** 和 **catch**。所有这些接收操作都以相同的方式进行构建：

```
<port-name>.<receiving-operation>(<matching-part>) [from <sender>] [<assignment-part>]
```

`<port-name>`和`<receiving-operation>`定义用于接收一个条目的端口和操作。在一对多连接的情况下，可以使用 **from** 子句来选择特定的发送方实体`<sender>`。待接收的条目必须满足`<matching-part>`中规定的条件，即它必须匹配。`<matching-part>`可使用具体的值、模板引用、可变值、常量、表达式、函数等，以推定匹配的条件。

操作语义假设存在一个通用的 *MATCH-ITEM* 函数：

MATCH-ITEM (*myItem*, `<matching-part>`, `<sender>`)

如果*myItem*满足`<matching-part>`的条件，如果`<sender>`已经发送*myItem*，那么返回**true**，否则返回**false**。

8.4.6 从收到的项中获取信息

来自接收到消息、程序调用、程序调用应答和异常的信息，可以在接收函数 **receive**、**getcall**、**getreply** 和 **catch** 的`<assignment-part>`（见第 8.4.5 节）中获取。`<assignment-part>`描述程序调用和应答的参数如何返回在应答、消息、异常中编码的值，以及`<sender>`实体的标识符如何赋给各变量。

操作语义假设存在一个通用的 *RETRIEVE-INFO* 函数：

RETRIEVE-INFO (*myItem*, `<assignment-part>`)

获取所有根据`<assignment-part>`要获取的值，并赋给赋值部分中所列的变量。通过 *VAR-SET* 操作来完成赋值，即带相同位置的变量同时得以更新。

8.5 函数、可选步骤和测试用例的调用记录

根据其名称和实际参数列表，调用（或执行）函数、可选步骤和测试用例。实际参数为引用参数提供引用，并为值参数提供具体的值，这在函数或测试用例定义的形式参数中进行定义。操作语义通过使用图 35 中所示的调用记录来处理函数、可选步骤和测试用例的调用。*BEHAVIOUR-ID* 值是函数或测试用例的名称，值参数为形式参数<*parId*$_1$> ... <*parId*$_n$>提供具体的值<*parId*$_1$> ... <*parId*$_n$>。引用参数为现有变量和定时器的位置提供引用。在能够执行函数或测试用例之前，必须建立适当的调用记录。

behaviour-id	value-parameter					reference-parameter				
		parId$_1$	…	parId$_n$			parId$_1$	…	parId$_n$	
		value$_1$	…	value$_n$			loc$_1$	…	loc$_n$	

图 35/Z.143－调用记录的结构

8.5.1 调用记录的处理

函数或测试用例名称以及实际参数值，可以通过使用点记法来获取，例如 *myCallRecord.parId*$_n$ 或 *myCallRecord.behaviour-id*，其中，*myCallRecord* 是指向调用记录的指针。

对于调用的结构，假设 NEW-CALL-RECORD 函数是可用的：

NEW-CALL-RECORD(*myBehaviour*)

为函数或测试用例*myBehaviour* 创建一个新的调用记录，并返回一个指向新记录的指针。新调用记录的参数字段拥有未定义的值。

myEntity.*INIT-CALL-RECORD*(*myCallRecord*)

为在测试部件或模块控制*myEntity*的实际范围中值处理和参数引用创建变量和定时器。用调用记录中提供的相应值来初始化用于值参数处理的变量。处理引用参数的变量与定时器获得所提供的位置。此外，它们还获得部件另一个范围单元中现有变量或定时器的一个值，在该范围单元中创建调用记录。

8.6 TTCN-3模块的计算程序

8.6.1 计算步骤

TTCN-3 模块的计算程序结构上分成：

1） 初始化阶段；

2） 更新阶段；

3） 选择阶段；以及

4） 执行阶段。

在模块控制终止前，重复第 2）、第 3）、第 4）阶段。通过综合非正式文本、伪代码和在之前各条中引入的函数，来描述计算程序。

8.6.1.1 步骤I：初始化

初始化阶段包括下列动作：

a） 变量的声明和初始化：

— *INIT-FLOW-GRAPHS*()；　　// 流程图处理的初始化。
　　// 第 8.6.2 节对 *INIT-FLOW-GRAPHS* 做了解释。

— *Entity* := **NULL**;　　// 用于指实体状态的*Entity*。实体状态或者表示
　　// 模块控制，或者表示测试部件。

注 — 以下全局变量形成模块状态：*ALL-ENTITY-STATES*、*ALL-PORT-STATES*、*MTC*、*TC-VERDICT* 和 *DONE*，在解释TTCN-3模块期间对模块状态进行处理（见第8.3.1节）。

— ALL-ENTITY-STATES := **NULL**;

— ALL-PORT-STATES := **NULL**;

— *MTC* := **NULL**;

— *TC-VERDICT* := **none**;

— *DONE* := **NULL**;

— SNAP-DONE := 0;

b） 模块控制的创建和初始化

— *Entity*:= *NEW-ENTITY* (*GET-UNIQUE-ID*(),*GET-FLOW-GRAPH* (<*moduleId*>));
　　// 创建一个新的实体状态，并用表示名为
　　// <*moduleId*>的模块的控制行为的、流程图
　　// 起始节点进行初始化。第 8.6.2 节对
　　// *GET-UNIQUE-ID* 做了解释。

— *Entity*.*INIT-VAR-SCOPE*()；　　// 新变量范围

— *Entity*.*INIT-TIMER-SCOPE*()；　　// 新定时器范围

— *Entity*.*VALUE-STACK*.*push*(**MARK**)；　　// 推入值堆栈的标记

— *ALL-ENTITY-STATES*.*append*(*Entity*)；　　// 推入模块状态的新实体。

8.6.1.2 步骤II：更新

更新阶段涉及超出本操作语义范围但影响 TTCN-3 模块解释的所有动作。更新阶段包括以下动作：

a) **时间进程**：更新所有运行定时器，也就是说，（可能）降低运行定时器的*TIME-LEFT* 值，如果因更新定时器到期，那么更新相应的定时器绑定，即*TIME-LEFT*设为0.0，*STATUS*设为`TIMEOUT`；

注 1 — 定时器更新包括更新所有处于模块状态的运行*TIMER-GUARD* 定时器。*TIMER-GUARD* 定时器用于保护测试用例的运行和调用操作。

b) **SUT的行为**：将（可能）从SUT收到的消息、远程程序调用、对远程程序调用的应答，以及异常，都放入端口队列中，在该队列上将进行相应的接收。

注 2 — 本操作语义没有对时间进程和SUT行为做任何假设。

8.6.1.3 步骤III：选择

选择阶段由以下两个动作组成：

a) **选择**：选择一个非阻塞的实体，即具有*STATUS* 值为**ACTIVE** 或**SNAPSHOT** 的实体；

b) **存储**：存储在全局变量*Entity* 中所选实体的标识符。

8.6.1.4 步骤IV：执行

执行阶段由以下两个动作组成：

a) **所选实体的执行步骤**：执行*Entity* *CONTROL-STACK* 中的顶层流程图节点；

b) **检查终止准则**：如果模块控制已终止，即实体状态的列表为空，那么停止执行，否则继续阶段II。

注 — 所选实体的执行步骤可视为程序调用。当执行步骤终止时，即返回控制，对终止准则进行检查。

8.6.2 全局函数

计算程序使用 *INIT-FLOW-GRAPHS* 和 *GET-UNIQUE-ID* 全局函数：

a) *INIT-FLOW-GRAPHS*假设为初始化流程图处理的函数。处理可包括创建流程图、处理指向流程图指针和流程图节点。

b) *GET-UNIQUE-ID*假设为每次当它被调用时返回一个唯一标识符的函数。唯一标识符可以以计数变量的形式来实现，计数变量在每次调用*GET-UNIQUE-ID* 时增加和返回。

下面各条中使用的、用于描述流程图节点执行的伪代码，使用 *CONTINUE-COMPONENT*、*RETURN*、*****动态错误*****函数：

a) *CONTINUE-COMPONENT*：实际的测试部件用节点继续其执行，节点取决于控制栈的顶层，也就是说，控制不会返回给本条中所描述的模块计算程序。

b) *RETURN*将控制返回给本条中所描述的模块计算程序。*RETURN*是计算阶段“选定实体的计算步骤”的最后动作。

c) *****动态错误*****指的是发生一个动态错误。错误处理过程本身超出了本操作语义的范围。如果发生一个动态错误，那么测试用例行为之后的所有行为意味着都是未加定义的。在这种情况下，应清除分配给测试用例的资源，并将**error** 判定赋给测试用例。将控制提供给跟在发生了错误的执行语句之后的、控制部分中的语句。这通过流程图片段<动态错误>来建模（第9.18b节）。

注 — 动态错误的出现与测试行为有关。操作语义规定的动态错误表示在TTCN-3使用上的问题，如错误使用或竞争条件。

d) *`APPLY-OPERATOR`* 当作通用函数使用，用于描述表达式中的运算符计算（如+、*、/ 或 –）（见第9.18.4节）。

9 TTCN-3构件的流程图片段

操作语义以流程图的方式来表述 TTCN-3 行为。表述行为的流程图的构建算法在第 8.2 节中进行描述。它基于流程图和流程图片段模板，为构建 TTCN-3 模块中所定义的模块控制、测试用例、可选步骤、函数和部件类型定义的具体流程图，必须使用这些流程图和流程图片段。可以在本条款中找到有关流程图片段模板的定义。它们按字母次序而不是按逻辑次序出现。

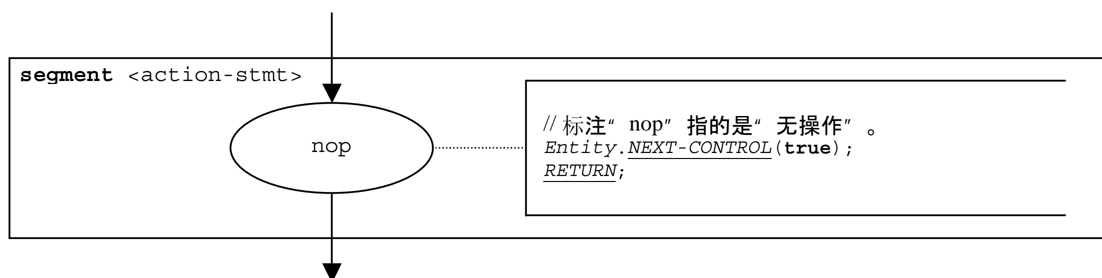
流程图片段定义以图的形式提供。流程图节点出现在图的左侧，节点相关的注释和流程线显示在图的右侧。以基本节点相关的伪代码形式来为引用节点和注释提供描述性注释。伪代码描述如何解释一个基本节点，即改变模块状态。对 TTCN-3 模块，它利用第 8 节中定义的函数和 TTCN-3 模块计算过程中声明和初始化的全局变量（见第 8.6 节）。在第 8 节中，对伪代码使用的所有函数和关键字做了综述。

9.1 Action 语句

action 语句的语法结构是：

```
action (<informal description>)
```

图 36 中的流程图片段<action-stmt>对 **action** 语句的执行过程进行定义：



注 — **action**语句的<informal description>参数对操作语义没有任何意义，因此未在流程图片段中予以表述。

图 36/Z.143－流程图片段<action-stmt>

9.2 Activate 语句

activate 语句的语法结构是：

```
activate(<altstep-name>([<act-par-desc1>, ... , <act-par-descn>]))
```

<altstep-name>表示可选步骤的名称，激活之作为缺省行为，$<act\text{-}par\text{-}descr_1>$，...，$<act\text{-}par\text{-}descr_n>$描述在可选步骤激活之时它的实际参数值。

假设对每个$<act\text{-}par\text{-}desc_1>$，对应的形参标识符$<f\text{-}par\text{-}Id_1>$是已知的，也就是说，可将上述语法结构扩展为：

```
activate(<altstep-name>((<f-par-Id1>,<act-par-desc1>), ... , (<f-par-Idn>,<act-par-descn>)))
```

图 37 中的流程图片段<activate-stmt>对 **activate** 语句的执行过程进行定义。执行过程在结构上分为三个步骤。在第一个步骤中，创建可选步骤<function-name>的调用记录。在第二个步骤中，计算实际参数的值，并将之赋给调用记录中的对应字段。在第三个步骤中，将调用记录作为激活缺省之实体 <u>*DEFAULT-LIST*</u> 的第一个元素。

注 — 对激活为缺省行为的可选步骤，只允许值参数。在图37中，通过流程图片段<value-par-calculation>对值参数的处理情况进行描述，它在第9.24.1节中定义。

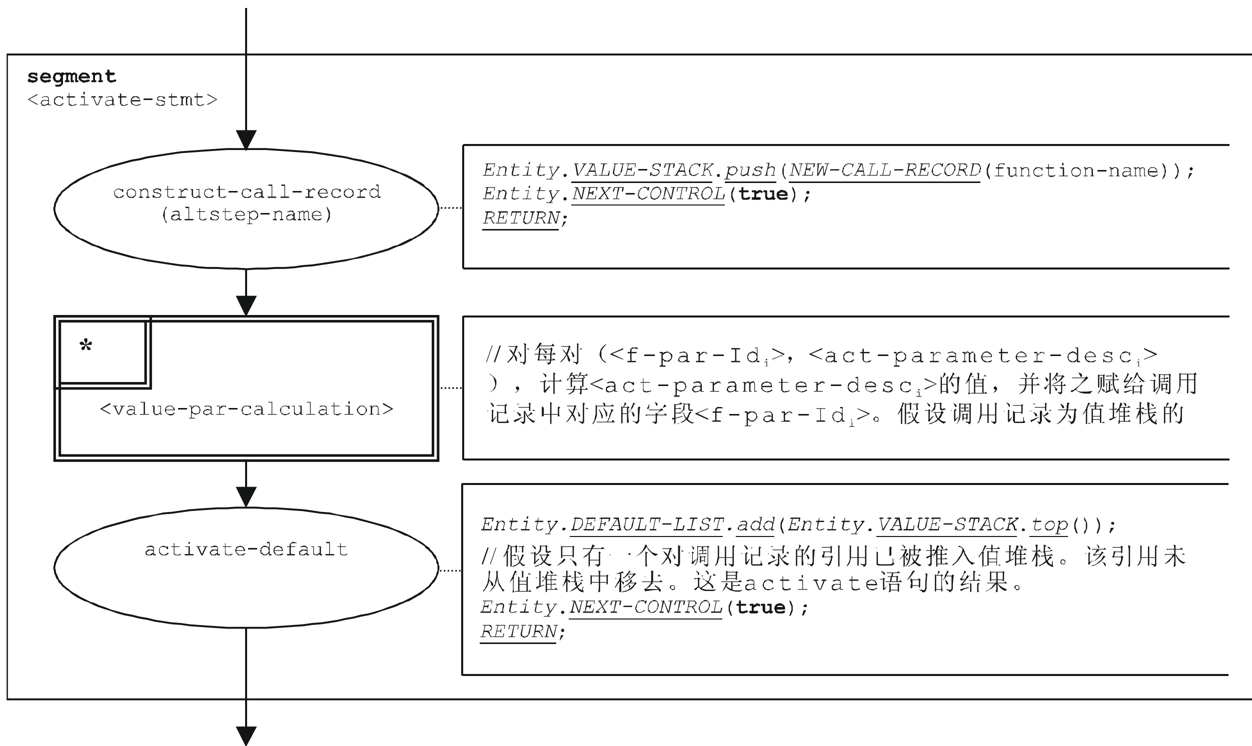


图 37/Z.143－流程图片段<activate-stmt>

9.3 Alt 语句

alt 语句是 TTCN-3 的最复杂和最重要语句。它执行快照语义，并依据消息、应答、调用和异常的接收、超时的发生以及部件的终止规定分支情况。此外，唤起 TTCN-3 缺省机制也与 **alt** 语句有关。

图 38 提供了 **alt** 语句的流程图表示。依据消息、应答、调用和异常接收、超时发生以及部件终止的不同选项藏在流程图片段<receiving-branch>中。

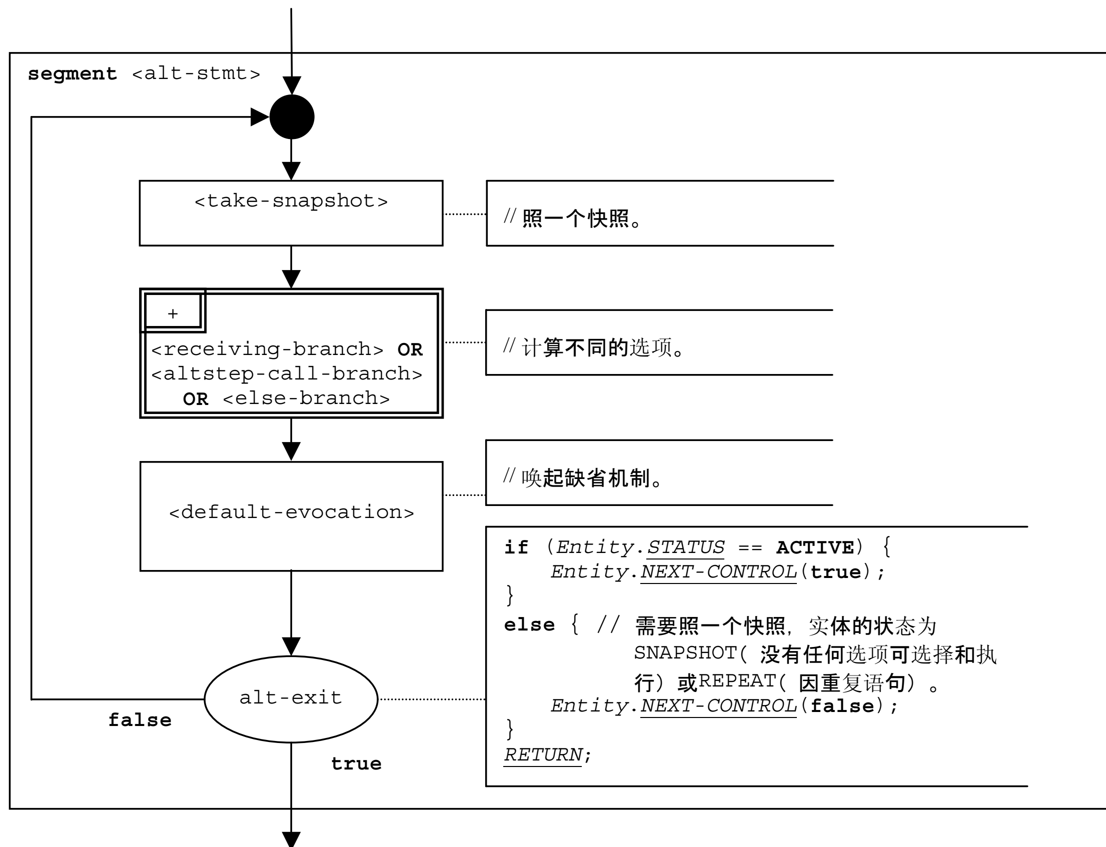


图 38/Z.143－流程图片段<alt-stmt>

9.3.1 流程图片段 <take-snapshot>

图 39 中的流程图片段<take-snapshot>照快照的过程。快照记录端口、定时器和已停止部件的值。

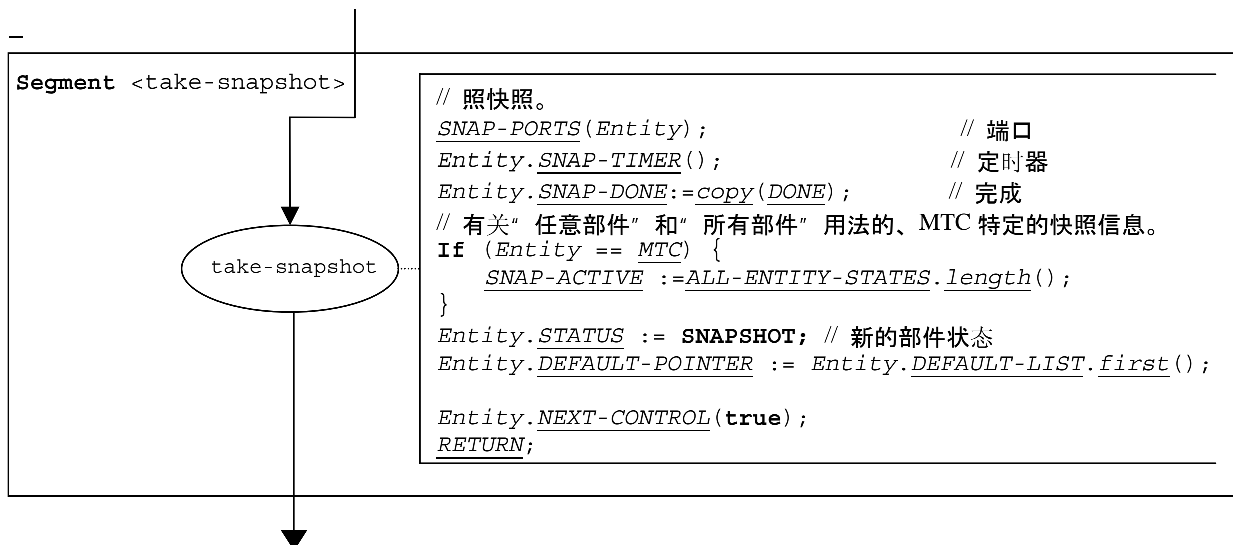


图 39/Z.143－流程图片段<take-snapshot>

9.3.2 流程图片段<receiving-branch>

流程图片段<receiving-branch>的执行过程如图 40 所示：

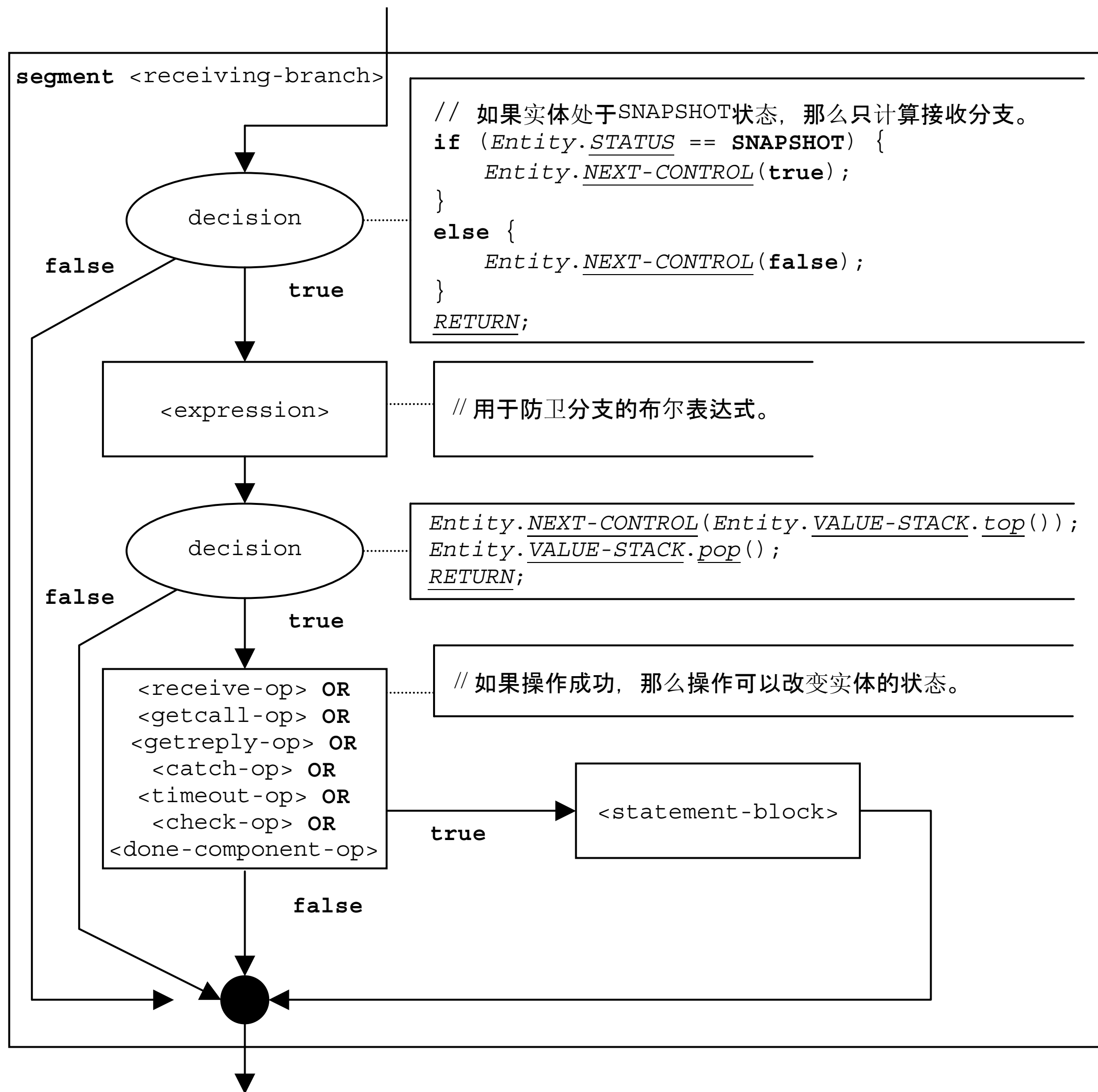


图 40/Z.143－流程图片段<receiving-branch>

9.3.3 流程图片段<altstep-call-branch>

图 41 中的流程图片段`<altstep-call-branch>`对 **alt** 语句内可选步骤的调用进行描述。

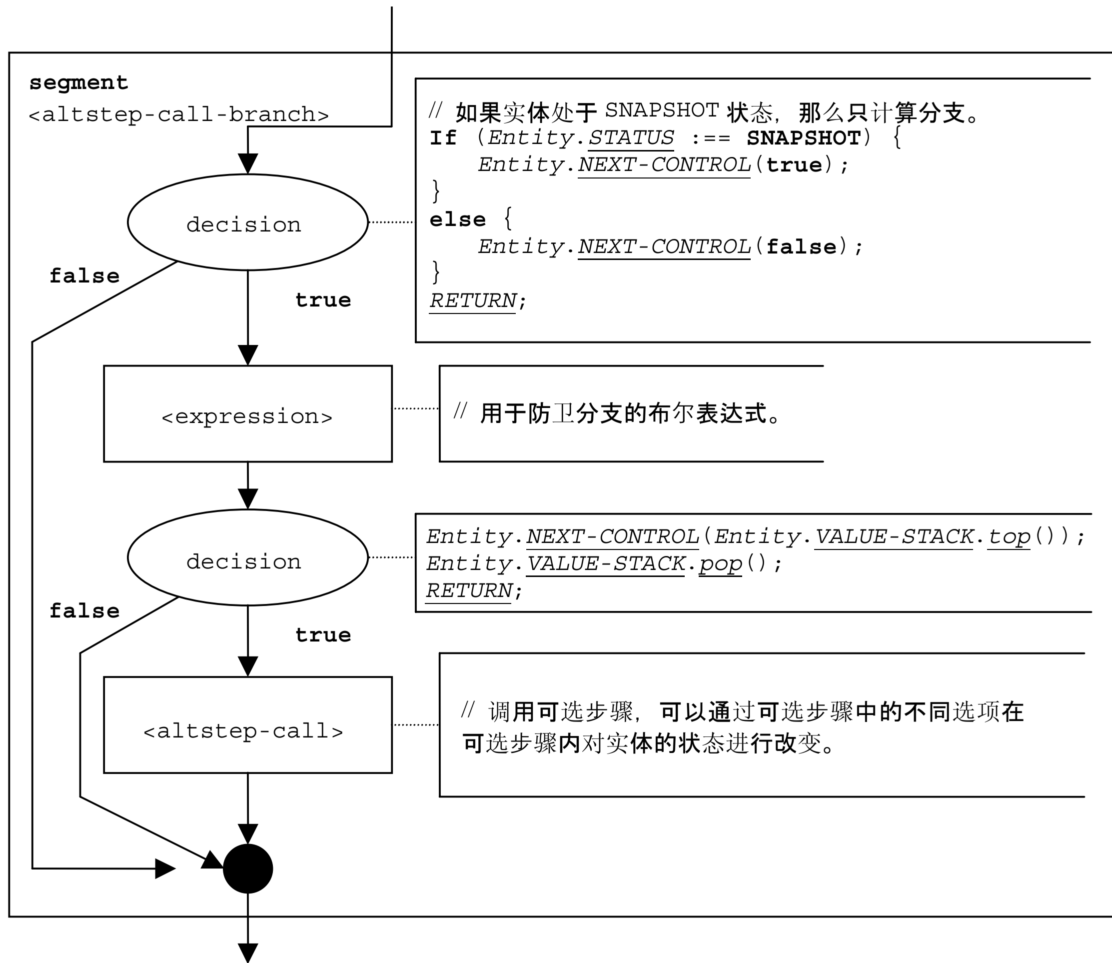


图 41/Z.143－流程图片段<altstep-call-branch>

9.3.4 流程图片段<else-branch>

图 42 中的流程图片段<else-branch>对 **alt** 语句内 **else** 分支的执行过程进行描述。

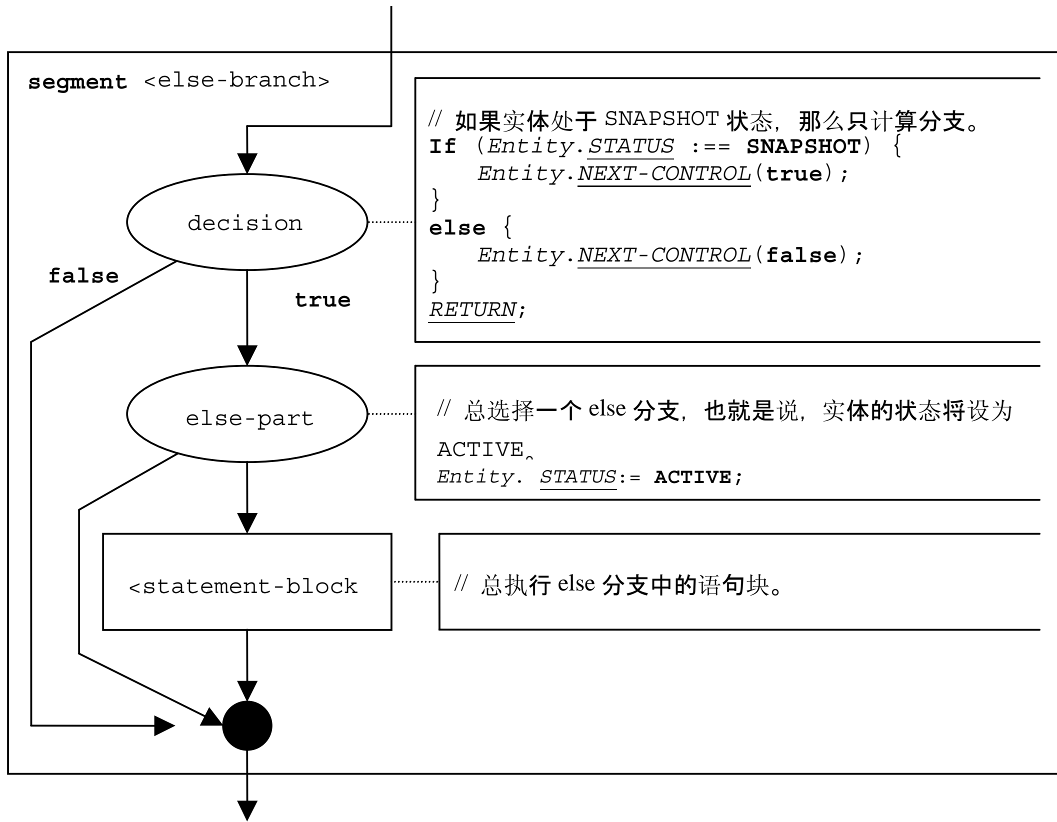


图 42/Z.143－流程图片段<else-branch>

9.3.5 流程图片段<default-evocation>

图 43 中的流程图片段`<default-evocation>`对 **alt** 语句结尾处缺省行为的唤起进行描述。

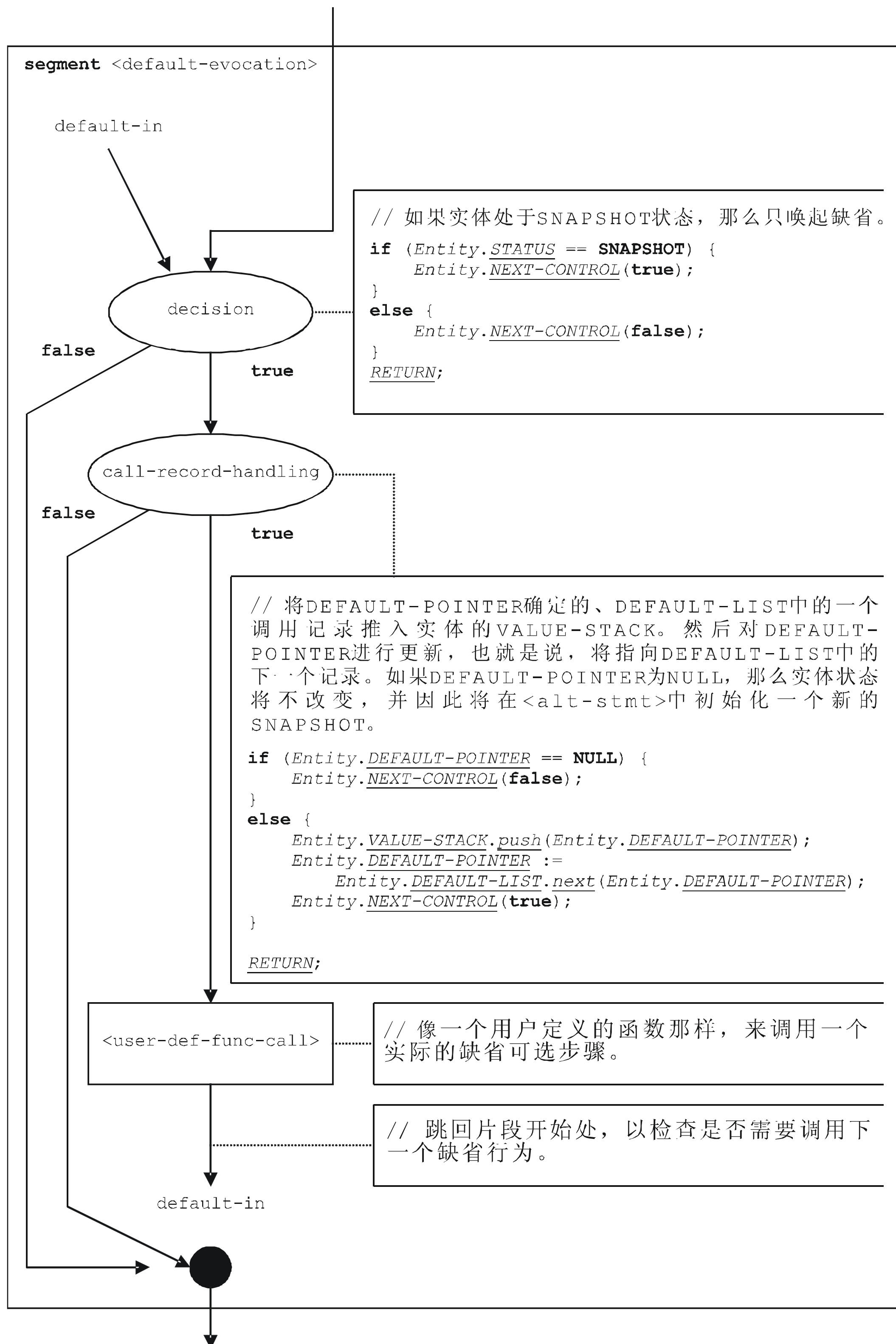


图 43/Z.143－流程图片段<default-evocation>

9.4 可选步骤调用

如图 44 所示，像一个函数调用那样，来处理一个可选步骤的调用。

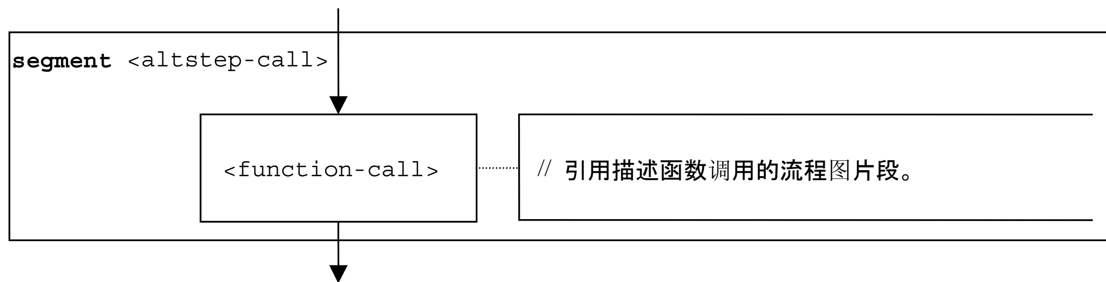


图 44/Z.143－流程图片段<altstep-call>

9.5 Assignment 语句

`assignment` 语句的语法结构是：

```
<varId> := <expression>
```

表达式`<expression>`的值赋给变量`<varId>`。图 45 中的流程图片段`<assignment-stmt>`对 assignment 语句的执行过程进行定义：

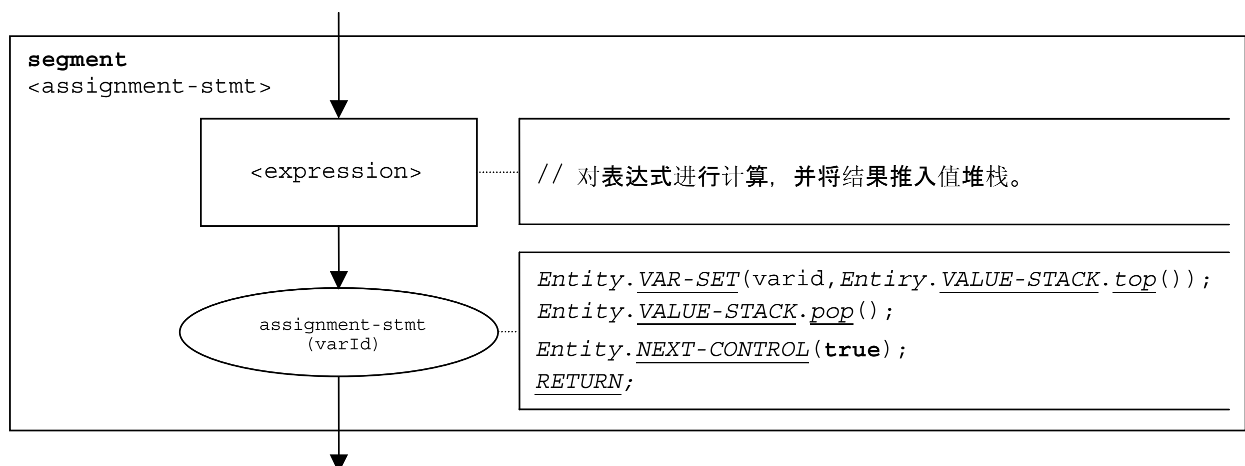


图 45/Z.143－流程图片段<assignment-stmt>

9.6 Call 操作

Call 操作的语法结构是：

```
<portId>.call (<callSpec> [<blocking-info>]) [to <component-expression>] [<call-reception-part>]
```

可选的`<blocking-info>`由关键字 **`nowait`** 或超时异常的持续时间组成。**`to`** 子句中可选的`<component-expression>`指的是接收方实体。它可以以变量值或函数返回值的形式来提供。可选的<call-reception-part>表示在阻塞 **`call`** 操作情况下的可选接收。

操作语义区分阻塞 **`call`** 操作和非阻塞 **`call`** 操作。如果在 **`call`** 操作中使用关键字 **`nowait`**，那么 **`call`** 操作是非阻塞的，或者如果所调用的过程是非阻塞的，那么通过关键字 **`noblock`** 进行定义。阻塞 **`call`** 操作有一个`<call-reception-part>`。

图 46 中的流程图片段<call-op>对 **call** 操作的执行过程进行定义。它反映了阻塞调用与非阻塞调用之间的区别。

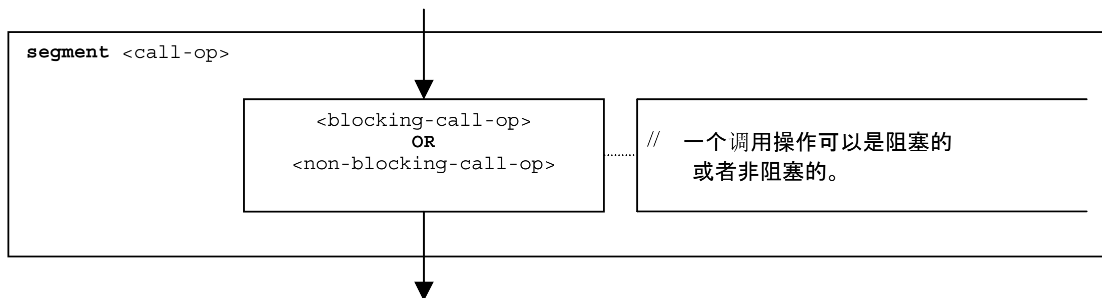


图 46/Z.143－流程图片段<call-op>

对阻塞 call 操作和非阻塞 call 操作，可以以表达式的形式对接收方实体进行描述。其可能性如图 47 和图 48 所示。

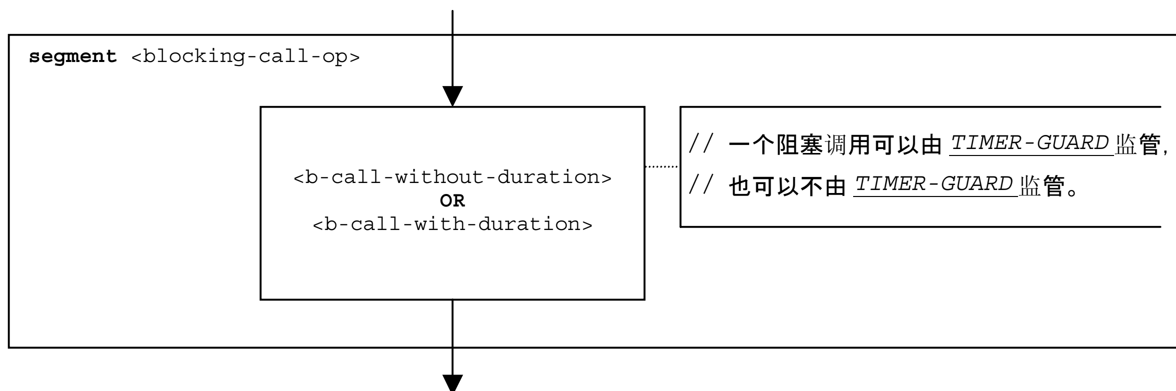


图 47/Z.143－流程图片段<blocking-call-op>

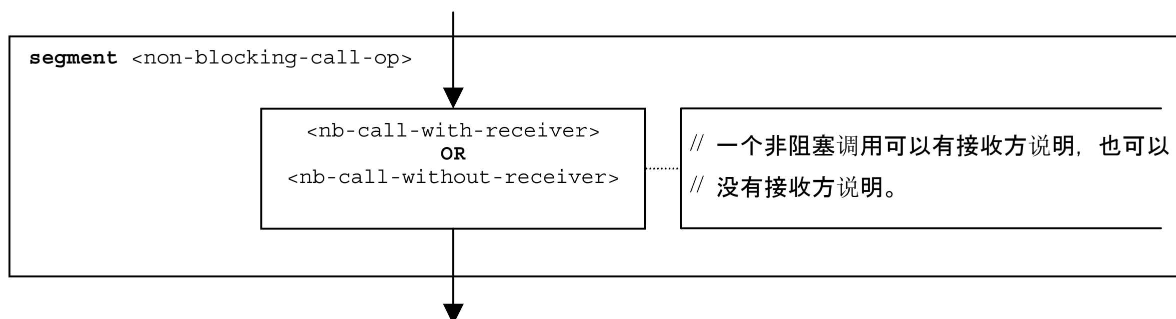


图 48/Z.143－流程图片段<non-blocking-call-op>

9.6.1 流程图片段<nb-call-with-receiver>

图 49 中的流程图片段<nb-call-with-receiver>对非阻塞 **call** 操作的执行过程进行定义，其中的接收方以表达式的形式进行描述。

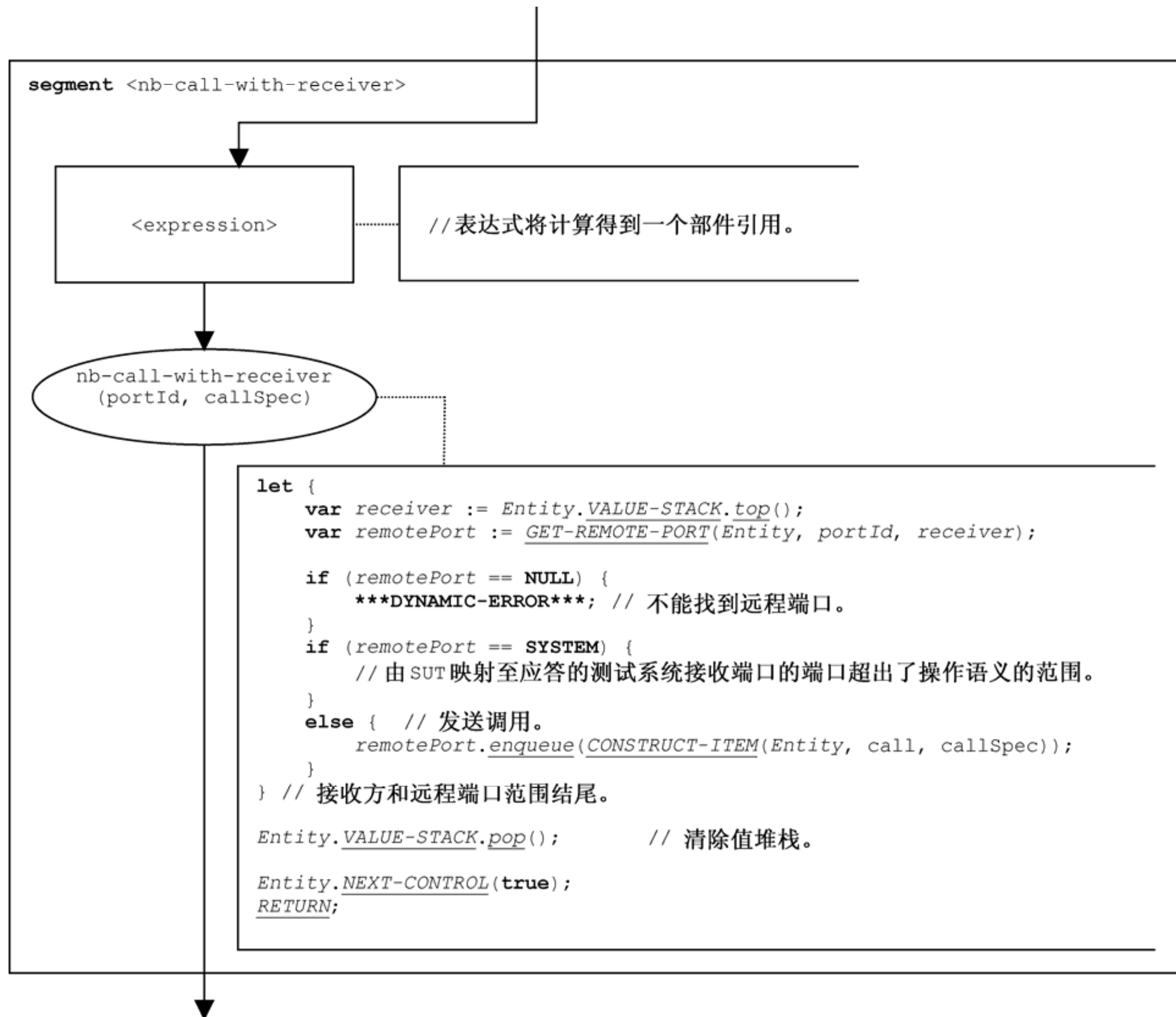


图 49/Z.143－流程图片段<nb-call-with-receiver>

9.6.2 流程图片段<nb-call-without-receiver>

图 50 中的流程图片段<nb-call-without-receiver>对不带 **to** 子句的非阻塞 **call** 操作的执行过程进行定义：

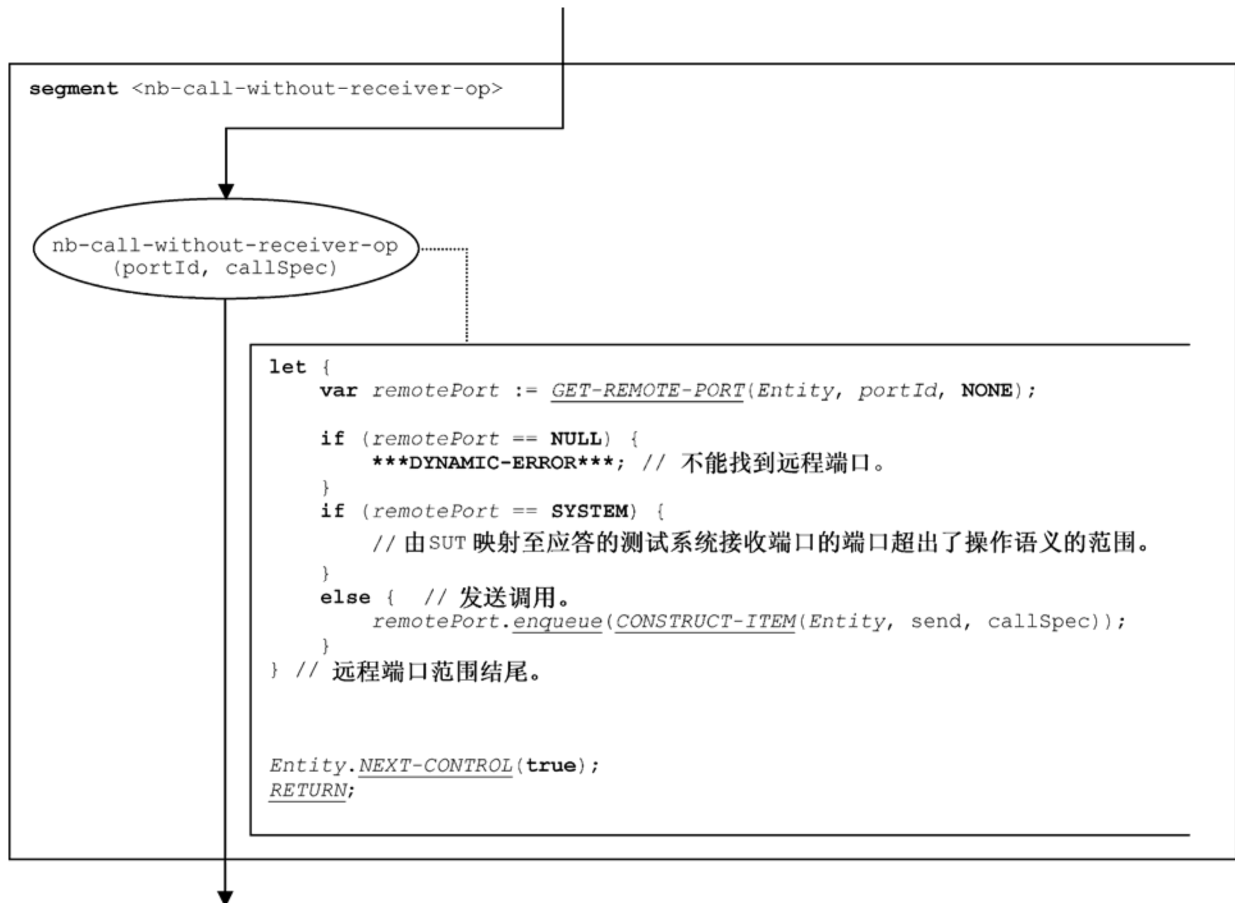


图 50/Z.143－流程图片段<nb-call-without-receiver>

9.6.3 流程图片段<b-call-without-duration>

阻塞调用操作通过一个后跟调用主体的非阻塞调用操作来建模，它处理应答和异常。图 51 中的流程图片段<b-call-without-duration>对阻塞调用操作（它没有一个给定的持续时间作为时间防卫）的执行过程进行定义：

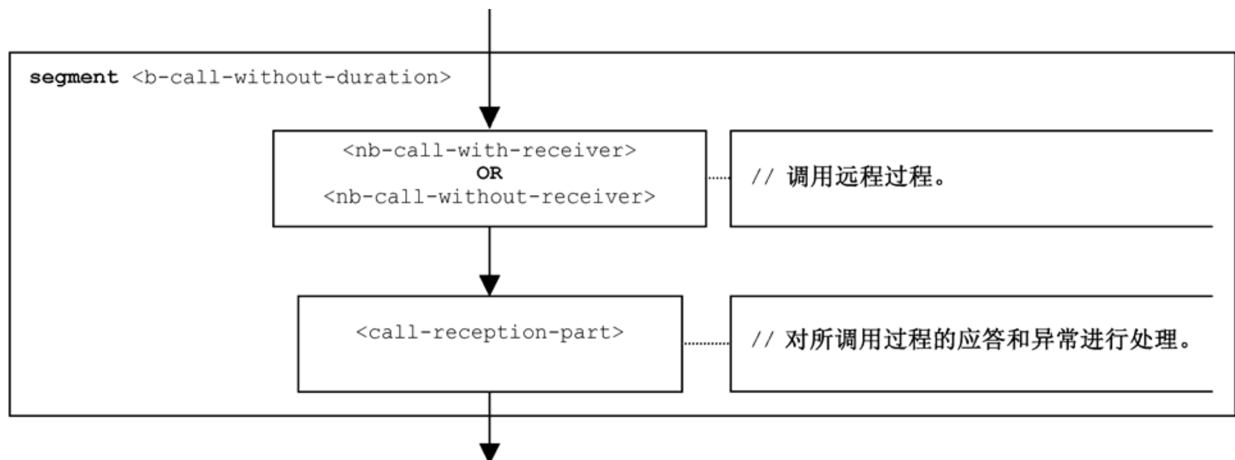


图 51/Z.143－流程图片段<b-call-without-duration>

9.6.4 流程图片段<b-call-with-duration>

图 52 中的流程图片段<b-call-with-duration>对阻塞调用操作（它有一个持续时间作为时间防卫）的执行过程进行定义：

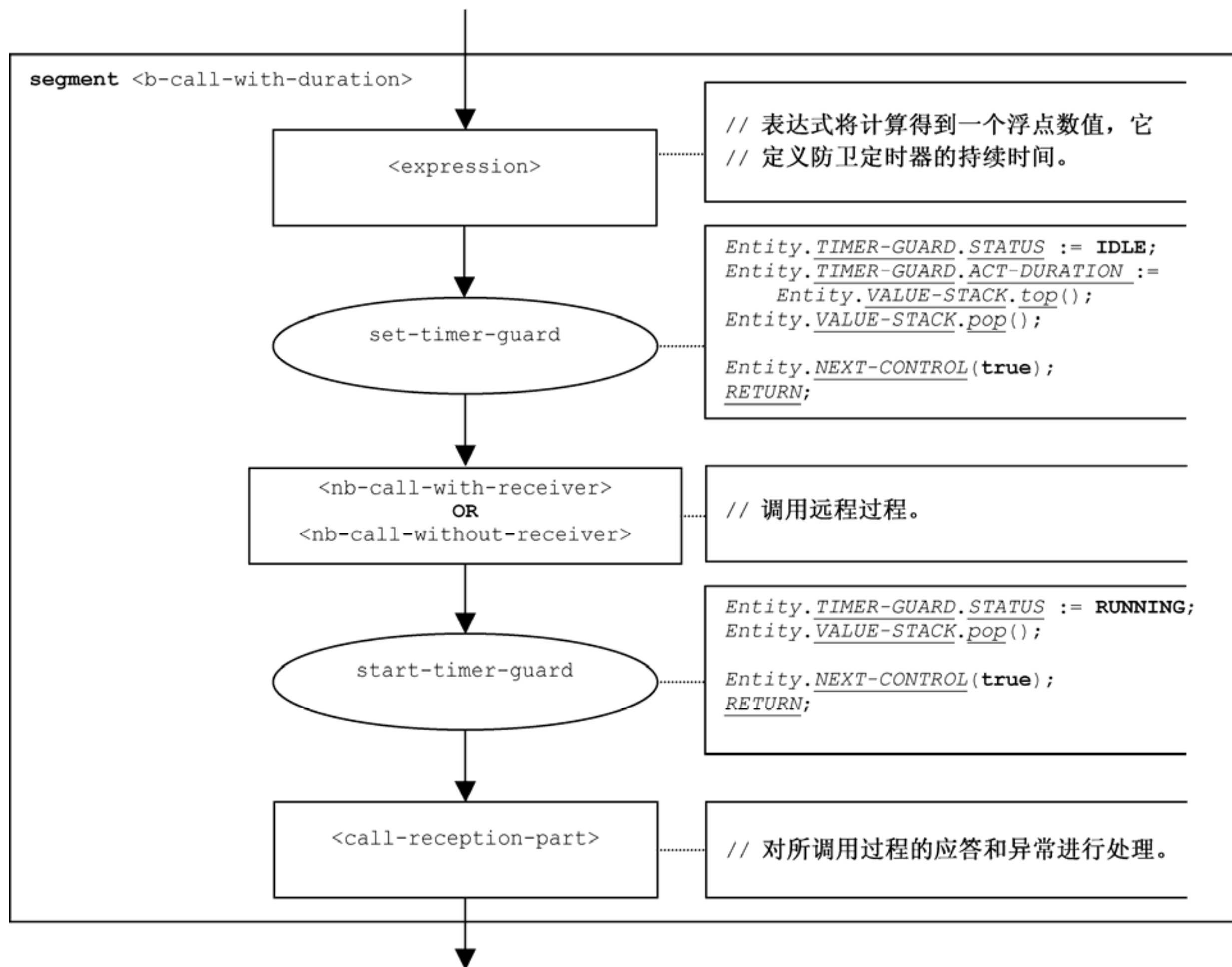


图 52/Z.143－流程图片段<b-call-with-duration>

9.6.5 流程图片段<call-reception-part>

图 53 中的流程图片段<call-reception-part>对阻塞调用操作的应答、异常和超时异常处理过程进行描述：

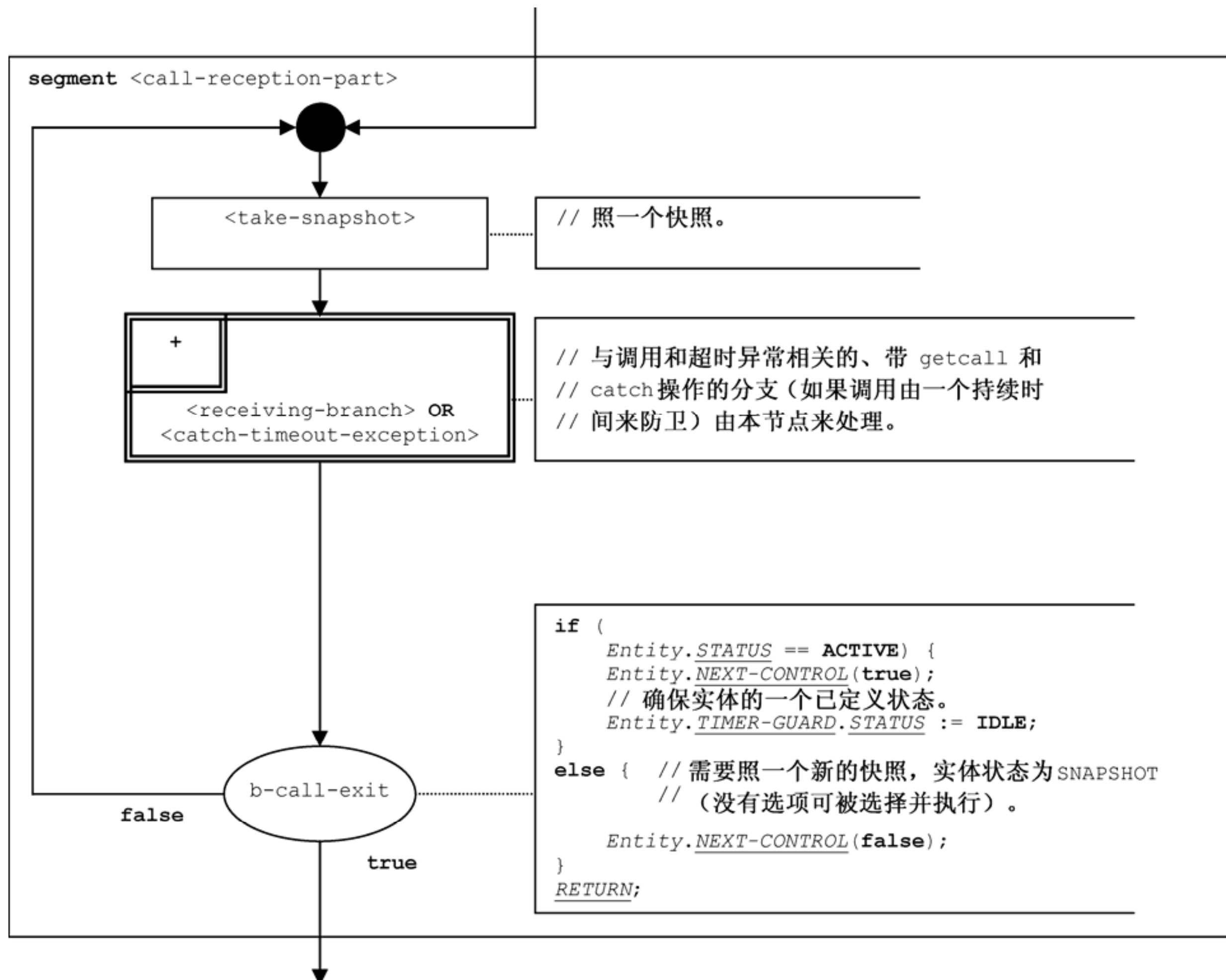


图 53/Z.143－流程图片段<call-reception-part>

9.6.6 流程图片段<catch-timeout-exception>

图 54 中的流程图片段<catch-timeout-exception>对阻塞调用操作（它通过一个持续时间来防卫）的超时异常处理过程进行描述：

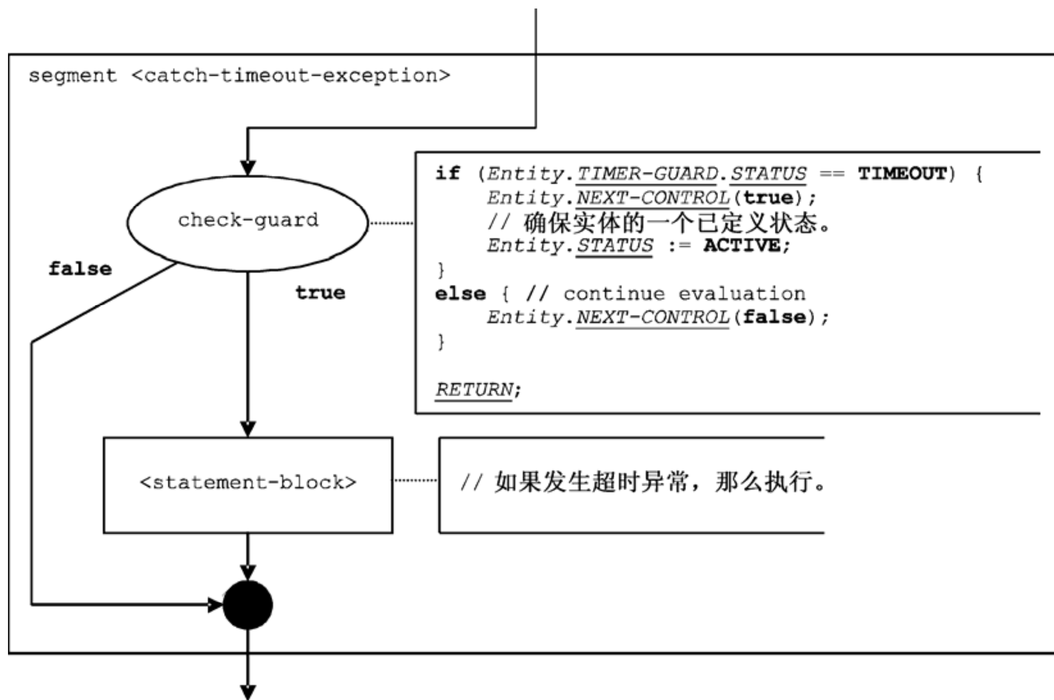


图 54/Z.143－流程图片段<catch-timeout-exception>

9.7 Catch 操作

catch 操作的语法结构是：

```
<portId>.catch (<matchingSpec>) [from <component_expression>] -> [<assignmentPart>]
```

除了关键字 **catch**，本语法结构与 **receive** 操作的语法结构等同。因此，操作语义以与 **receive** 操作相同的方式来处理 **catch** 操作。图 55 中的流程图片段<catch-op>也对 **catch** 操作的执行过程进行了定义。该图指的是与 **receive** 操作相关的流程图片段（见第 9.37 节）。

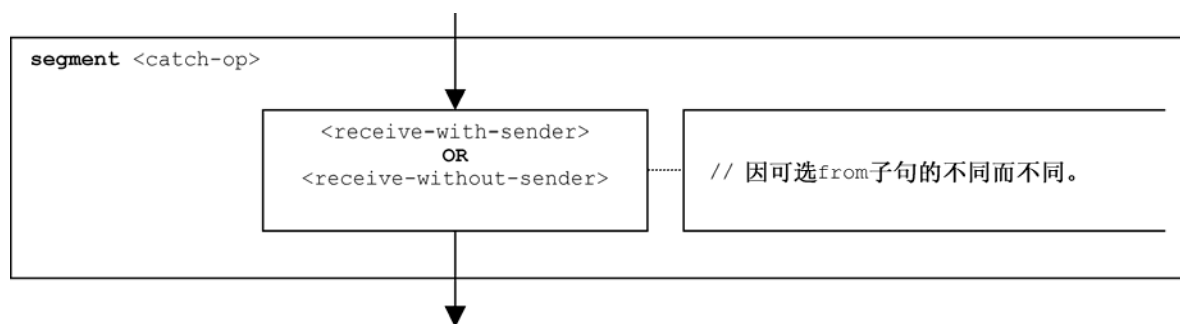


图 55/Z.143－流程图片段<catch-op>

9.8 Check 操作

check 操作的语法结构是：

```
<portId>.check( receive|getcall|catch|getreply (<matchingSpec>)
        [from <component-expression>]) [-> <assignmentPart>]
```

from 子句中可选的<component-expression>指的是发送方实体。它可以以变量值或函数返回值的形式来提供，也就是说，假设它是一个表达式。如果收到的信息匹配于匹配说明<matchingSpec>并匹配（可选的）**from** 子句，那么可选的<assignmentPart>表示接收到信息的赋值。

操作语义以相同的方式来处理 **receive**、**getcall**、**catch** 和 **getreply** 操作，也就是说，通过引用相同的流程图片段<receive-with-sender> 和 <receive-without-sender>来对它们进行描述。**check** 操作也以相同的方式来处理不同的操作。因此，图 56 中的流程图片段<check-op>对 **check** 操作的执行过程进行定义，也只引用两个流程图片段。流程图片段<receive-with-sender> 和 <receive-without-sender>唯一的不同之处是在匹配后不删除接收到的条目。

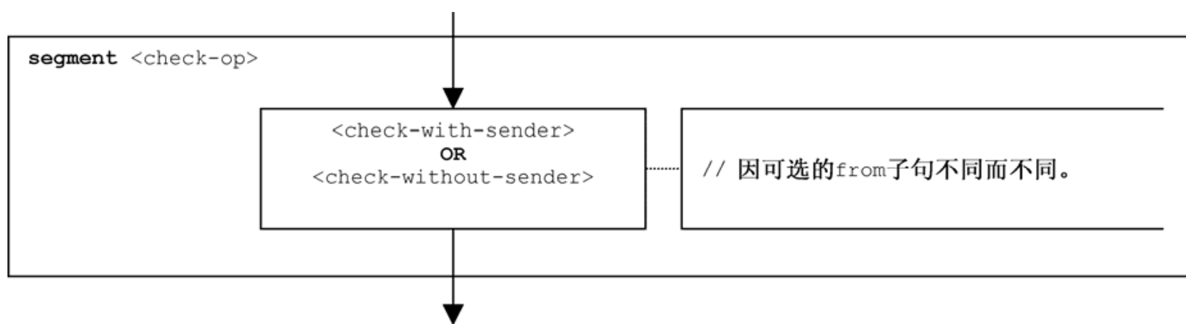


图 56/Z.143－流程图片段<check-op>

9.8.1 流程图片段<check-with-sender>

图 57 中的流程图片段`<check-with-sender>`对 **check** 操作的执行过程进行定义，其中的发送方以表达式的形式进行描述。

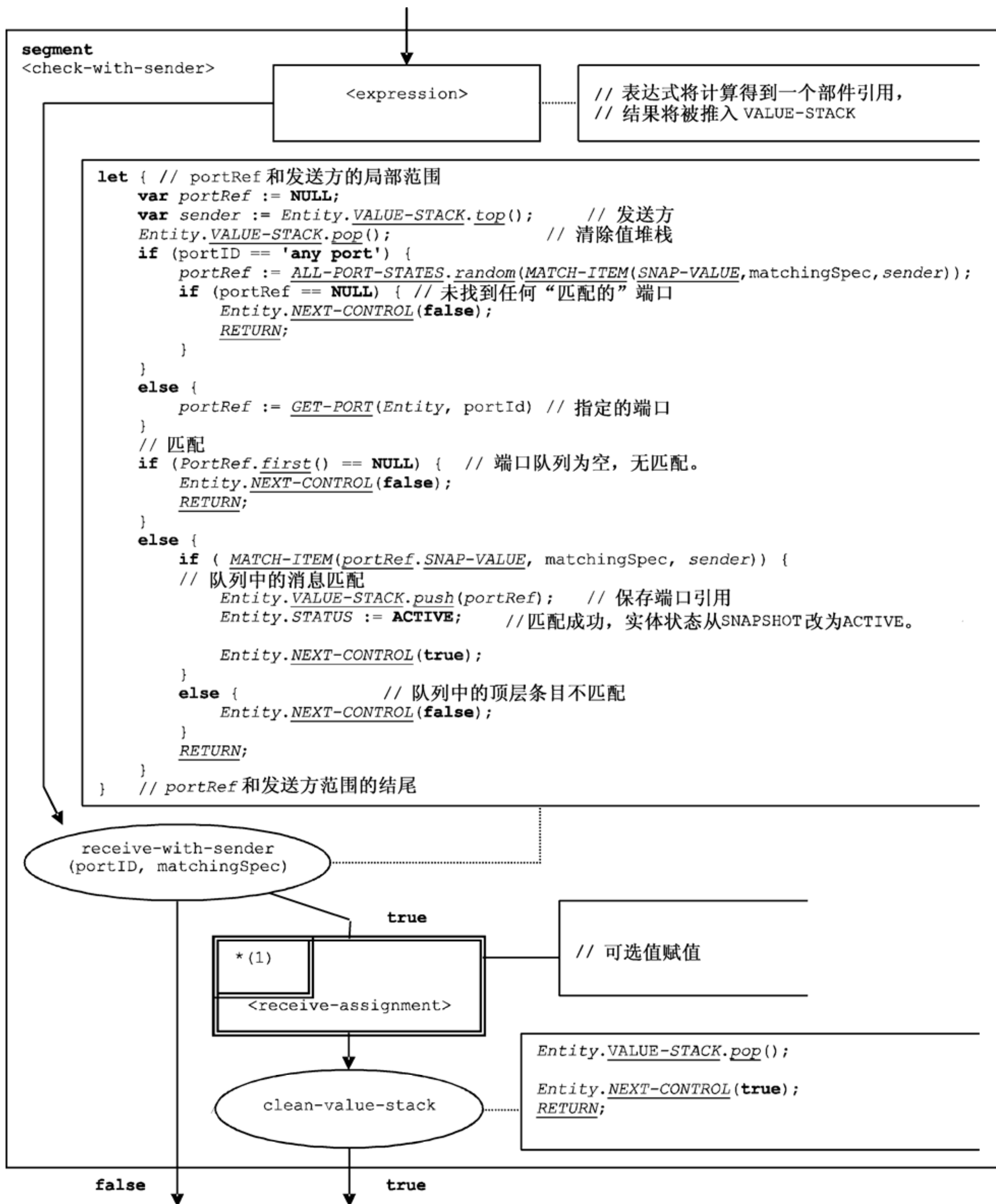


图 57/Z.143－流程图片段<check-with-sender>

9.8.2 流程图片段<check-without-sender>

图 58 中的流程图片段<check-without-sender>对不带 **from** 子句的 **check** 操作的执行过程进行定义：

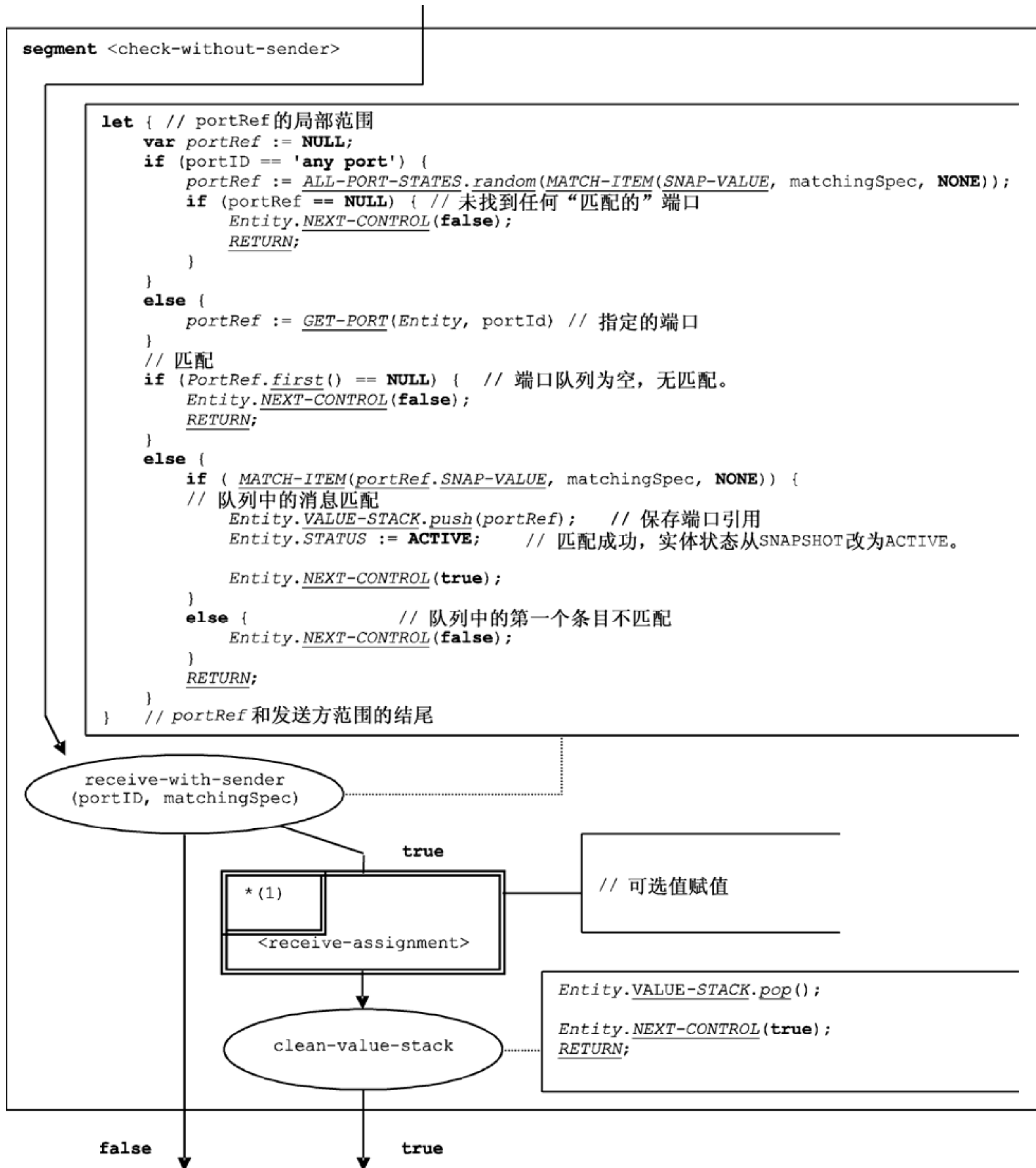


图 58/Z.143－流程图片段<check-without-sender>

9.9 Clear端口操作

clear 端口操作的语法结构是：

```
<portId>.clear
```

图 59 中的流程图片段<clear-port-op>对 **clear** 端口操作的执行过程进行定义。

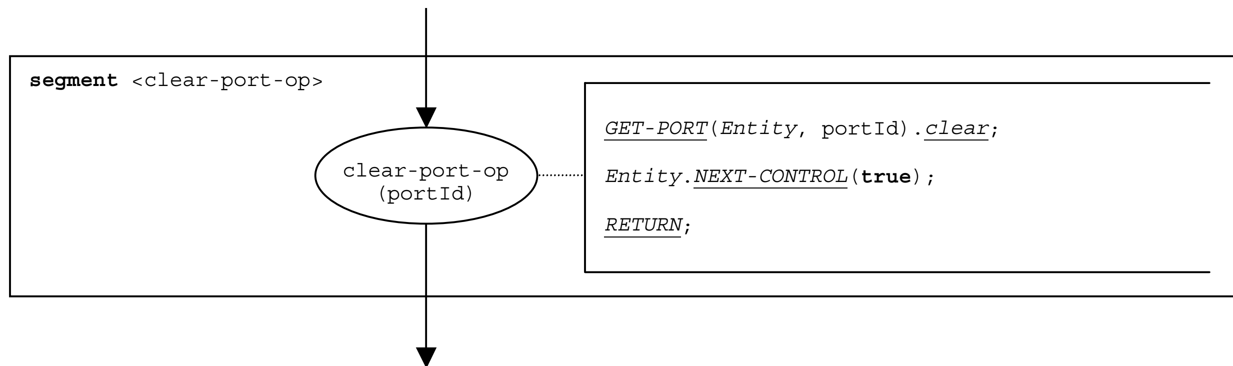


图 59/Z.143－流程图片段<clear-port-op>

9.10 Connect 操作

connect 操作的语法结构是：

```
connect(<component-expression1>:<portId1>, <component-expression2>:<portId2>)
```

认为标识符<portId1> 和 <portId2>是对应测试部件的端口标识符。端口所属的部件通过部件引用<component-expression$_1$> 和 <component-expression$_2$>的方式来引用。引用可以保存在变量中，或通过函数来返回，也就是说，它们是计算为部件引用的表达式。值堆栈用于保存部件引用。

图 60 中的流程图片段<connect-op>对 **connect** 操作的执行过程进行定义。在流程图描述中，待计算的第一个表达式指的是<component-expression$_1$>，第二个表达式指的是<component-expression$_2$>，也就是说，当执行 connect-op 节点时，<component-expression$_2$>位于值堆栈的顶层。

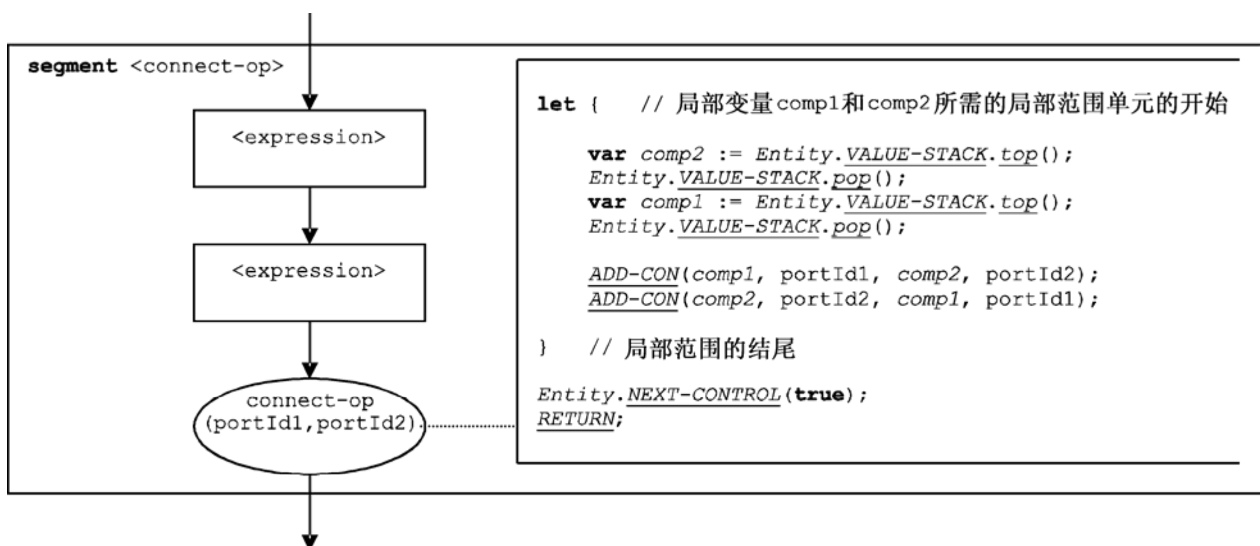


图 60/Z.143－流程图片段<connect-op>

9.11 Constant 定义

constant 定义的语法结构是：

```
const <constType> <constId> := <constType-expression>
```

认为一个常量的值是一个表达式，它计算得到一个常量类型的值。

注 — 在应用本语义之前，用预处理步骤中的值来替换全局常量（见第9.2节）。像带初始化值的变量声明那样来对局部常量进行处理。常量的正确用法，即常量绝不得出现在赋值语句的左侧，将在TTCN-3模块的静态语义分析期间进行检查。

图 61 中的流程图片段<constant-definition>对常量声明的执行过程进行定义，其中的常量值以表达式的形式来提供。

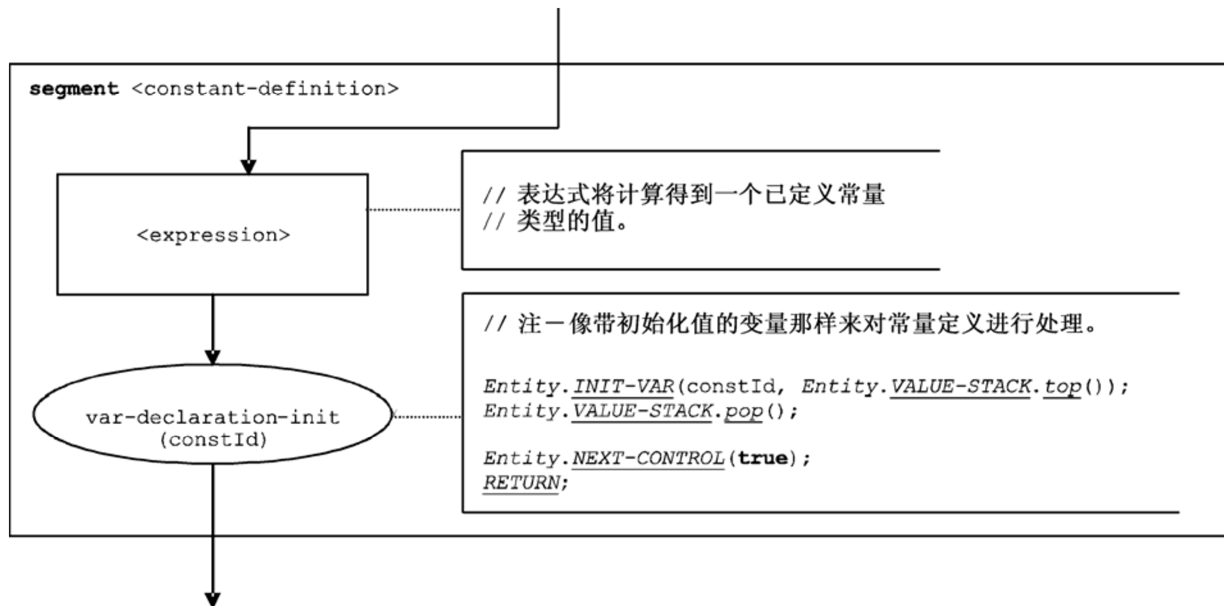


图 61/Z.143－流程图片段<constant-definition>

9.12 Create 操作

create 操作的语法结构是：

```
<componentTypeId>.create
```

图 62 中的流程图片段<create-op>对 **create** 操作的执行过程进行定义：

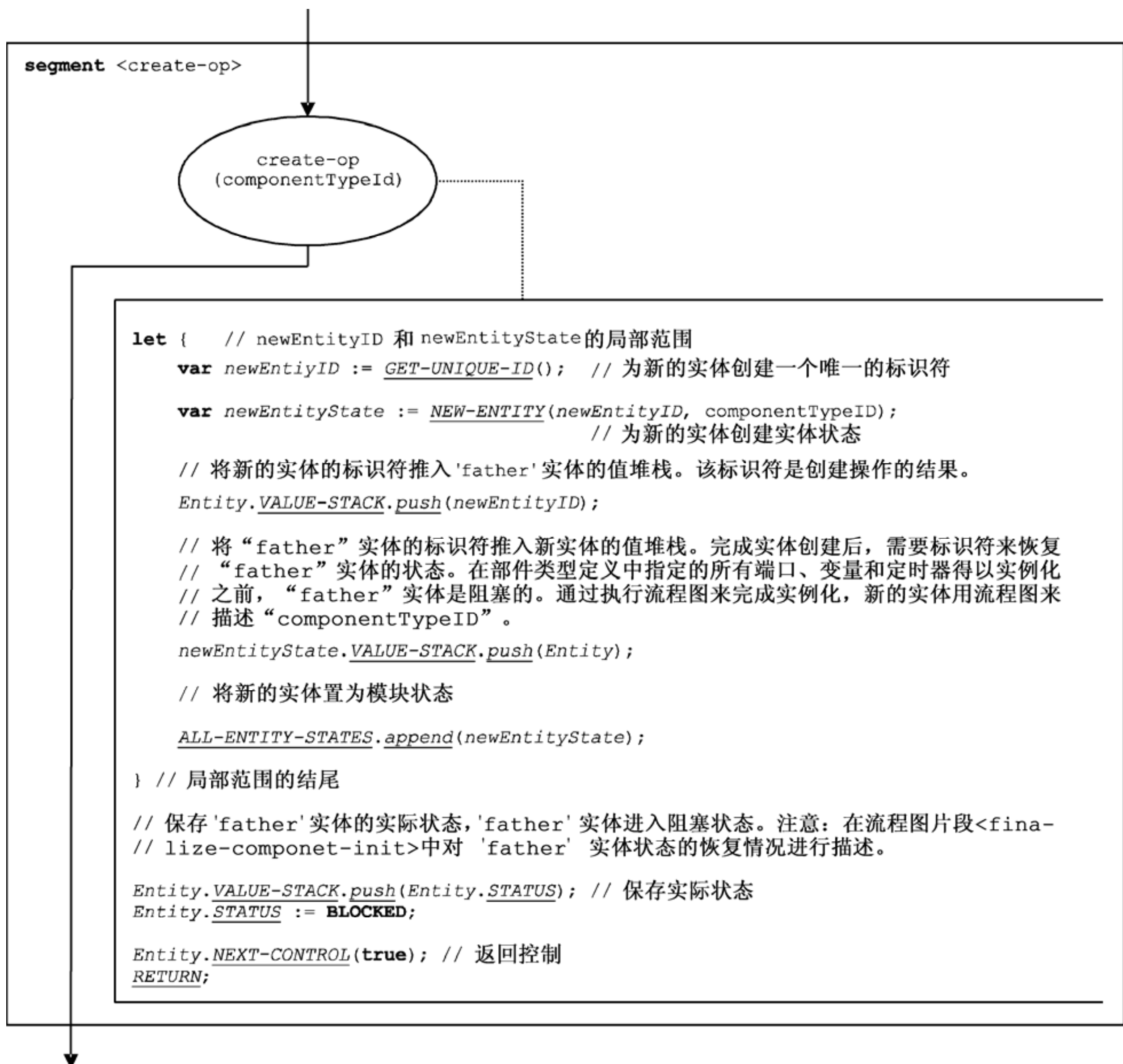


图 62/Z.143－流程图片段<create-op>

9.13 Deactivate 语句

deactivate 语句的语法结构是：

```
deactivate [(<default-expression>)]
```

deactivate 语句用于说明解除执行 **deactivate** 语句的、一个或所有活动缺省值的活动特性。如果解除一个缺省值的活动特性，那么可选的<default-expression>将计算得到一个缺省的引用，它确定待解除活动特性的缺省值。调用不带**<default-expression>**的 **deactivate** 语句将解除所有缺省值的活动特性。

图 63-a 中的流程图片段<deactivate-stmt>对 **deactivate** 语句的执行过程进行定义：

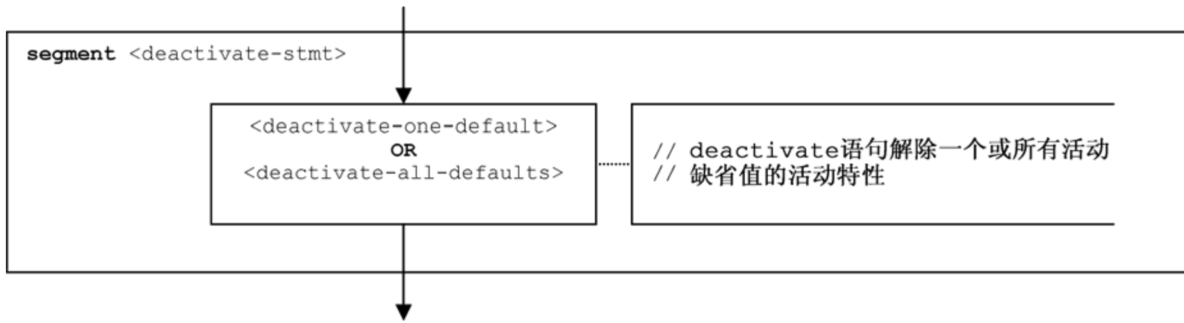


图 63-a/Z.143－流程图片段<deactivate-stmt>

9.13.1 流程图片段<deactivate-one-default>

图 63-b 中的流程图片段<deactivate-one-default>用于说明解除一个活动缺省值的活动特性。表达式<default-expression>的值将计算得到一个缺省的引用。可以以一个变量值或值返回函数的形式来提供表达式。**deactivate** 语句从执行 **deactivate** 语句的实体 *DEFAULT-LIST* 中清除指定的缺省值。

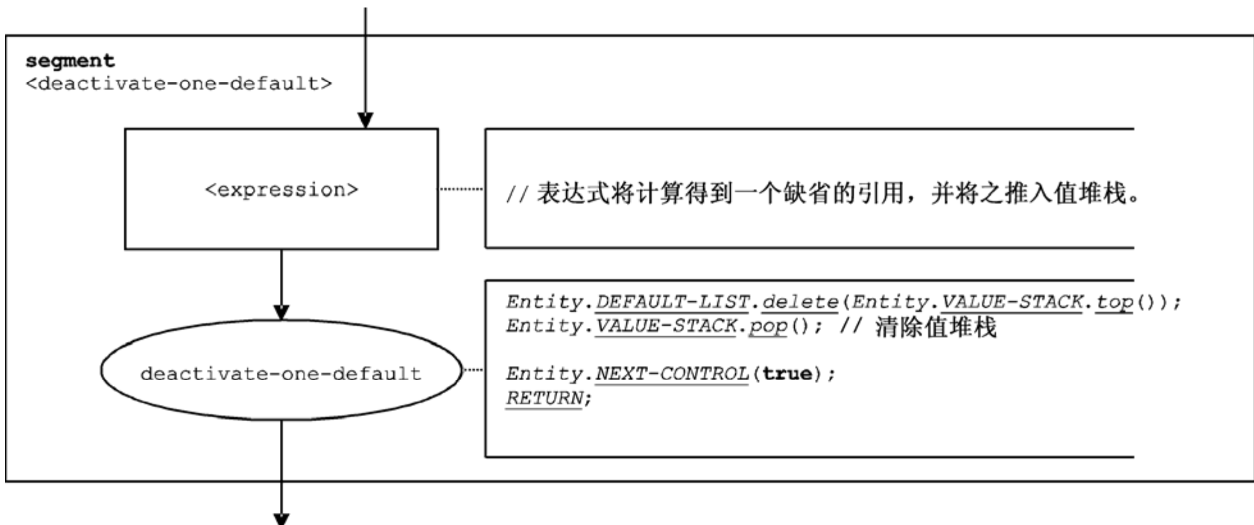


图 63-b/Z.143－流程图片段<deactivate-one-default>

9.13.2 流程图片段<deactivate-all-defaults>

图 63-c 中的流程图片段<deactivate-all-defaults>用于说明解除所有活动缺省值的活动特性。**deactivate** 语句清除执行 **deactivate** 语句的、实体的 *DEFAULT-LIST*。

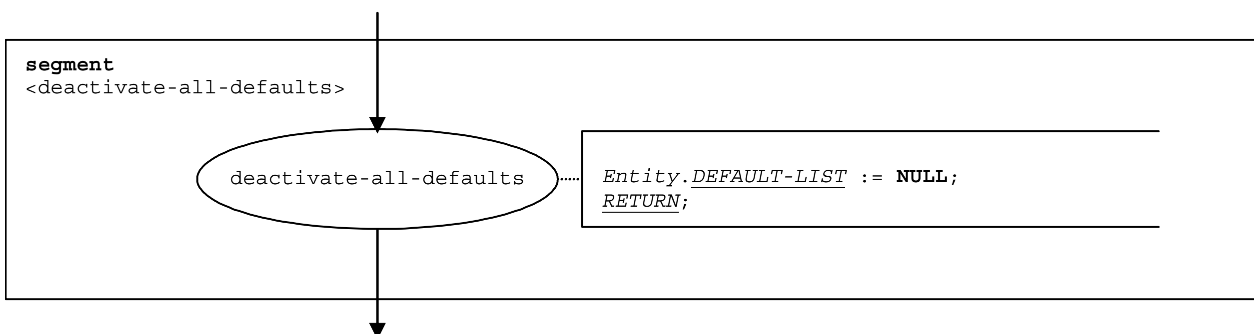


图 63-c/Z.143－流程图片段<deactivate-all-defaults>

9.14 Disconnect 操作

disconnect 操作的语法结构是：

```
disconnect(<component-expression1>:<portId1>,
    <component-expression2>:<portId2>)
```

认为标识符 <portId1> 和 <portId2> 是对应的测试部件的端口标识符。通过部件引用 <component-expression$_1$> 和 <component-expression$_2$>的方式来引用端口所属的部件。引用可以保存在变量中或通过函数返回，也就是说，它们是表达式，将计算得到部件引用。值堆栈用于保存部件引用。

图 64 中的流程图片段<disconnect-op>对 **disconnect** 操作的执行过程进行定义。在流程图片段中，待计算的第一个表达式指的是 <component-expression$_1$>，第二个表达式指的是 <component-expression$_2$>，也就是说，当执行 disconnect-op 节点时，<component-expression$_2$> 位于值堆栈的顶层。

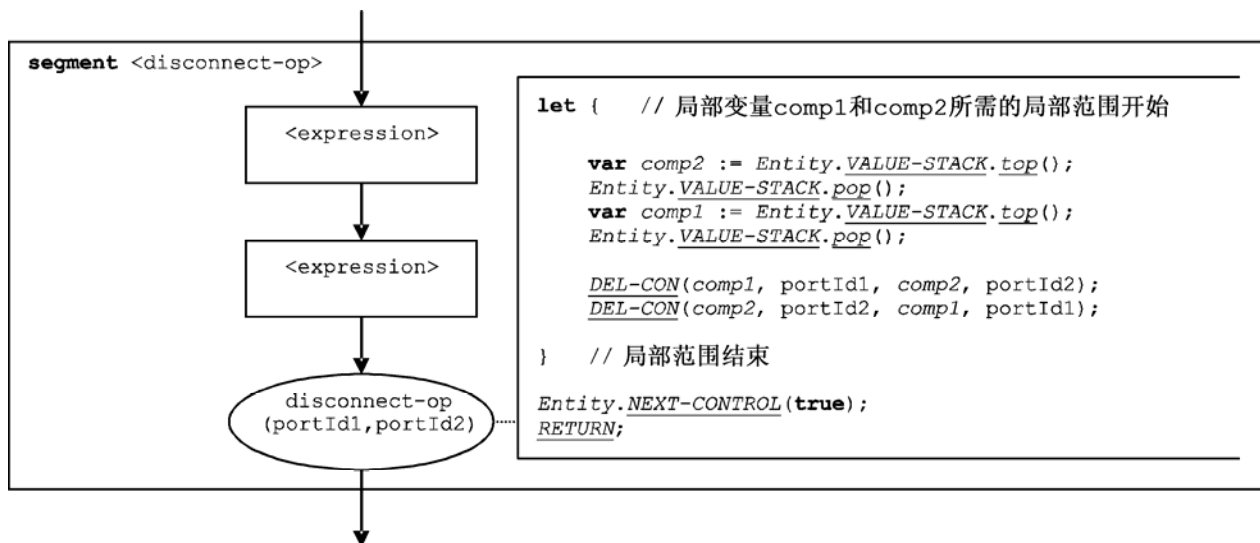


图 64/Z.143－流程图片段<disconnect-op>

9.15 Do-while 语句

do-while 语句的语法结构是：

```
do <statement-block>
while (<boolean-expression>)
```

图 65 中的流程图片段<do-while-stmt>对 **do-while** 语句的执行过程进行定义：

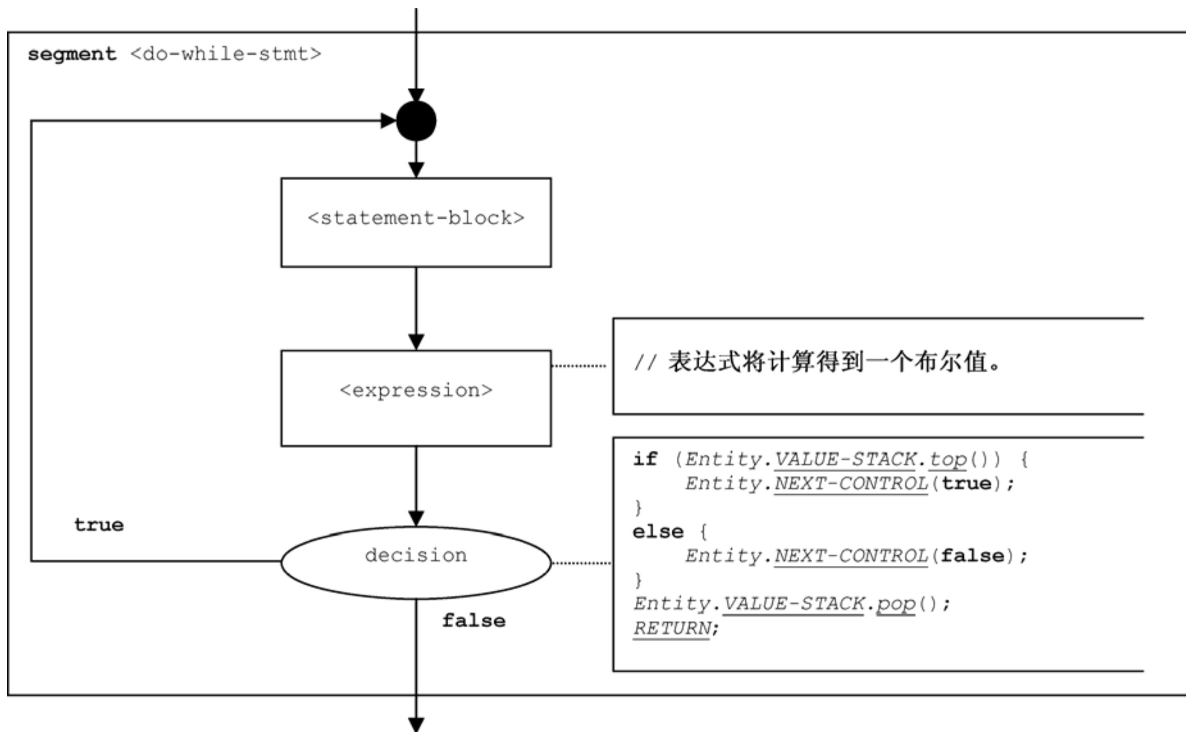


图 65/Z.143－流程图片段<do-while-stmt>

9.16 Done部件操作

done 部件操作的语法结构是：

```
<component-expression>.done
```

done 部件操作检查部件是正在运行还是已经停止。依据所检查的部件是正在运行还是已经停止，**done** 操作决定控制流如何继续进行。使用一个部件引用来确定要检查的部件。引用可以保存在变量中或通过函数返回，也就是说，它是一个表达式。为简化起见，认为关键字'**all component**'和'**any component**'是特殊的表达式。

图 66 中的流程图片段<done-op>对 **done** 部件操作的执行过程进行定义：

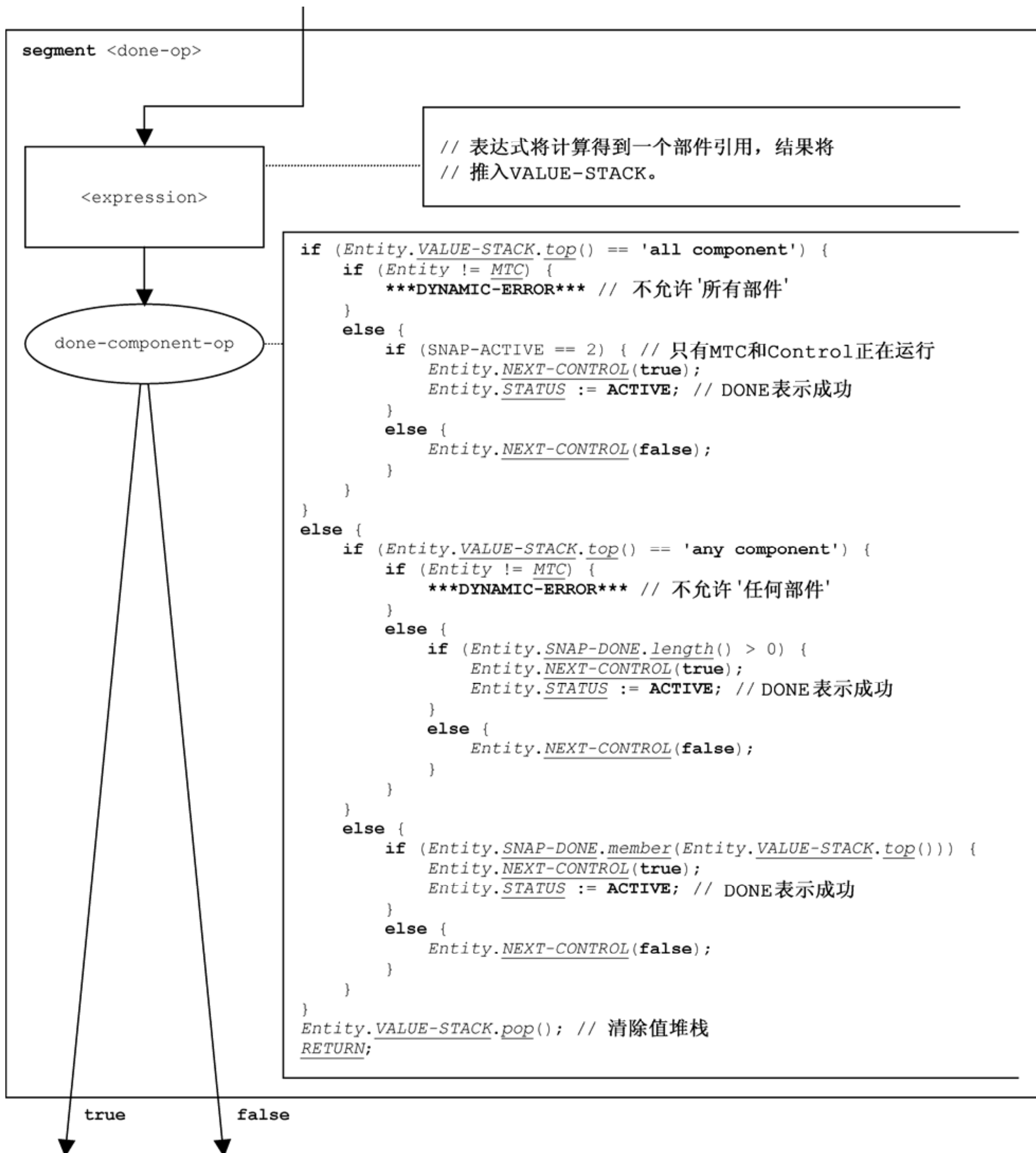


图 66/Z.143－流程图片段<done-component-op>

9.17 Execute 语句

execute 语句的语法结构是：

```
execute(<testCaseId>([<act-par1>, ... , <act-parn>)]) [, <float-expression>])
```

execute 语句描述带（可选）实参<act-par$_1$>，...，<act-par$_n$>的测试用例<testCaseId>的执行。可选地，执行语句可以通过一个以表达式形式提供的持续时间来防卫，表达式计算得到一个 **float**。在指定的持续时间内，如果测试用例不返回一个判定，那么发生超时异常，测试用例停止并返回一个 **error** 判定。

注 — 操作语义通过MTC的停止来构件测试用例停止的模型。实际上，其他机制可能更合适。

如果未发生超时异常，那么创建 MTC，在测试用例终止之前，控制实例（表示 TTCN-3 模块的控制部分）阻塞，对进一步的测试用例执行，将为 MTC 提供控制流。当 MTC 终止时，将控制流还给控制实例。

图 67 中的流程图片段<execute-stmt>对 **execute** 语句的执行过程进行定义：

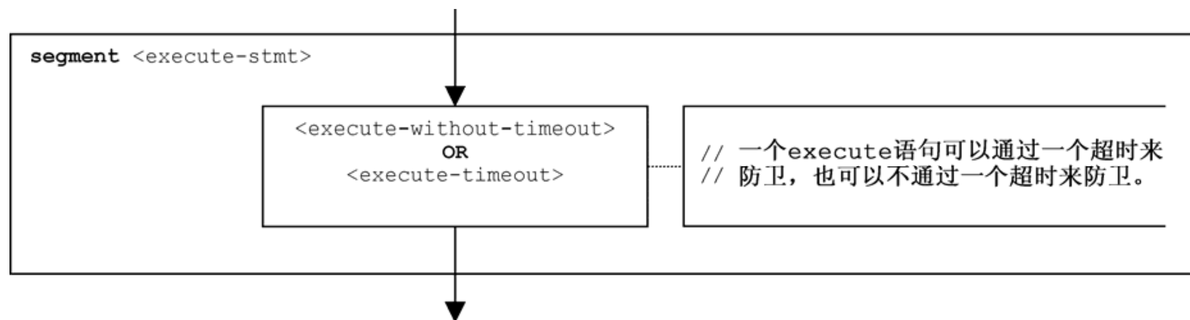


图 67/Z.143－流程图片段<execute-stmt>

9.17.1 流程图片段<execute-without-timeout>

测试用例的执行以创建 **mtc** 开始。然后，以测试用例定义中定义的行为开始 **mtc**。之后，模块控制等待，直至测试用例终止。可以通过使用 **create** 和 **start** 语句来对 MTC 的创建和开始进行描述。

```
var mtcType MyMTC := mtcType.create;
MyMTC.start(TestCaseName(P1...Pn));
```

通过使用 **create** 和 **start** 操作的流程图片段，图 68 中的流程图片段<execute-without-timeout>对不会发生超时异常的 **execute** 语句的执行过程进行定义：

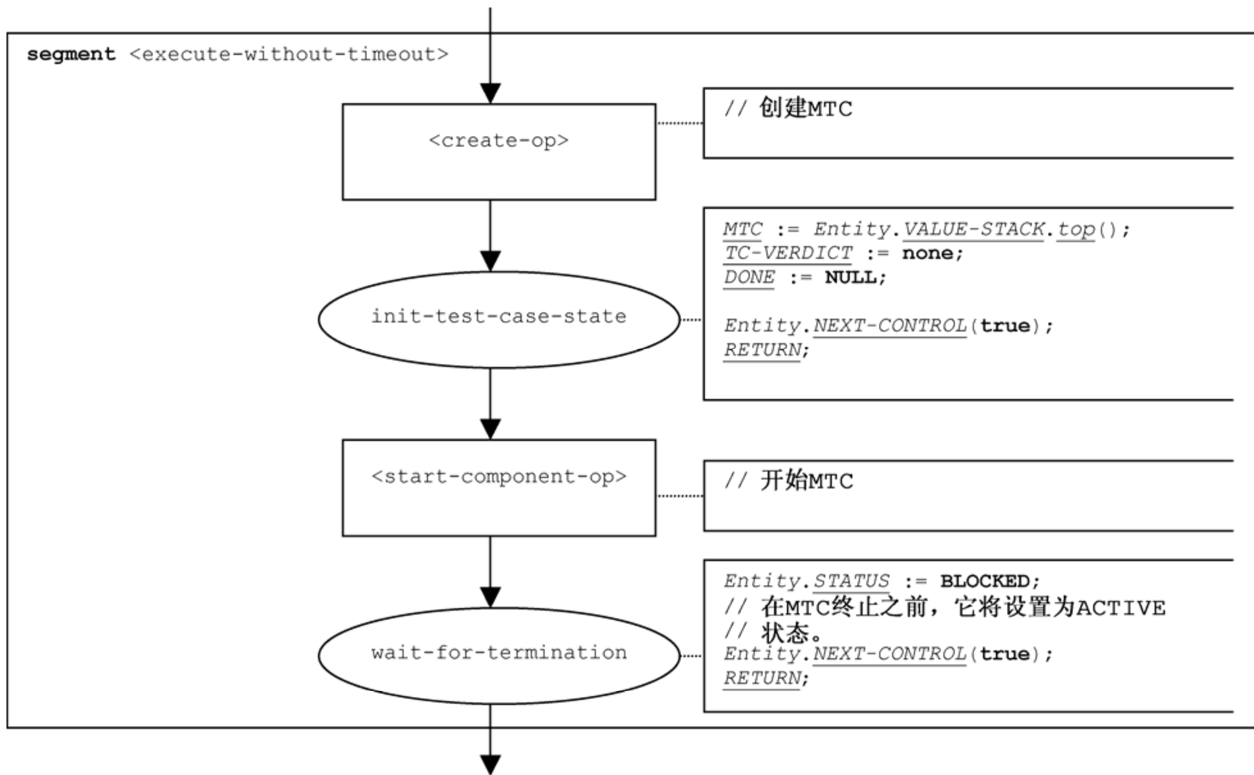


图 68/Z.143－流程图片段<execute-without-timeout>

9.17.2 流程图片段<execute-timeout>

图 69 中的流程图片段<execute-timeout>对通过一个超时值来防卫的 **execute** 语句的执行过程进行定义。流程图片段还通过 **create** 和 **start** 操作来对 MTC 的创建和开始进行建模。另外，*TIMER-GUARD* 用于防卫终止。

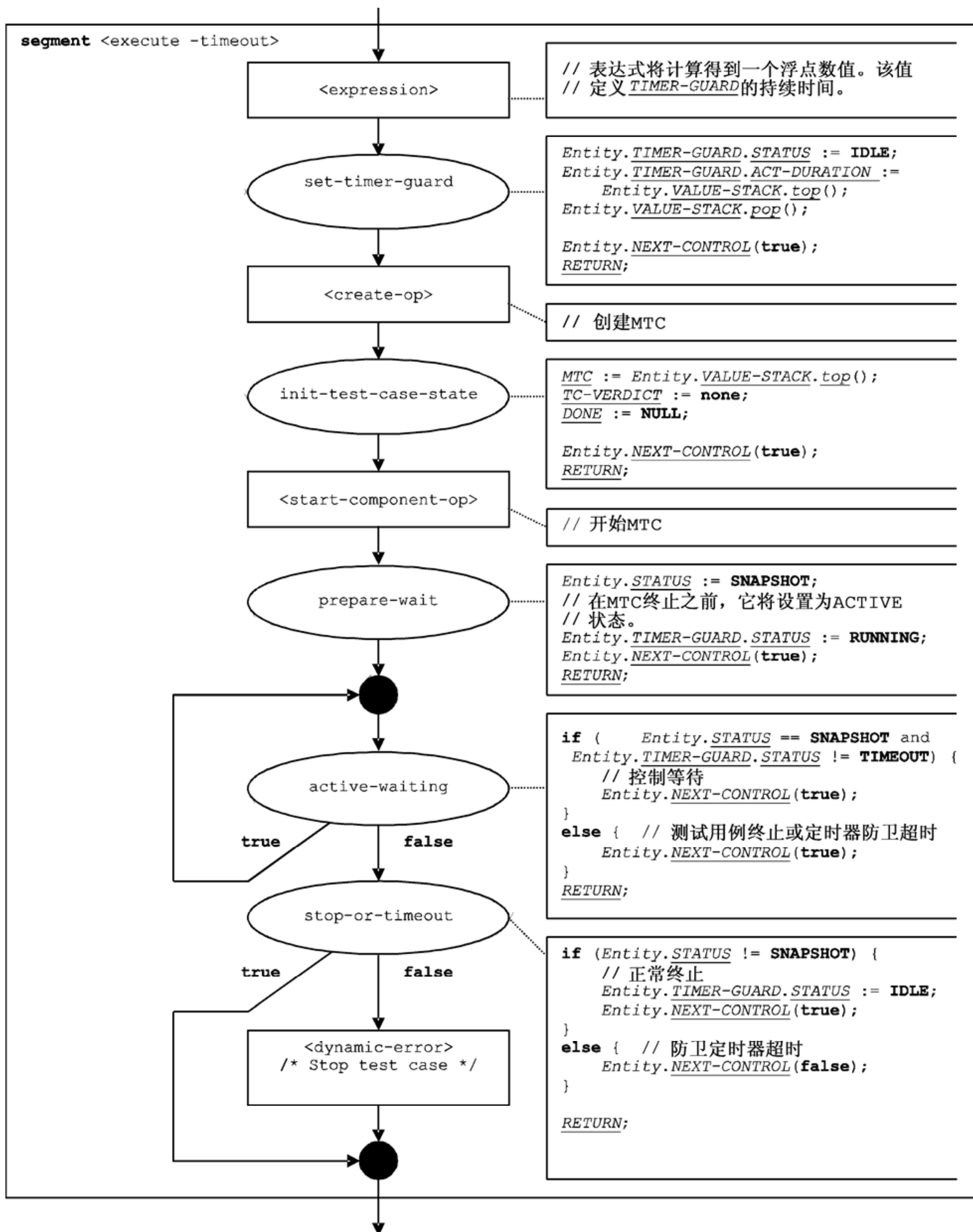


图 69/Z.143－流程图片段<execute-timeout>

9.18 表达式

对表达式的处理，需区别以下四种情况：

a） 表达式是一个文字值（或是一个常量）；

b） 表达式是一个变量；

c） 表达式是一个应用于一个或多个操作数的运算符；

d） 表达式是一个函数或操作调用。

表达式的语法结构是：

```
<lit-val> | <var-val> | <func-op-call> | <operand-appl>
```

其中：

<lit-val>表示一个文字值；

<var-val>表示一个变量值；

<func-op-call>表示一个函数或操作调用；

<operator-appl>表示算术运算符的应用，如＋、－、**not**等。

图 70 中的流程图片段<expression>对表达式的执行过程进行定义：

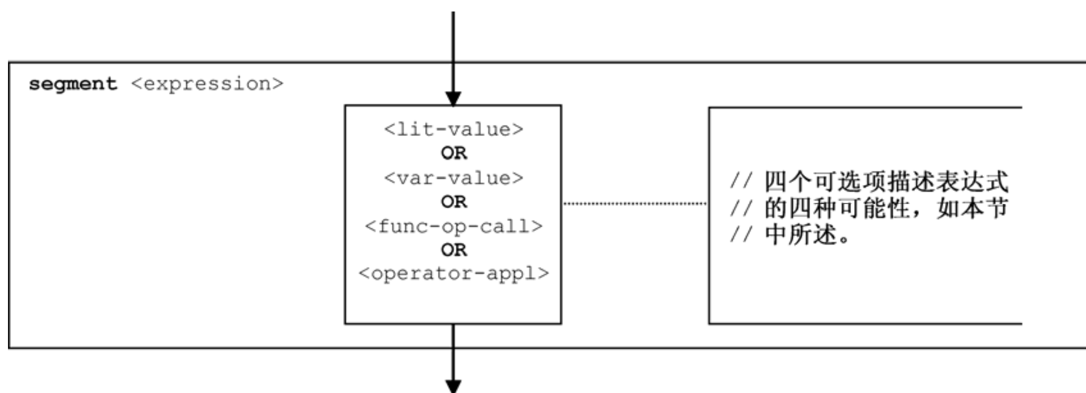


图 70/Z.143－流程图片段<expression>

9.18.1 流程图片段<lit-value>

图 71 中的流程图片段<lit-value>将一个文字值推入一个实体的值堆栈。

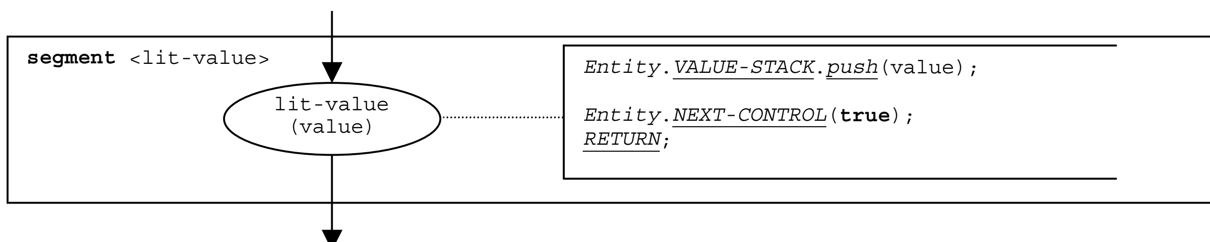


图 71/Z.143－流程图片段<lit-value>

9.18.2 流程图片段<var-value>

图 72 中的流程图片段<var-value>将一个变量值推入一个实体的值堆栈。

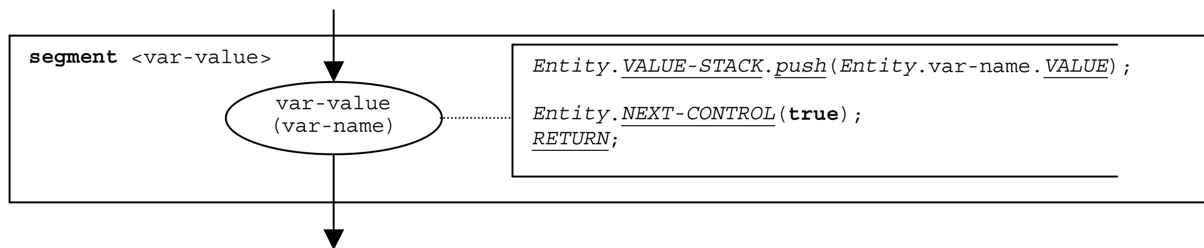


图 72/Z.143－流程图片段<var-value>

9.18.3 流程图片段<func-op-call>

图 73 中的流程图片段<func-op-call>指的是调用函数和操作，它返回一个值，该值将推入一个实体的值堆栈。认为所有这些调用都是表达式。

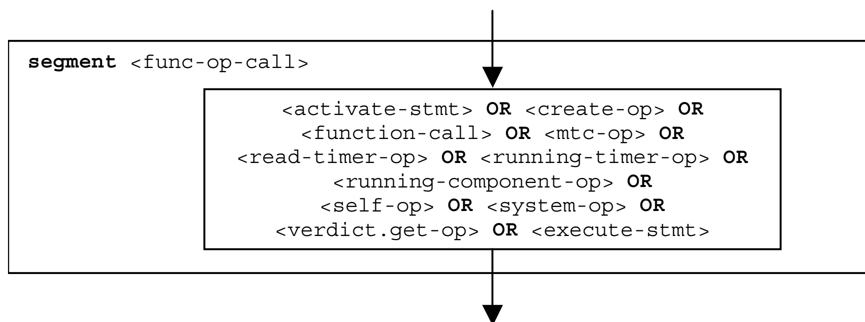


图 73/Z.143－流程图片段<func-op-call>

9.18.4 流程图片段<operator-appl>

图 74 中的流程图直接指的是以下假设，即逆记法用于计算运算符表达式。对运算符的操作数进行计算，并推入计算值堆栈。为应用运算符，从计算值堆栈中弹出操作数，并应用运算符。运算符的应用结果最终将推入计算值堆栈。认为操作符弹出和计算值推入二者都是运算符应用（图 74 中的 Entity.APPLY-OPERATOR(operator) 语句）的一部分，也就是说，不通过操作语义来建模。

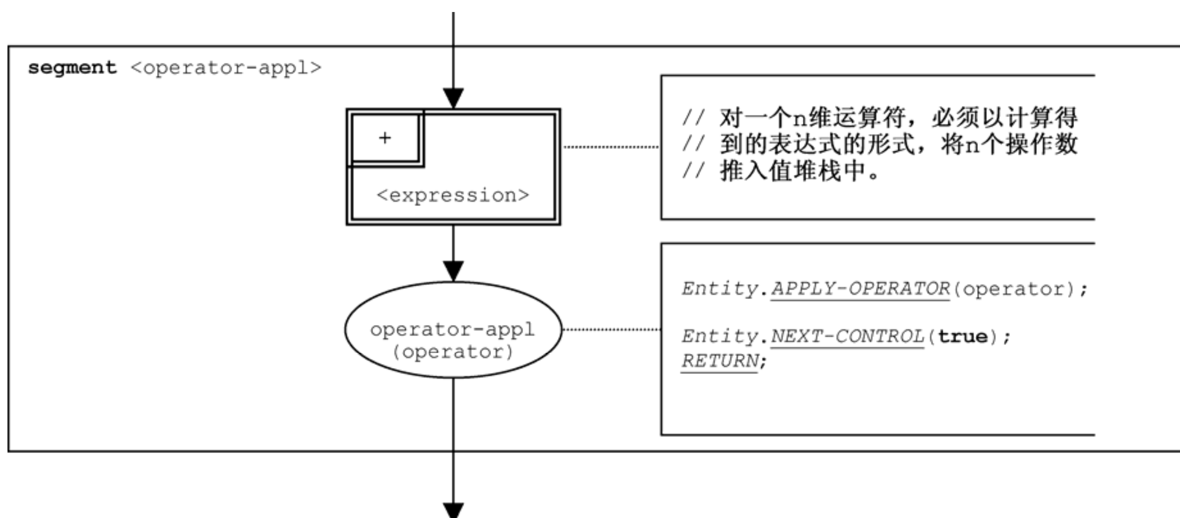


图 74-a/Z.143－流程图片段<operator-appl>

9.18b 流程图片段 <dynamic-error>

在发生动态错误的情况下，测试系统调用图 74-b 中的流程图片段<dynamic-error>。清除分配给测试用例的所有资源，并将 **error** 判定指派给测试用例。将控制提供给控制部分中的语句，控制部分位于发生错误的执行语句之后。

在测试用例不在规定的时间限度内终止的情况下（见第9.17.2节），模块控制调用流程图片段<dynamic-error>。

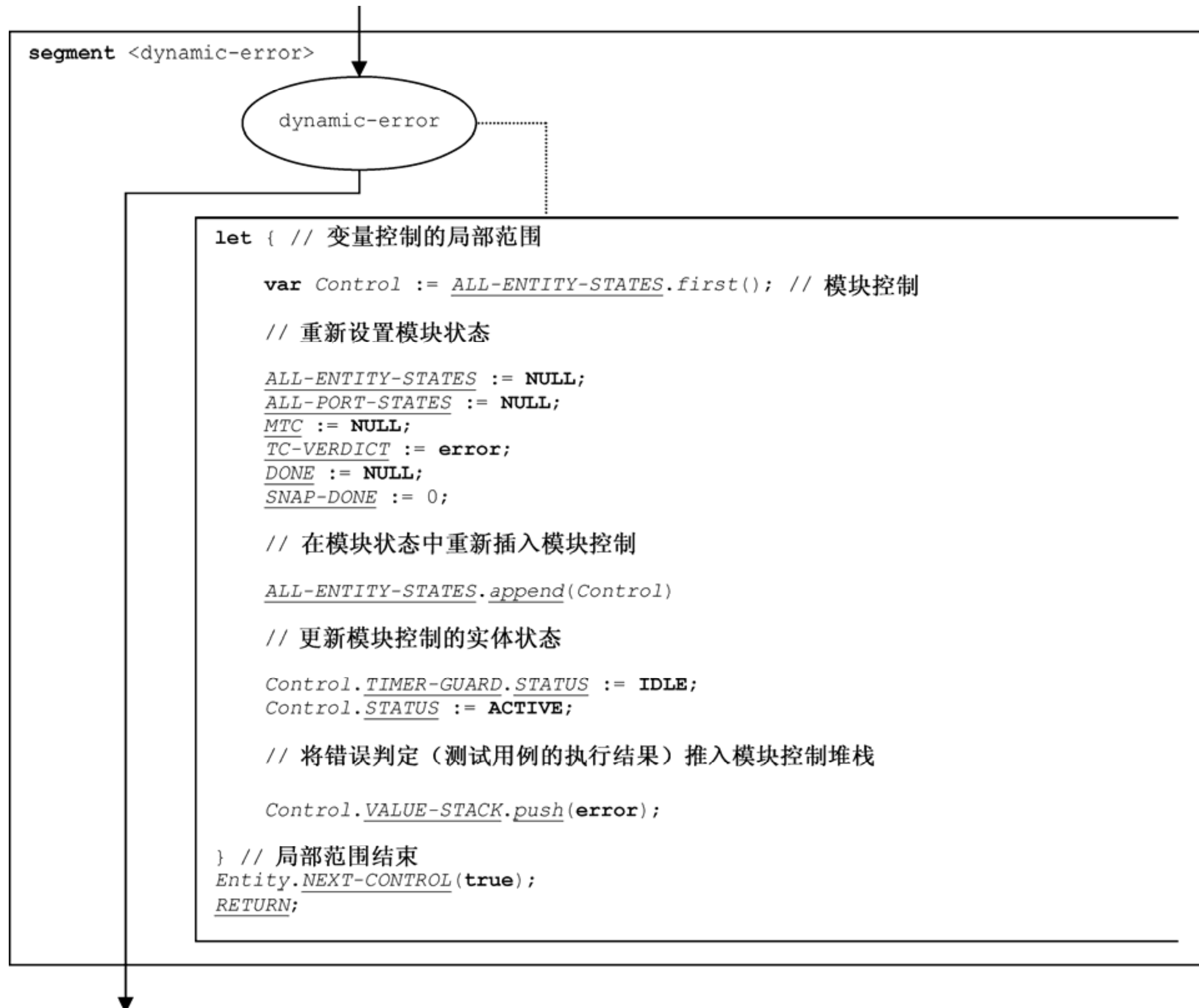


图 74-b/Z.143－流程图片段<dynamic-error>

9.19 流程图片段<finalize-component-init>

流程图片段<finalize-component-init>是描述部件类型定义的流程图的一部分。其执行过程在图 75 中进行定义。

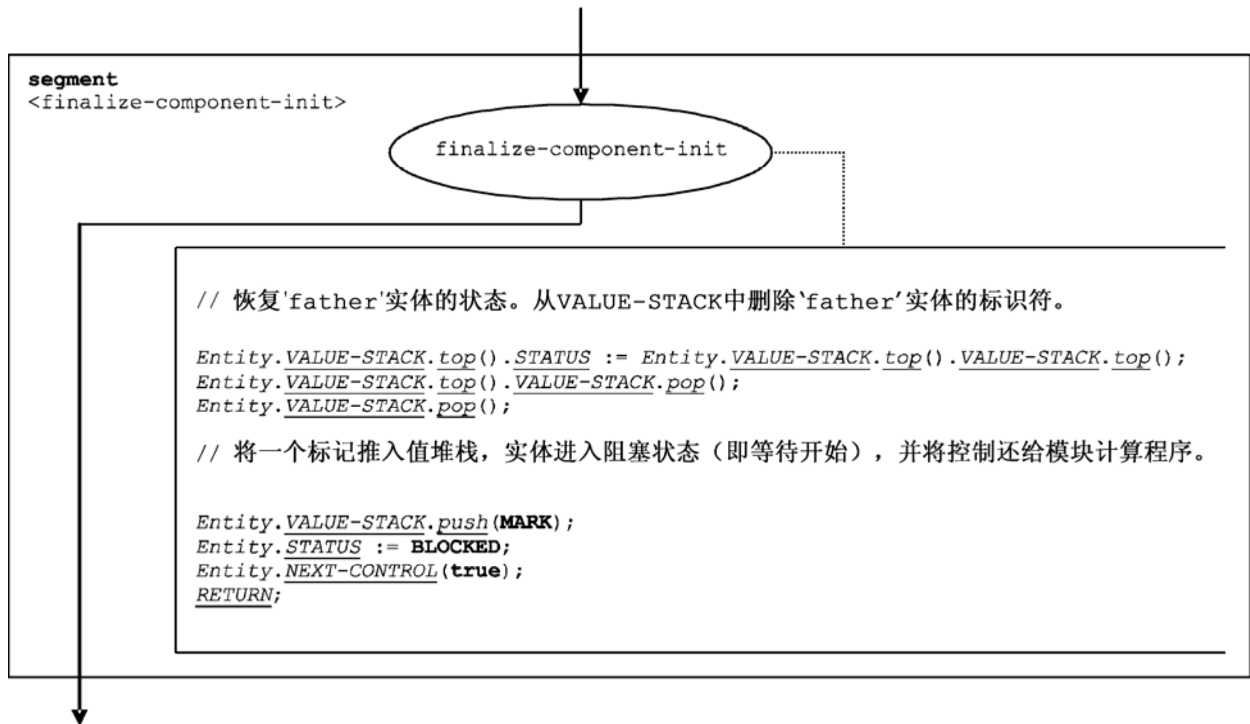


图 75/Z.143－流程图片段<finalize-component-init>

9.20 流程图片段 <init-component-scope>

流程图片段<init-component-scope>是描述部件类型定义的流程图的一部分。其执行过程在图 76 中进行定义。

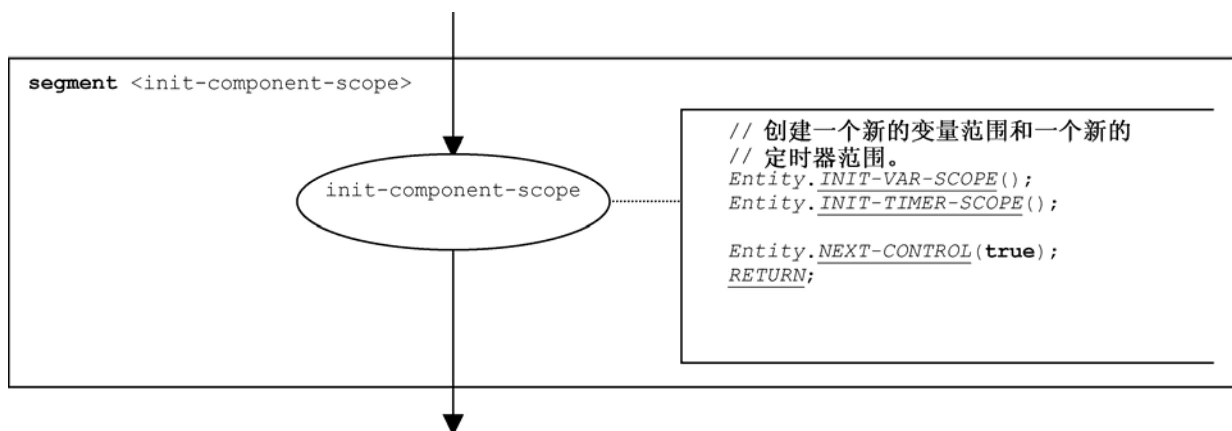


图 76/Z.143－流程图片段<init-component-scope>

9.21 流程图片段<parameter-handling>

流程图片段`<parameter-handling>`用在描述测试用例、可选步骤和函数的流程图之初。它初始化一个新的范围，并为参数处理创建变量和定时器。流程图片段`<parameter-handling>`假设被调用测试用例、可选步骤或函数的调用记录位于值堆栈的顶层。

流程图片段`<parameter-handling>`的执行过程如图 77 所示。

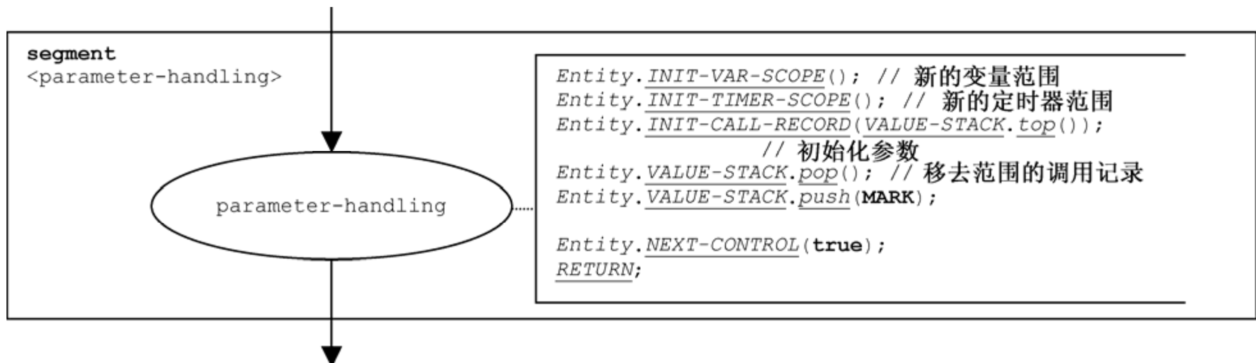


图 77/Z.143－流程图片段<parameter-handling>

9.22 流程图片段<statement-block>

语句块的语法结构是：

```
{ <statement1>; ... ; <statementn> }
```

一个语句块是一个范围单元。当进入一个范围单元时，需要初始化新的变量、定时器和值堆栈范围。当离开一个范围单元时，需要破坏该范围的所有变量、定时器和值堆栈。

注 1 — 语句块不是一个“正式的”TTCN-3概念。语句块只能作为函数体、可选步骤、测试用例和模块控制出现，以及出现在符合语句中，如**alt**、**if-else** 或 **do-while**。

注 2 — 接收操作和可选步骤调用不能出现在语句块中，它们嵌在**alt**语句或**call**操作中。

注 3 — 操作语义也处理像语句这样的操作和声明，也就是说，它们允许出现在语句块中。

注 4 — 一些TTCN-3函数，如**system** 或 **self**，认为是表达式，它们不能作为语句块中的单独语句。其流程图表示未列在图78中。

图 78 中的流程图片段<statement-block>对语句块的执行过程进行定义：

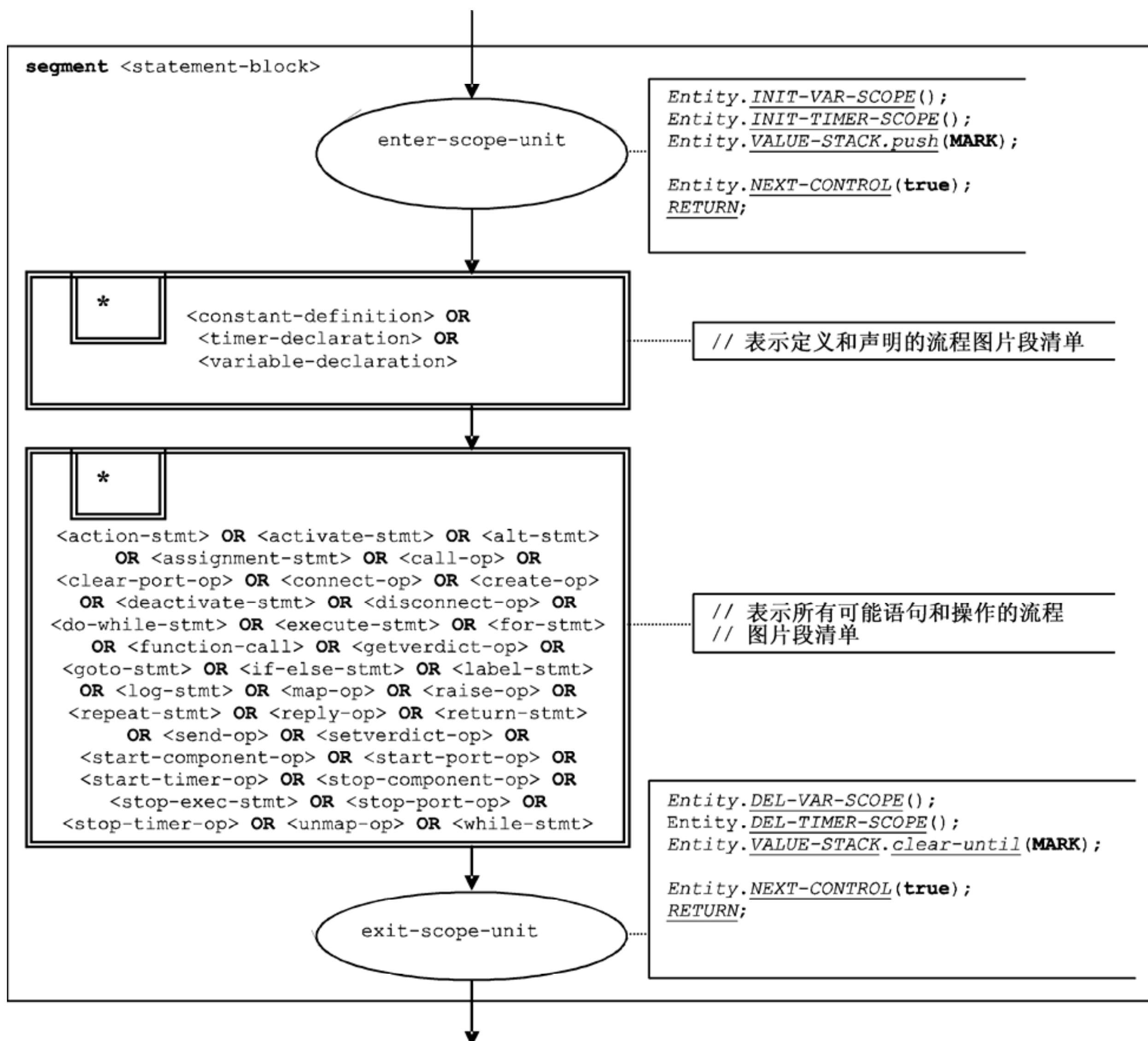


图 78/Z.143－流程图片段<statement-block>

9.23 For 语句

for 语句的语法结构是：

```
for (<assignment>|<variable-declaration>, <boolean expression>, <assignment>) <statement-block>
```

下标变量的初始化以及对应的下标变量处理认为是对下标变量的赋值。也允许在 **for-statement** 中直接声明和初始化下标变量。<boolean-expression>描述 **for-statement** 指定之循环的终止准则，<statement-block>描述循环体。

图79中的流程图片段<for-stmt>对 **for-statement** 的执行过程进行定义。带赋值 assignment <var-declaration-init>的初始<assignment>或可选变量声明（见第9.57.1节）描述下标变量的初始化。decision 节点 **true** 分支中的<assignment>描述对下标变量的处理。**for-statement** 是有关最近声明之下标变量的范围单元，通过节点 enter-var-scope 和 exit-var-scope 来建模。

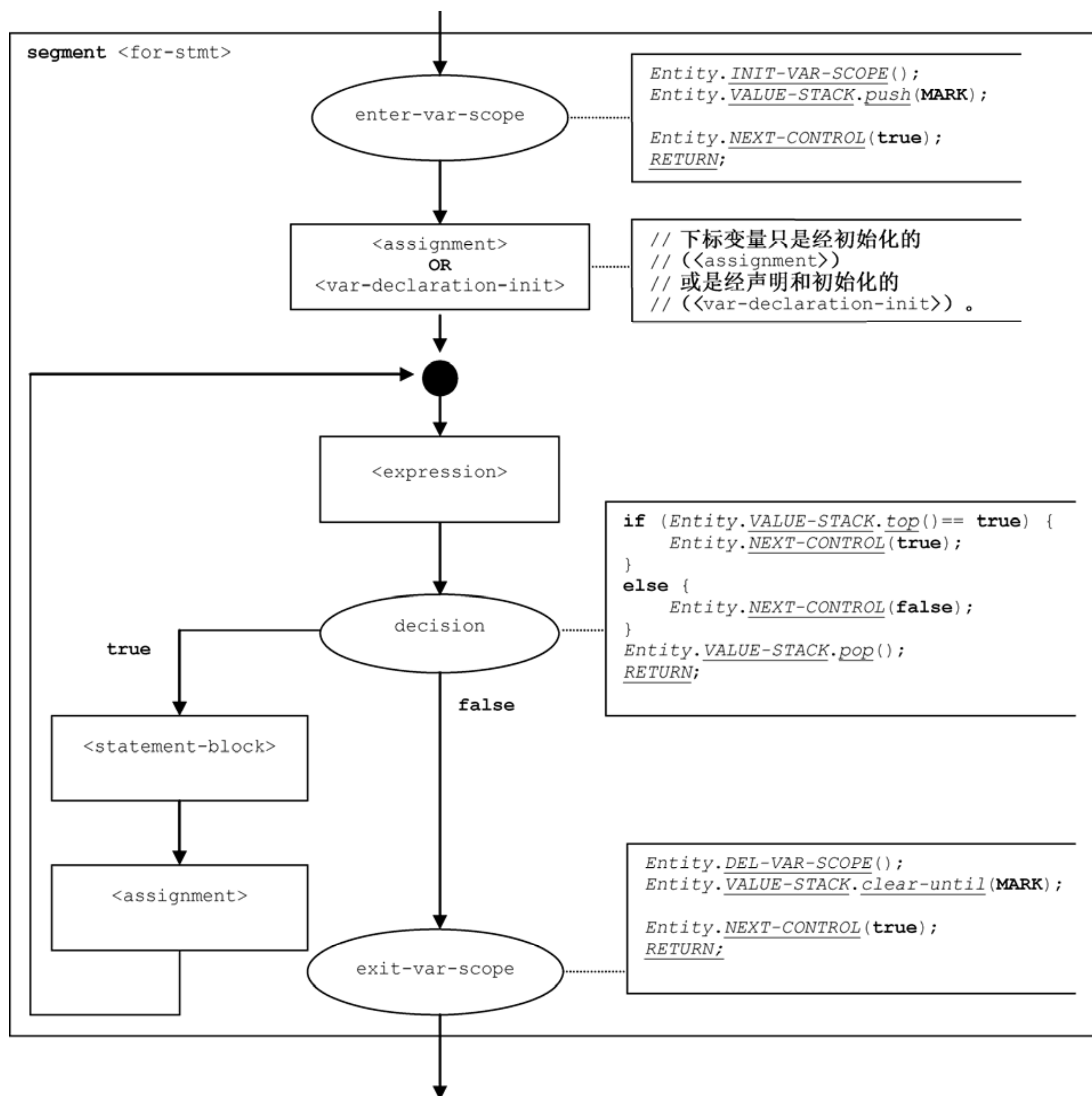


图 79/Z.143－流程图片段<for-stmt>

9.24 函数调用

函数调用的语法结构是：

```
<function-name>([<act-par-desc1>, ... , <act-par-descn>])
```

<function-name>表示一个函数的名称，<act-par-descr$_1$>，...，<act-par-descr$_n$>描述函数调用的实际的参数值。

注 1 — 以相同的方式来处理函数调用和可选步骤调用。因此，可选步骤调用（见第9.4节）指的是本条款。

假设对每个<act-par-desc$_1$>，对应的形参标识符<f-par-Id$_1$>都是已知的，也就是说，可以将上述语法结构扩展为：

```
<function-name>((<f-par-Id1>,<act-par-desc1>), ... , (<f-par-Idn>,<act-par-descn>))
```

图 80 中的流程图片段<function-call>对函数调用的执行过程进行定义。执行过程分为三个步骤。在第一个步骤中，创建函数<function-name>的调用记录。在第二个步骤中，对实际的参数值进行计算，并将之赋给调用记录中的对应字段。在第三个步骤中，需要区分两种情况：被调用的函数是一个用户定义的函数（<user-def-func-call>），也就是说，对函数存在一个流程图表示，或者被调用的函数是一个预定义的函数或外部函数（<predef-ext-func-call>）。在用户定义的函数调用的情况下，控制提供给被调用的函数。在预定义或外部函数的情况下，假设调用函数可用于在一步中执行函数。由被调用的函数负责完成对引用参数和返回值（必须推入值堆栈中）的正确处理，也就是说，超出了本操作语义的范围。

注 2 — 如果函数调用创建一个可选步骤调用的模型，那么只能选择<user-def-func-call>分支，原因是存在一个有关被调用可选步骤的流程图表示。

注 3 — <function call>片段也可用于描述**execute**语句中MTC的开始。在这种情况下，构件一个有关测试用例的调用记录，并将只选择<user-def-func-call>分支。

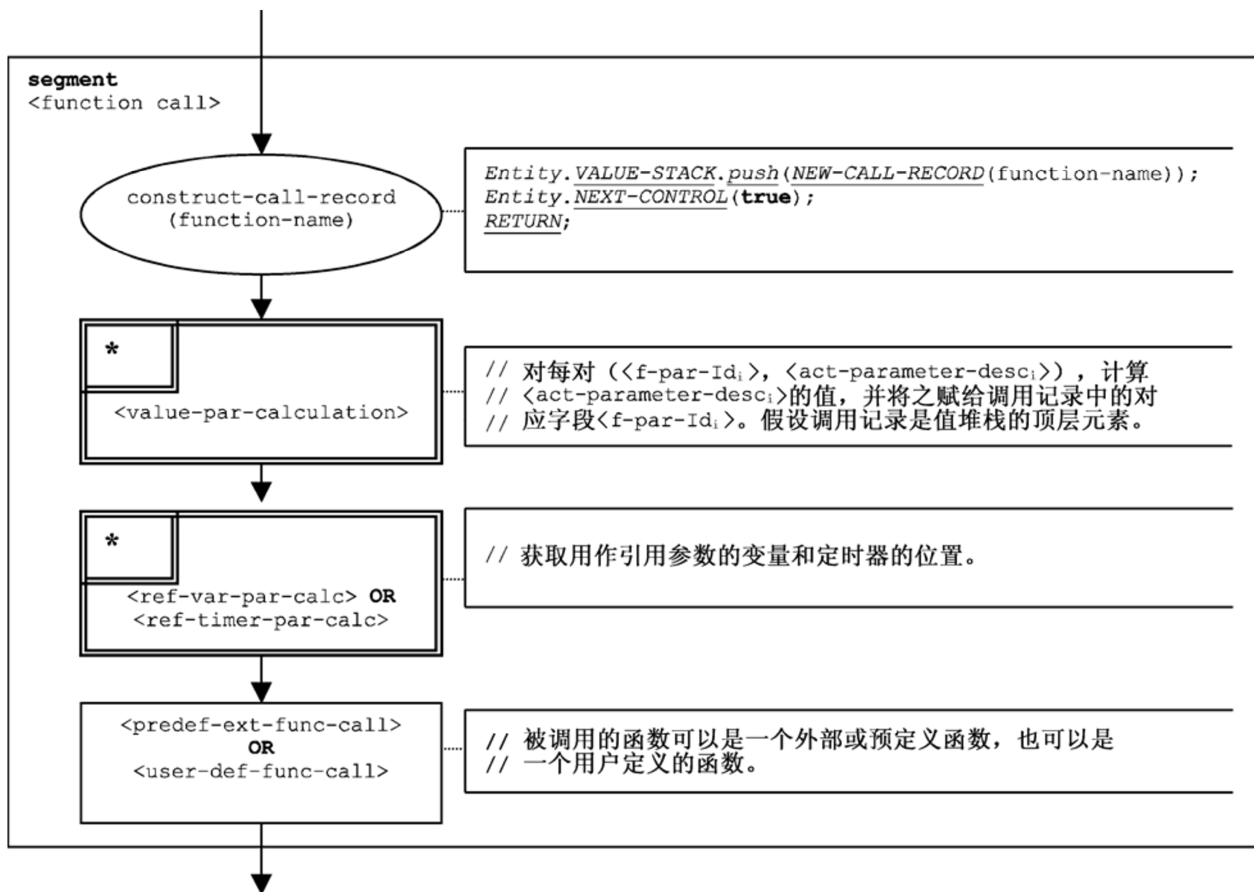


图 80/Z.143－流程图片段<function-call>

9.24.1 流程图片段<value-par-calculation>

流程图片段<value-par-calculation>用于计算实际的参数值，并将之赋给有关函数、可选步骤和测试用例的调用记录中的对应字段。

假设调用记录是值堆栈的顶层元素，需要处理一对：

(<f-par-Id_i>, <act-parameter-$desc_i$>)

需要计算<act-parameter-$desc_i$>，<f-par-Id_i>为形参的标识符，它在值堆栈的调用记录中有一个对应的字段。

流程图片段<value-par-calculation>的执行过程如图 81 所示：

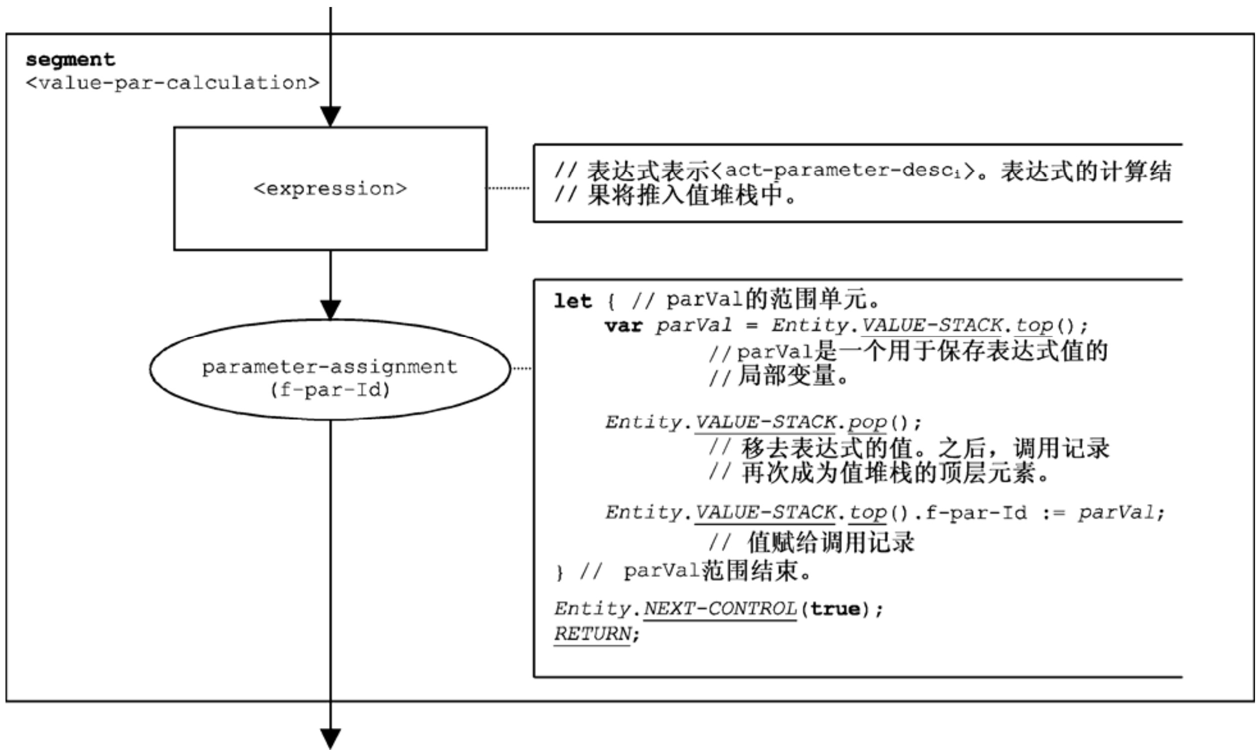


图 81/Z.143－流程图片段<value-par-calculation>

9.24.2 流程图片段<ref-par-var-calc>

流程图片段<ref-par-var-calc>用于获取用作实际引用参数的变量的位置，并将之赋给有关函数、可选步骤和测试用例的调用记录中的对应字段。

假设调用记录是值堆栈的顶层元素，需要处理一对：

```
(<f-par-Idᵢ>, <act-parᵢ>)
```

<act-par$_i$>是实参，需要获取其位置，<f-par-Id$_i$>为形参的标识符，它在值堆栈的调用记录中有一个对应的字段。

流程图片段<ref-par-var-calc>的执行过程如图 82 所示：

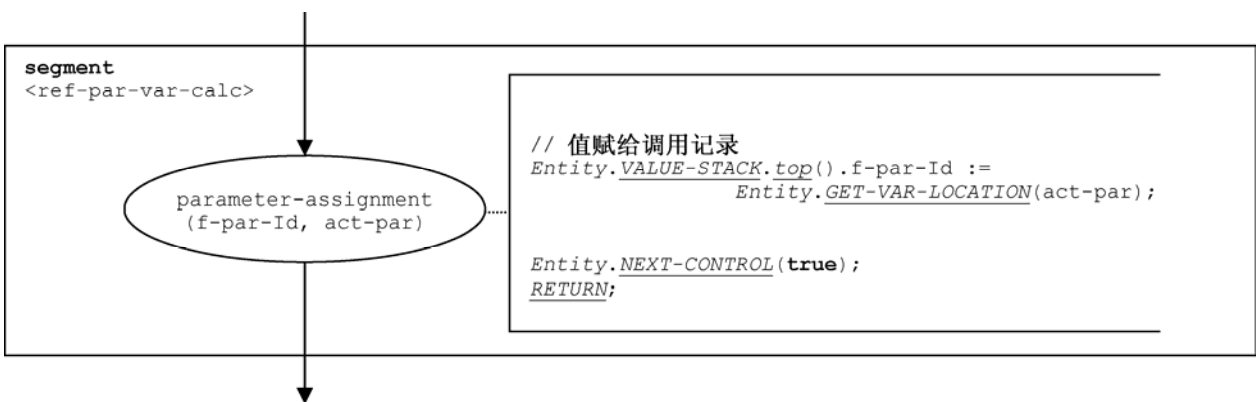


图 82/Z.143－流程图片段<ref-par-var-calc>

9.24.3 流程图片段<ref-par-timer-calc>

流程图片段`<ref-par-timer-calc>`用于获取用作实际引用参数的定时器的位置，并将之赋给有关函数、可选步骤和测试用例的调用记录中的对应字段。

假设调用记录是值堆栈的顶层元素，需要处理一对：

```
(<f-par-Idi>, <act-pari>)
```

`<act-par`$_i$`>`是实参，需要获取其位置，`<f-par-Id`$_i$`>`为形参的标识符，它在值堆栈的调用记录中有一个对应的字段。

流程图片段`<ref-par-timer-calc>`的执行过程如图 83 所示：

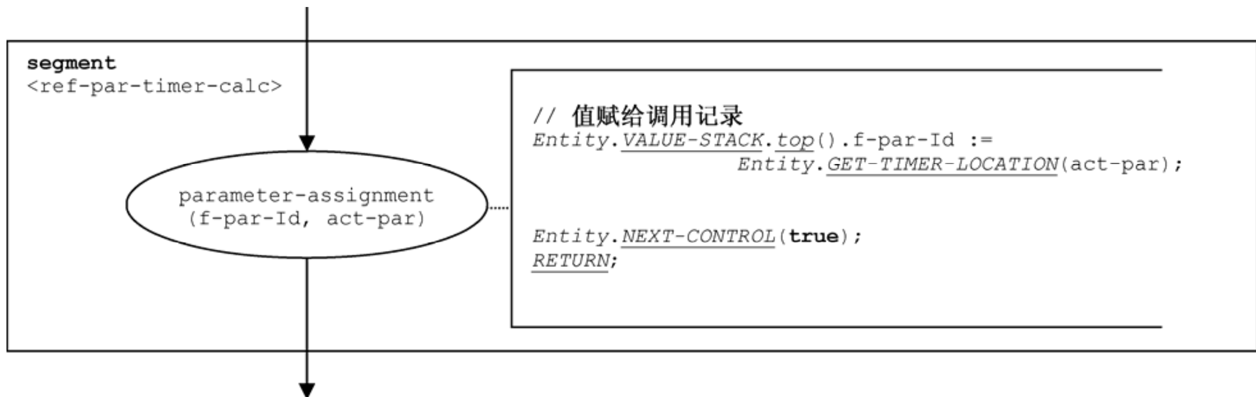


图 83/Z.143－流程图片段<ref-par-timer-calc>

9.24.4 流程图片段<user-def-func-call>

图 84 中的流程图片段`<user-def-func-call>`对控制转给被调用的用户定义函数过程进行描述：

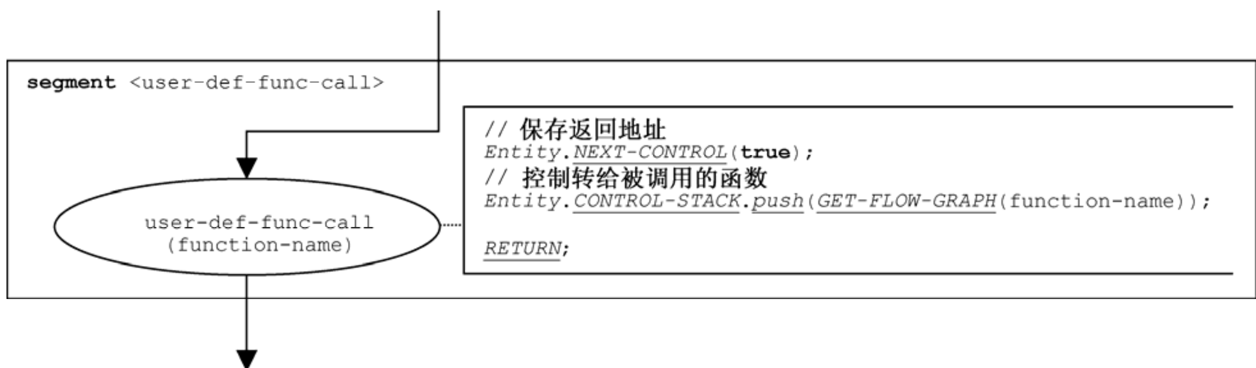


图 84/Z.143－流程图片段<user-def-func-call>

9.24.5 流程图片段<predef-ext-func-call>

图 85 中的流程图片段<predef-ext-func-call>对预定义或外部函数的调用进行描述：

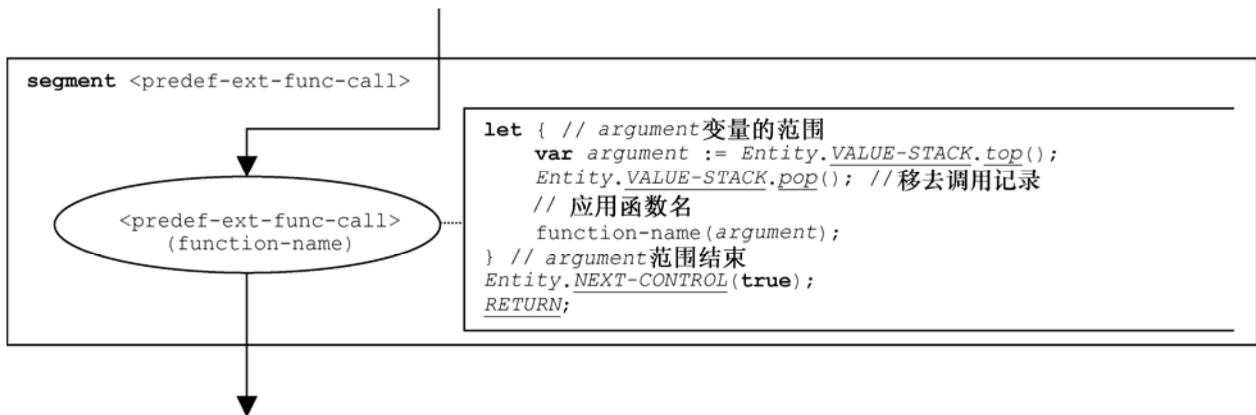


图 85/Z.143－流程图片段<predef-ext-func-call>

9.25 Getcall 操作

getcall 操作的语法结构是：

```
<portId>.getcall (<matchingSpec>) [from <component_expression>] -> [<assignmentPart>]
```

除了关键字 **getcall**，本语法结构等同于 **receive** 操作的语法结构。因此，操作语义以与 **receive** 操作相同的方式来处理 **getcall** 操作。图 86 中的流程图片段<getcall-op>对 **getcall** 操作的执行过程进行了定义，该图指的是与 **receive** 操作（见第 9.37 节）相关的流程图片段。

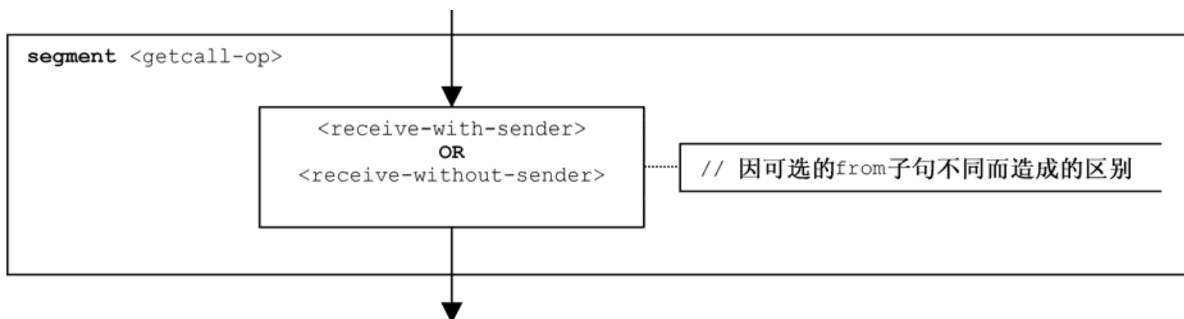


图 86/Z.143－流程图片段<getcall-op>

9.26 Getreply 操作

getreply 操作的语法结构是：

```
<portId>.getreply (<matchingSpec>) [from <component-expression>] [-> <assignmentPart>]
```

除了关键字 **getreply**，本语法结构等同于 **receive** 操作的语法结构。因此，操作语义以与 **receive** 操作相同的方式来处理 **getreply** 操作。图 87 中的流程图片段<getreply-op>对 **getreply** 操作的执行过程进行了定义，该图指的是与 **receive** 操作（见第 9.37 节）相关的流程图片段。

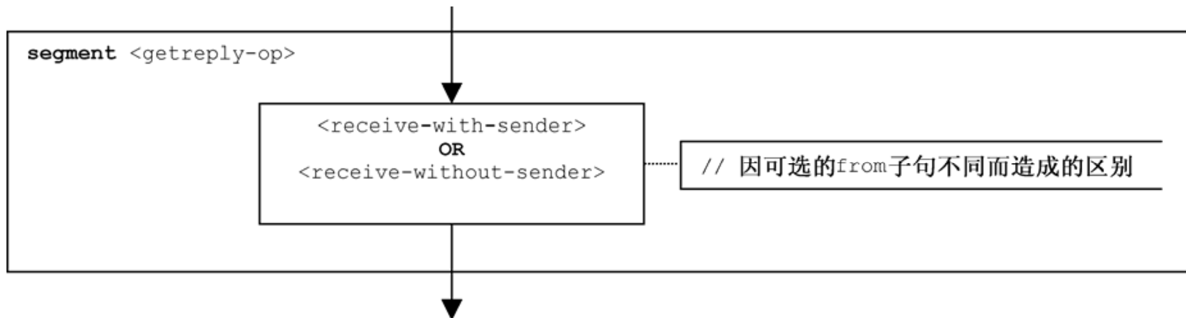


图 87/Z.143－流程图片段<getreply-op>

9.27 Getverdict 操作

getverdict 操作的语法结构是：

```
getverdict
```

图 88 中的流程图片段<getverdict-op>对 **getverdict** 操作的执行过程进行定义：

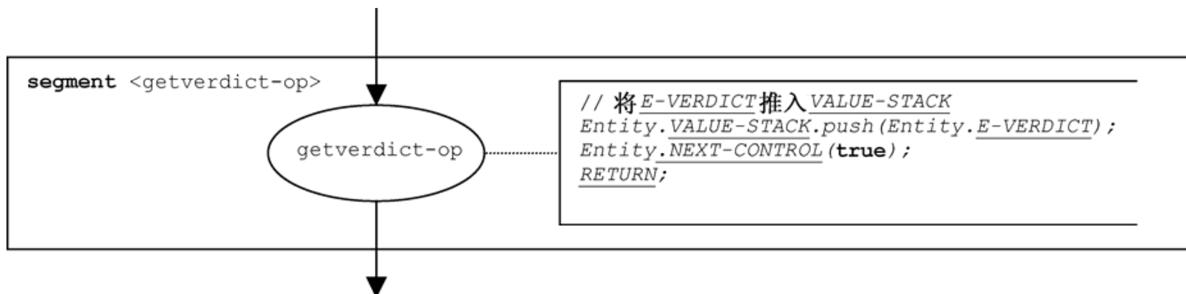


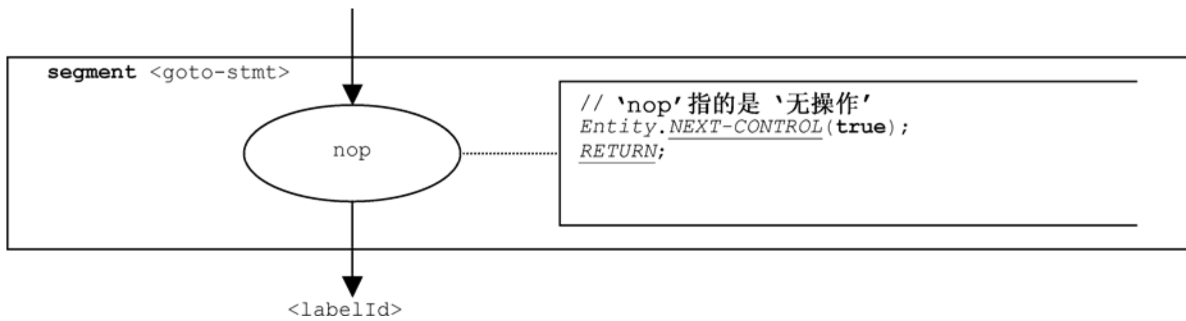
图 88/Z.143－流程图片段<getverdict-op>

9.28 Goto 语句

goto 语句的语法结构是：

```
goto <labelId>
```

图 89 中的流程图片段<goto-stmt>对 **goto** 语句的执行过程进行定义：



注 — **goto**语句的<labelId>参数表示将控制转至标签 *<labelId>*所定义的位置（也见第9.30节）。

图 89/Z.143－流程图片段<goto-stmt>

9.29 If-else 语句

if-else 语句的语法结构是：

```
if (<boolean-expression>) <statement-block1>
    [else <statement-block2>]
```

if-else 语句的 else 部分是可选的。

图 90 中的流程图片段<if-else-stmt>对 **if-else** 语句的执行过程进行定义：

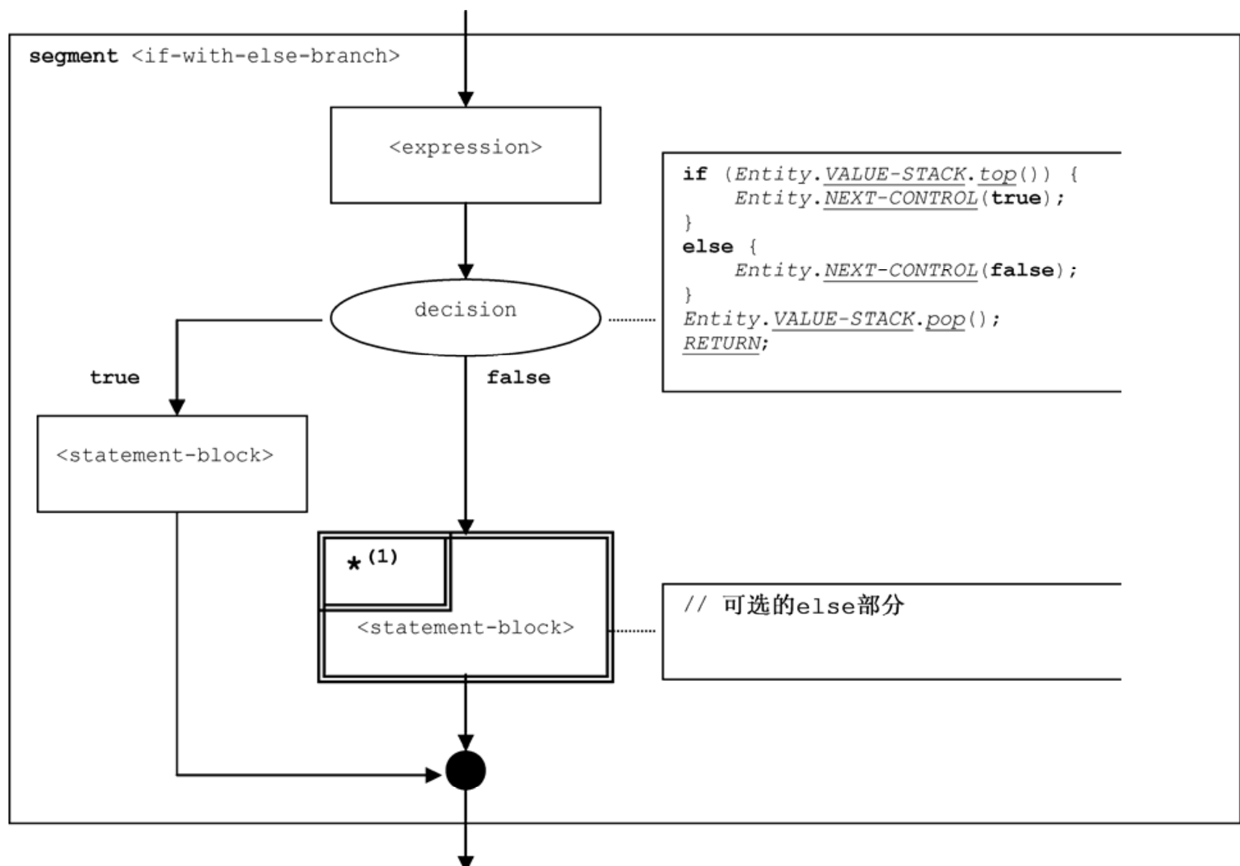


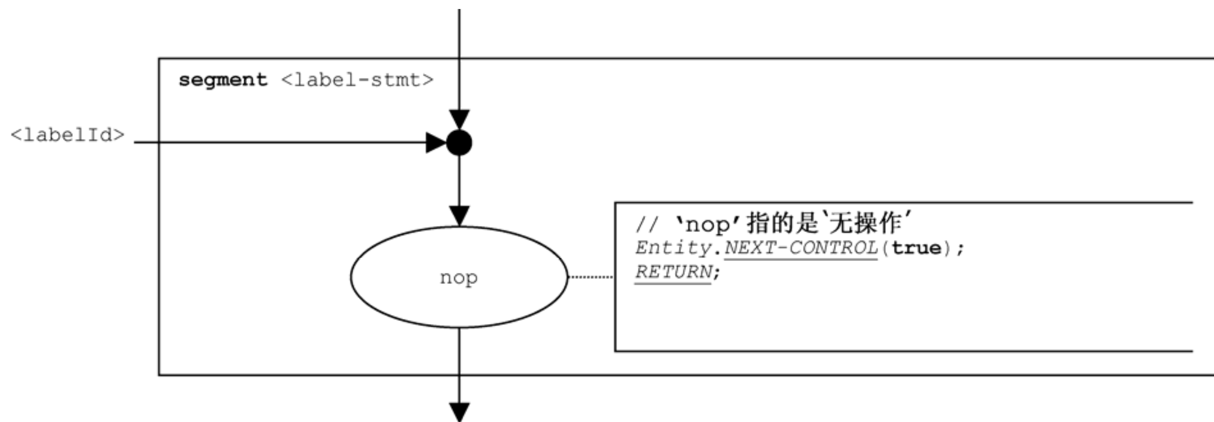
图 90/Z.143－流程图片段<if-else-stmt>

9.30 Label 语句

label 语句的语法结构是：

```
label <labelId>
```

图 91 中的流程图片段<label-stmt>对 **label** 语句的执行过程进行定义：



注 — label语句的<labelId>参数表示通过**goto**语句跳转至某个标签的可能性（也可参见第9.28节）。

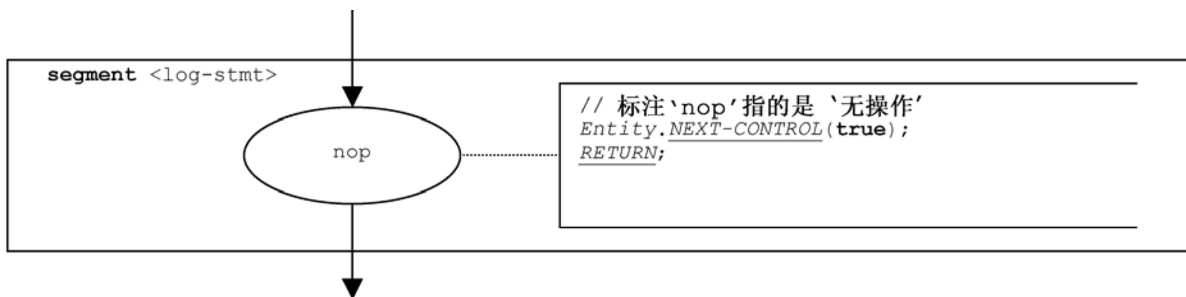
图 91/Z.143－流程图片段<label-stmt>

9.31 Log 语句

log 语句的语法结构是：

```
log (<informal-description>)
```

图 92 中的流程图片段<log-stmt>对 **log** 语句的执行过程进行定义：



注 — **log**语句的<informal description>参数对操作语义没有任何意义，因此不在流程图片段中做描述。

图 92/Z.143－流程图片段<log-stmt>

9.32 Map 操作

map 操作的语法结构是：

```
map(<component-expression>:<portId1>, system:<portId2>)
```

认为标识符<portId1> 和 <portId2>是对应测试部件和测试系统接口的端口标识符。通过部件引用<component-expression>的方式来引用<portId1>所属的部件。引用可以保存在变量中或通过函数返回，也就是说，它是一个表达式，将计算得到一个部件引用。值堆栈用于保存部件引用。

注 — **map**操作不在乎**system**:<portId>语句是作为第一个参数还是第二个参数出现。为简化起见，假设它总为第二个参数。

图 93 中的流程图片段<map-op>对 **map** 操作的执行过程进行定义：

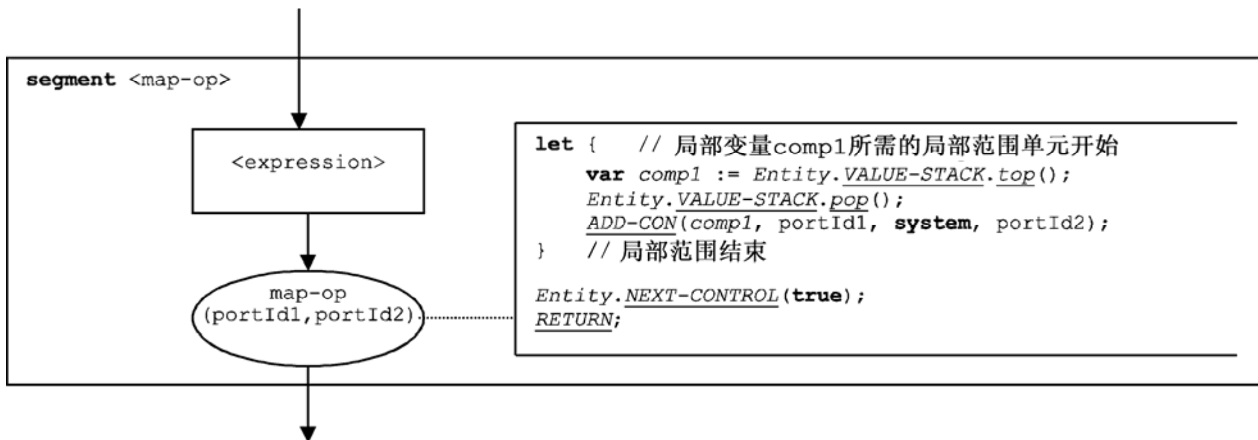


图 93/Z.143－流程图片段<map-op>

9.33 Mtc 操作

mtc 操作的语法结构是：

```
mtc
```

图 94 中的流程图片段<mtc-op>对 **mtc** 操作的执行过程进行定义：

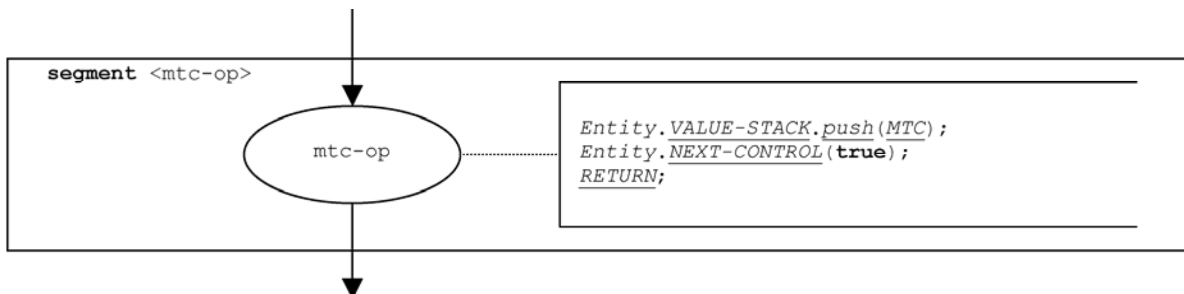


图 94/Z.143－流程图片段<mtc-op>

9.34 Port声明

port 声明的语法结构是：

```
<portType> <portName>
```

可以在部件类型定义中找到端口声明。端口声明的作用是在创建对应类型的一个新部件时创建一个新的端口。图 95 中的流程图片段<port-declaration>对端口声明的执行过程进行定义：

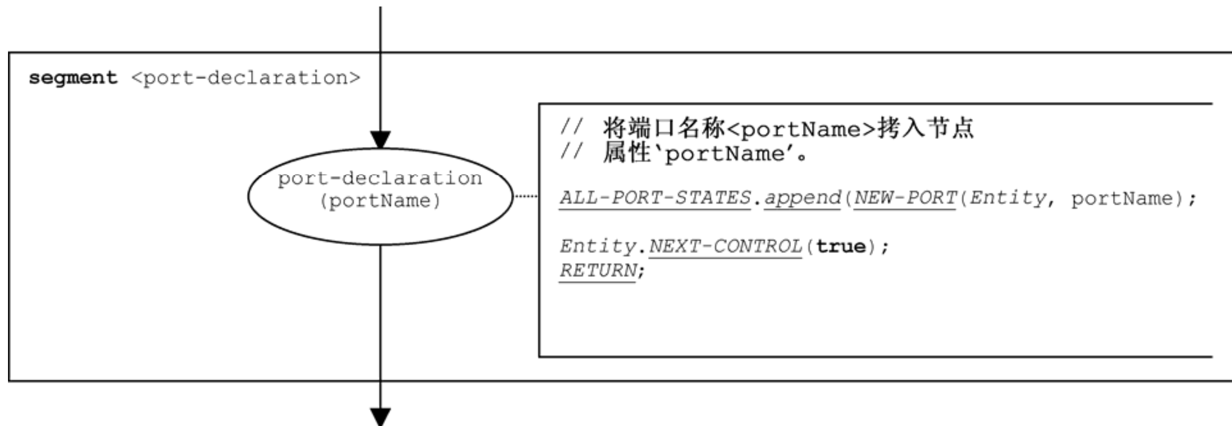


图 95/Z.143－流程图片段<port-declaration>

9.35 Raise 操作

raise 操作的语法结构是：

```
<portId>.raise (<exceptSpec>) [to <component-expression>]
```

to 子句中的可选<component-expression>指的是接收方实体。它可以以一个变量值的形式来提供，也可以以一个函数的返回值的形式来提供。

图 96 中的流程图片段<raise-op>对 **raise** 操作的执行过程进行定义：

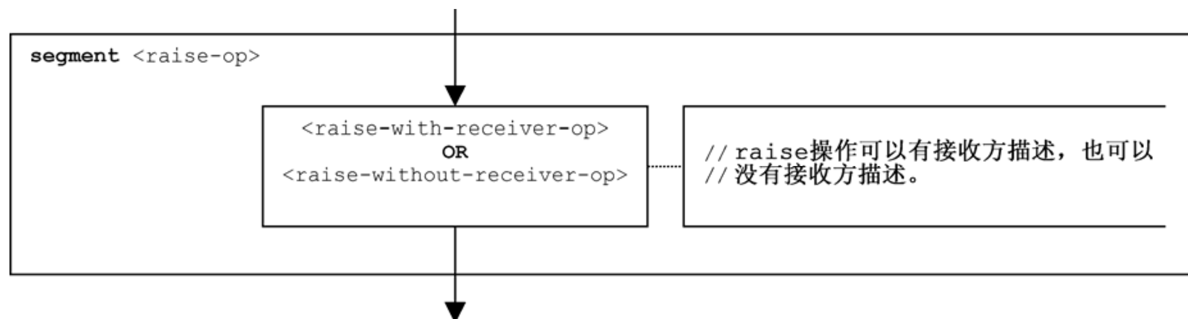


图 96/Z.143－流程图片段<raise-op>

9.35.1 流程图片段<raise-with-receiver-op>

图 97 中的流程图片段<raise-with-receiver-op>对 **raise** 操作的执行过程进行定义，当中的接收方以一个表达式的形式来描述。

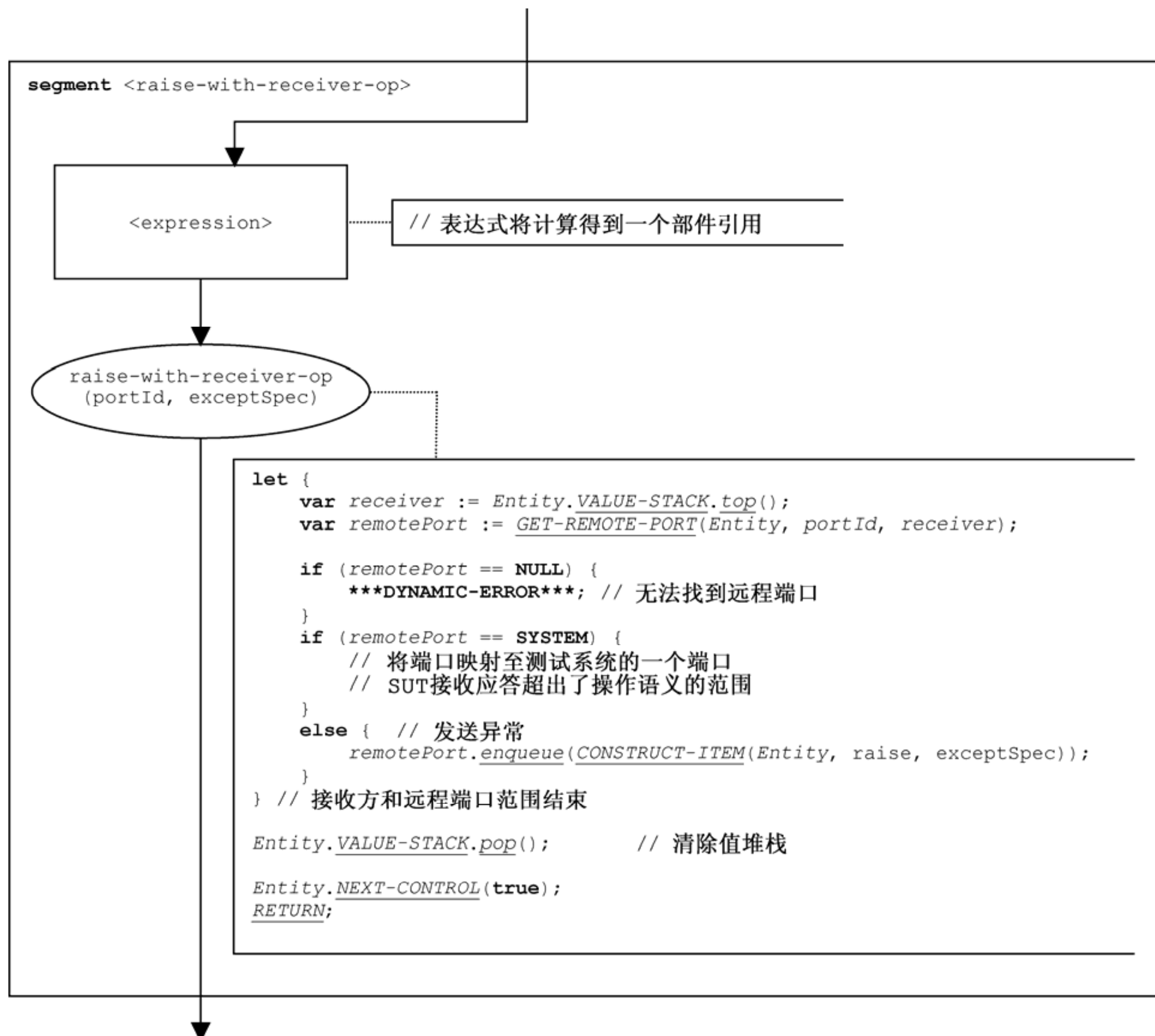


图 97/Z.143－流程图片段<raise-with-receiver-op>

9.35.2 流程图片段<raise-without-receiver-op>

图 98 中的流程图片段`<raise-without-receiver-op>`对不带 **`to`** 子句的 **`raise`** 操作的执行过程进行定义：

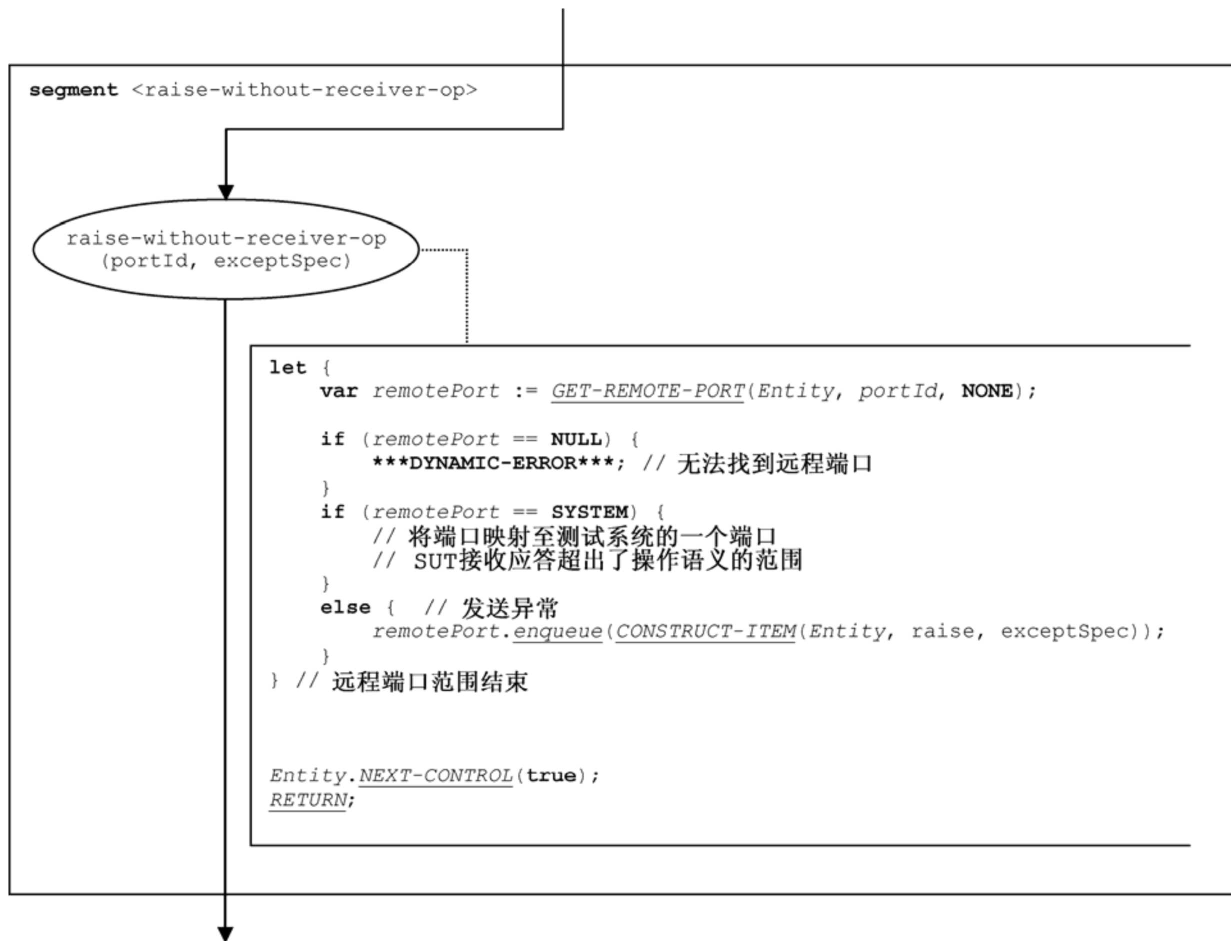


图 98/Z.143－流程图片段<raise-without-receiver-op>

9.36 Read定时器操作

`read` 定时器操作的语法结构是：

```
<timerId>.read
```

图 99 中的流程图片段`<read-timer-op>`对 **`read`** 定时器操作的执行过程进行定义。

`read` 定时器操作区分其在 **`alt`** 语句或阻塞 **`call`** 操作的布尔防卫中的用法与在所有其他情况中的用法之间的差别。如果用在布尔防卫中，那么 **`read`** 定时器操作的结果基于实际的快照，即定时器绑定的 *SNAP-STATUS* 和 *SNAP-VALUE* 条目，而在所有其它情况下，为用于确定操作结果的、定时器绑定的 *STATUS*、*ACT-DURATION* 和 *TIME-LEFT* 条目。

```
segment <read-timer-op>

read-timer-op
(timerId)

let { // 变量myValue的局部范围

    var float myValue;

    if (Entity.STATUS == SNAPSHOT) {
        if (Entity.timerId.SNAP-STATUS == RUNNING) {
            myValue := Entity.timerId.SNAP-VALUE;
        }
        else {
            myValue := 0.0;
        }
    }
    else {
        if (Entity.timerId.STATUS == RUNNING) {
            myValue := Entity.timerId.ACT-DURATION - Entity.timerId.TIME-LEFT;

        }
        else {
            myValue := 0.0;
        }
    }

    Entity.VALUE-STACK.push(myValue);

} // 局部范围结束

Entity.NEXT-CONTROL(true);
RETURN;
```

图 99/Z.143－流程图片段<read-timer-op>

9.37 Receive 操作

receive 操作的语法结构是：

```
<portId>.receive (<matchingSpec>) [from <component-expression>] [-> <assignmentPart>]
```

from 子句中的可选<component-expression>指的是发送方实体。它可以以一个变量值的形式来提供，也可以以一个函数的形式来提供，也就是说，假设它是一个表达式。如果收到的信息匹配于匹配说明<matchingSpec>并匹配（可选的）**from** 子句，那么可选的<assignmentPart>表示接收到信息的赋值。

图 100 中的流程图片段<receive-op>对 **receive** 操作的执行过程进行定义：

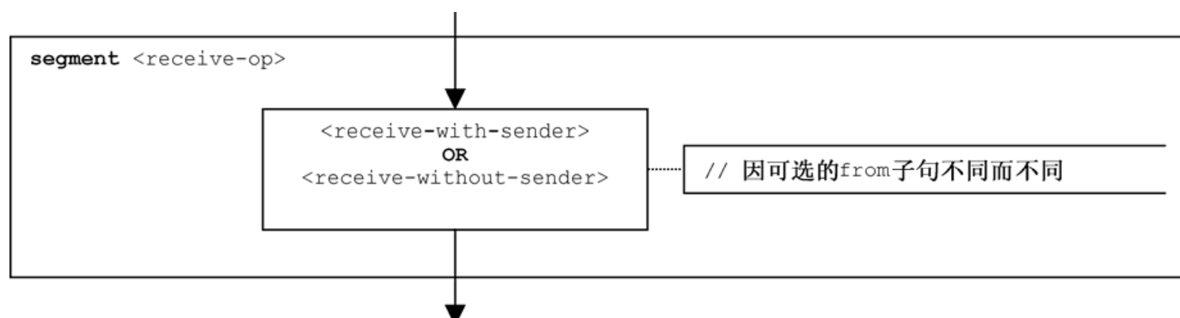


图 100/Z.143－流程图片段<receive-op>

9.37.1 流程图片段<receive-with-sender>

图 101 中的流程图片段<receive-with-sender>对 **receive** 操作的执行过程进行定义，当中的发送方以一个表达式的形式来描述。

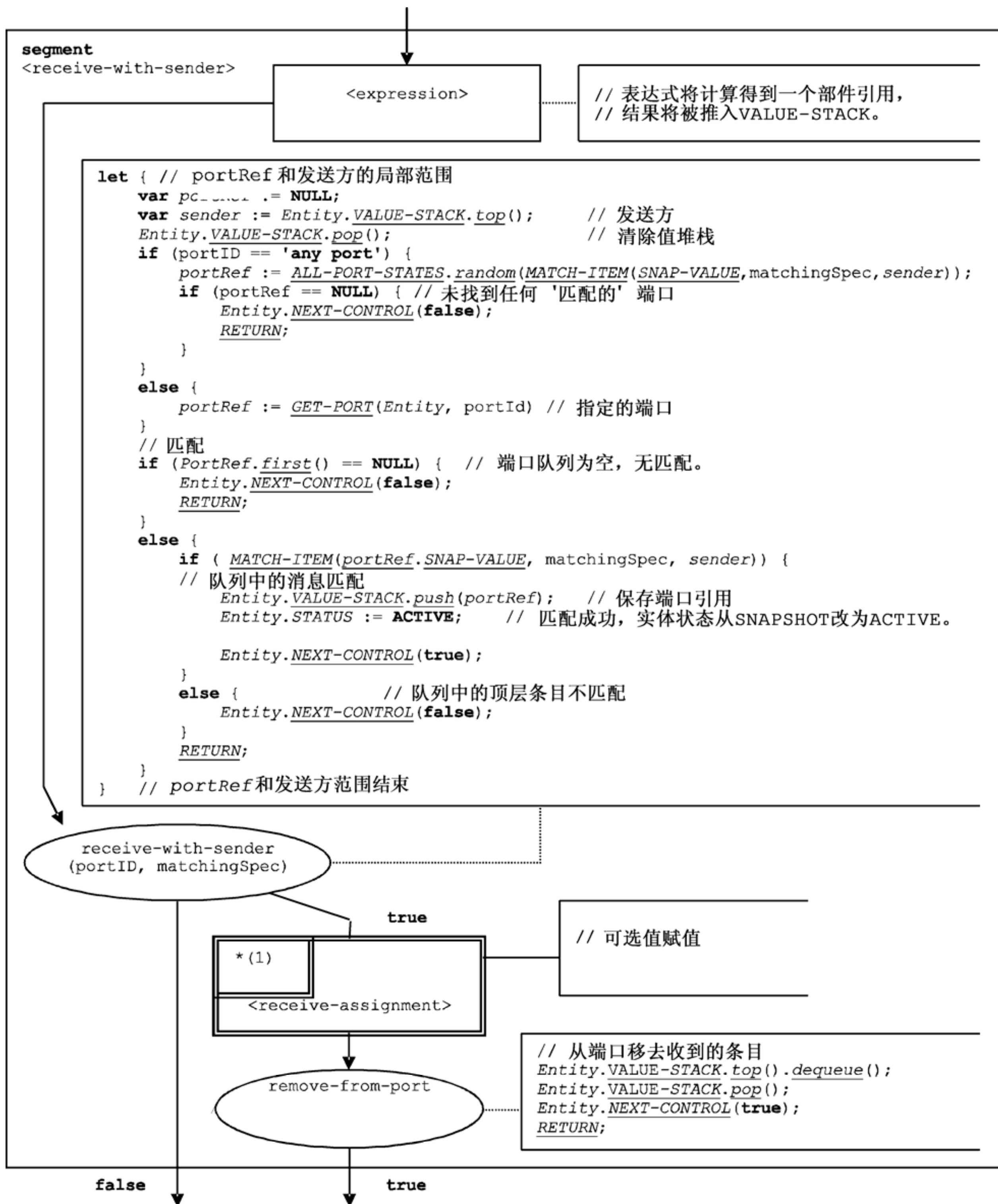


图 101/Z.143－流程图片段<receive-with-sender>

9.37.2 流程图片段<receive-without-sender>

图 102 中的流程图片段`<receive-without-sender>`对不带 **`from`** 子句的 **`receive`** 操作的执行过程进行定义：

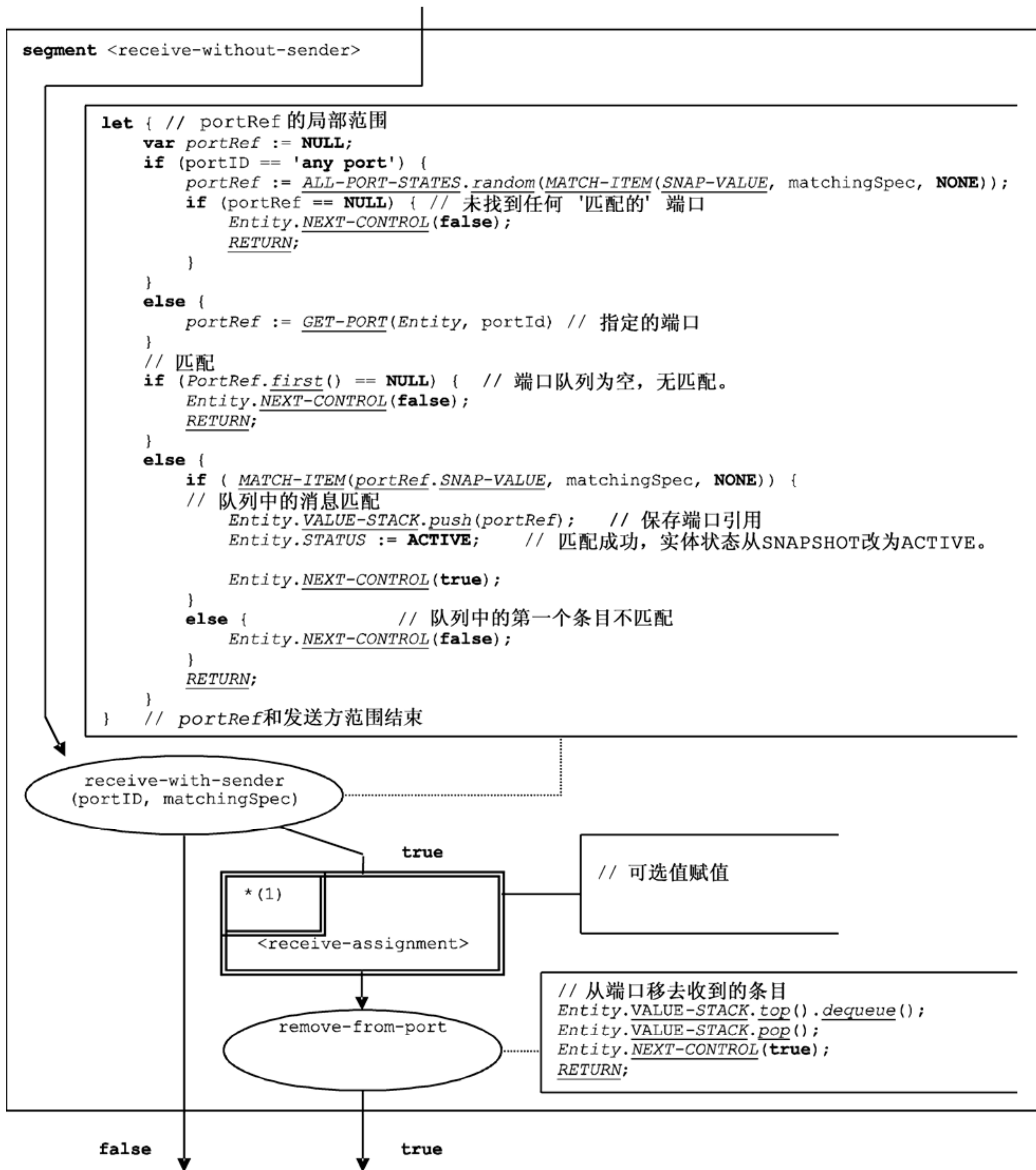


图 102/Z.143－流程图片段<receive-without-sender>

9.37.3 流程图片段<receive-assignment>

图 103 中的流程图片段<receive-assignment>对从收到的消息获取信息及其对变量的赋值进行定义：

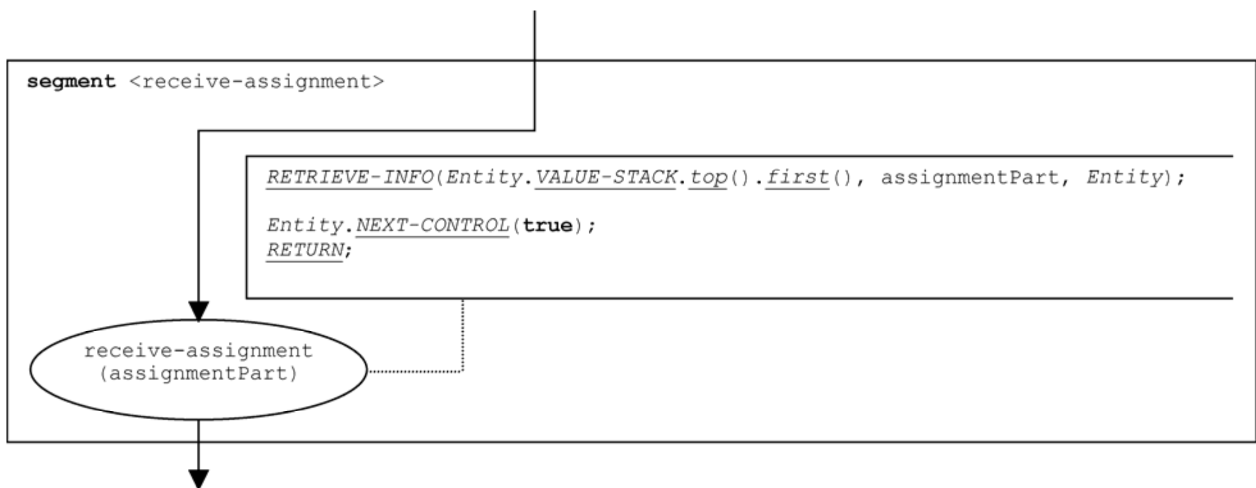


图 103/Z.143－流程图片段<receive-assignment>

9.38 Repeat 语句

repeat 语句的语法结构是：

```
repeat
```

基本地，**repeat** 语句是一个不带返回值的 **return** 语句，它也将实体状态变为 **REPEAT**。状态 **REPEAT** 将强行对当中的 **repeat** 语句已执行的 **alt** 语句进行重新计算。图 104 中的流程图片段<repeat-stmt>对 **repeat** 语句的执行过程进行定义：

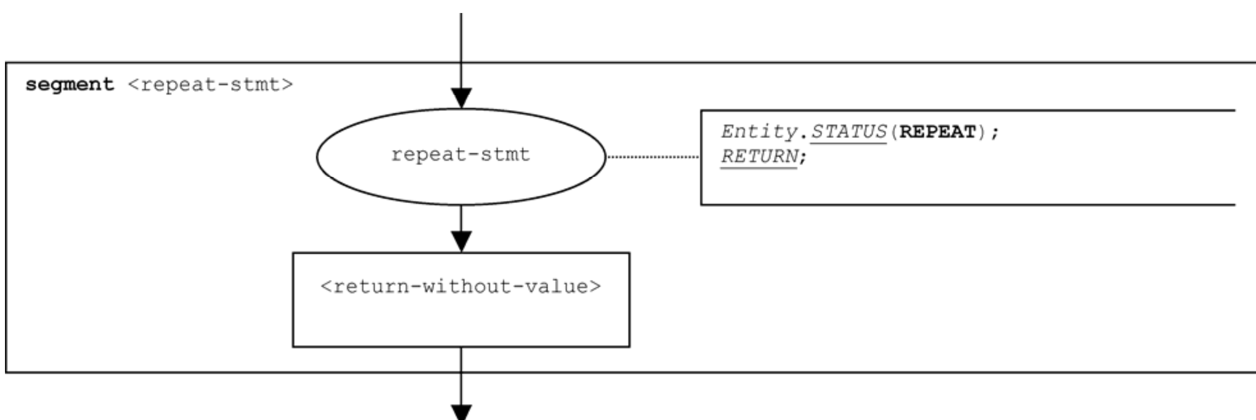


图 104/Z.143－流程图片段<repeat-stmt>

9.39 Reply 操作

reply 操作的语法结构是：

```
<portId>.reply (<replySpec>) [to <component-expression>]
```

to 子句中的可选<component-expression>指的是接收方实体。它可以以一个变量值的形式来提供，也可以以一个函数的返回值形式来提供。

图 105 中的流程图片段<reply-op>对 **reply** 操作的执行过程进行定义：

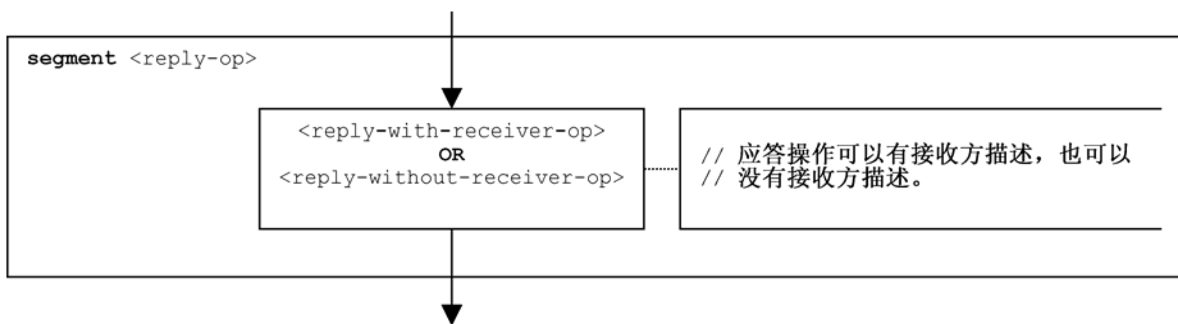


图 105/Z.143－流程图片段<reply-op>

9.39.1 流程图片段<reply-with-receiver-op>

图 106 中的流程图片段<reply-with-receiver-op>对 **reply** 操作的执行过程进行定义，当中的接收方以一个表达式的形式来描述。

```
segment <reply-with-receiver-op>

<expression>        // 表达式将计算得到一个部件引用

reply-with-receiver-op
(portId, replySpec)

let {
    var receiver := Entity.VALUE-STACK.top();
    var remotePort := GET-REMOTE-PORT(Entity, portId, receiver);

    if (remotePort == NULL) {
        ***DYNAMIC-ERROR***; // 无法找到远程端口
    }
    if (remotePort == SYSTEM) {
        // 将端口映射至测试系统的一个端口
        // SUT接收应答超出了操作语义的范围

    }
    else {  // 发送应答
        remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
    }
} // 接收方和远程端口范围结束

Entity.VALUE-STACK.pop();          // 清除值堆栈

Entity.NEXT-CONTROL(true);
RETURN;
```

图 106/Z.143－流程图片段<reply-with-receiver-op>

9.39.2 流程图片段<reply-without-receiver-op>

图 107 中的流程图片段`<reply-without-receiver-op>`对不带 **to** 子句的 **reply** 操作的执行过程进行定义：

图 107/Z.143－流程图片段<reply-without-receiver-op>

9.40 Return 语句

return 语句的语法结构是：

```
return [<expression>]
```

可选`<expression>`描述一个可能的函数返回值。执行 **return** 语句意味着控制离开实际的范围单元，也就是说，必须删除只有在本范围内才已知的变量和定时器，并必须更新值堆栈。如果 **return** 语句是行为描述中的最后一个语句，那么它具有一个 **stop** 部件操作的作用

注 — 测试用例和模块控制总以**stop**部件操作结束。这归因于其流程图描述（见第8.2节）。只有其它测试部件可以用一个**return**语句来终止。

图 108 中的流程图片段`<return-stmt>`对 **return** 语句的执行过程进行定义：

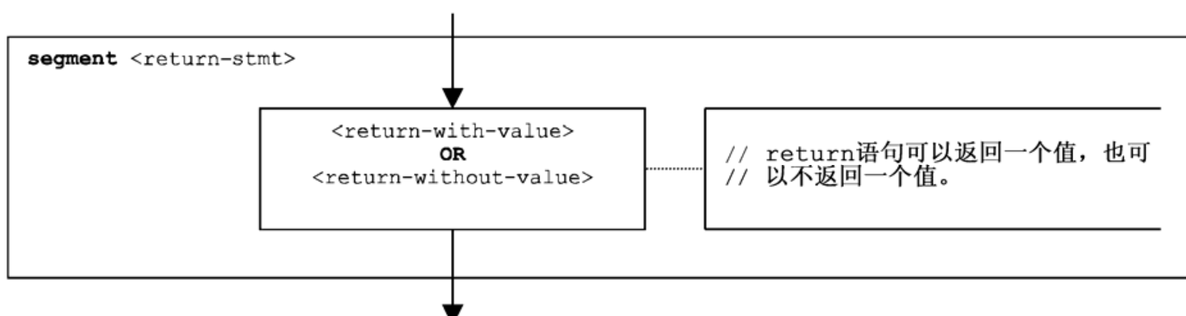


图 108/Z.143－流程图片段<return-stmt>

9.40.1 流程图片段<return-with-value>

图 109 中的流程图片段`<return-with-value>`对以表达式形式返回一个指定值的 **`return`** 语句的执行过程进行定义：

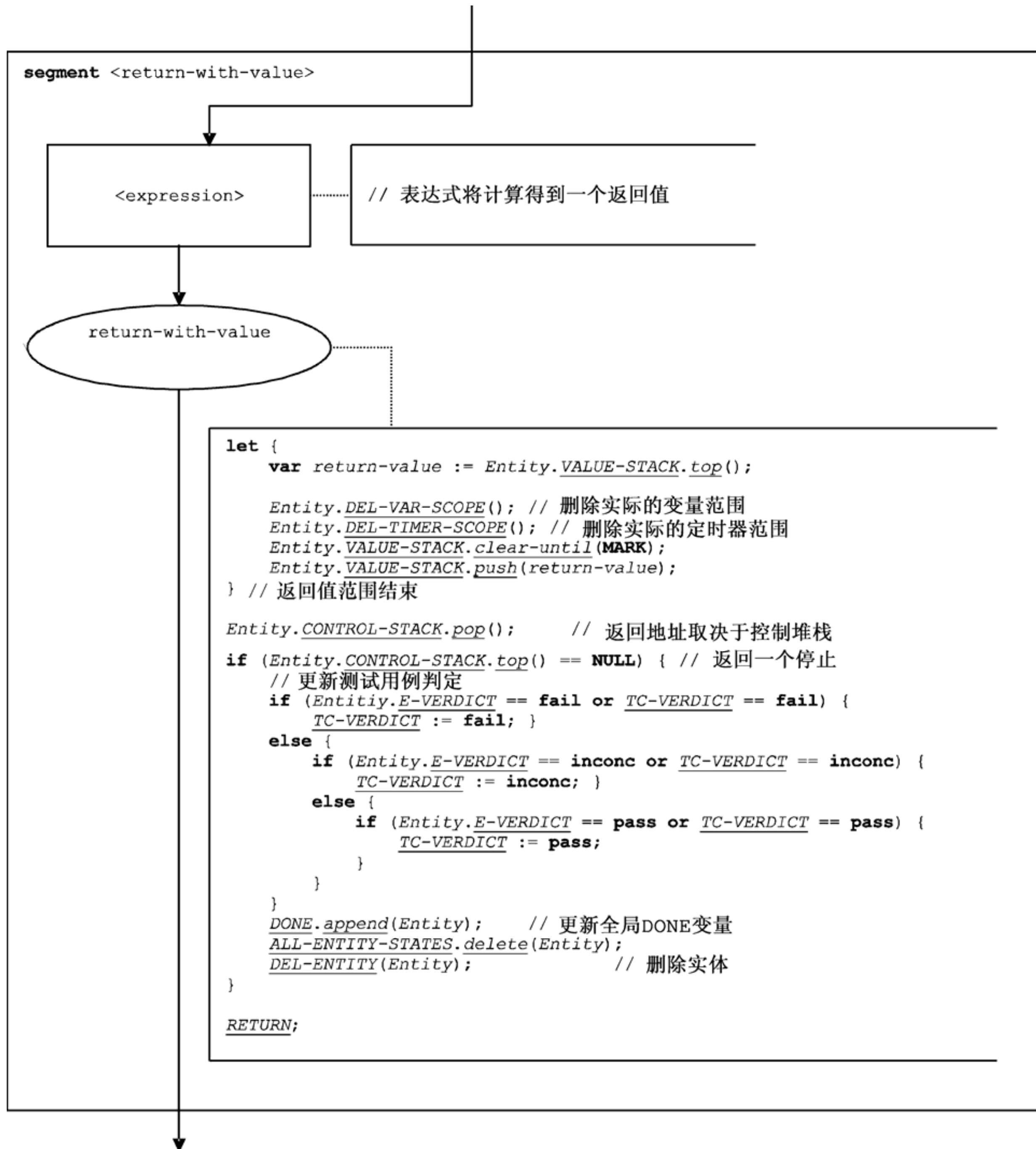


图 109/Z.143－流程图片段<return-with-value>

9.40.2 流程图片段<return-without-value>

图 110 中的流程图片段<return-without-value>对不返回值的 **return** 语句的执行过程进行定义：

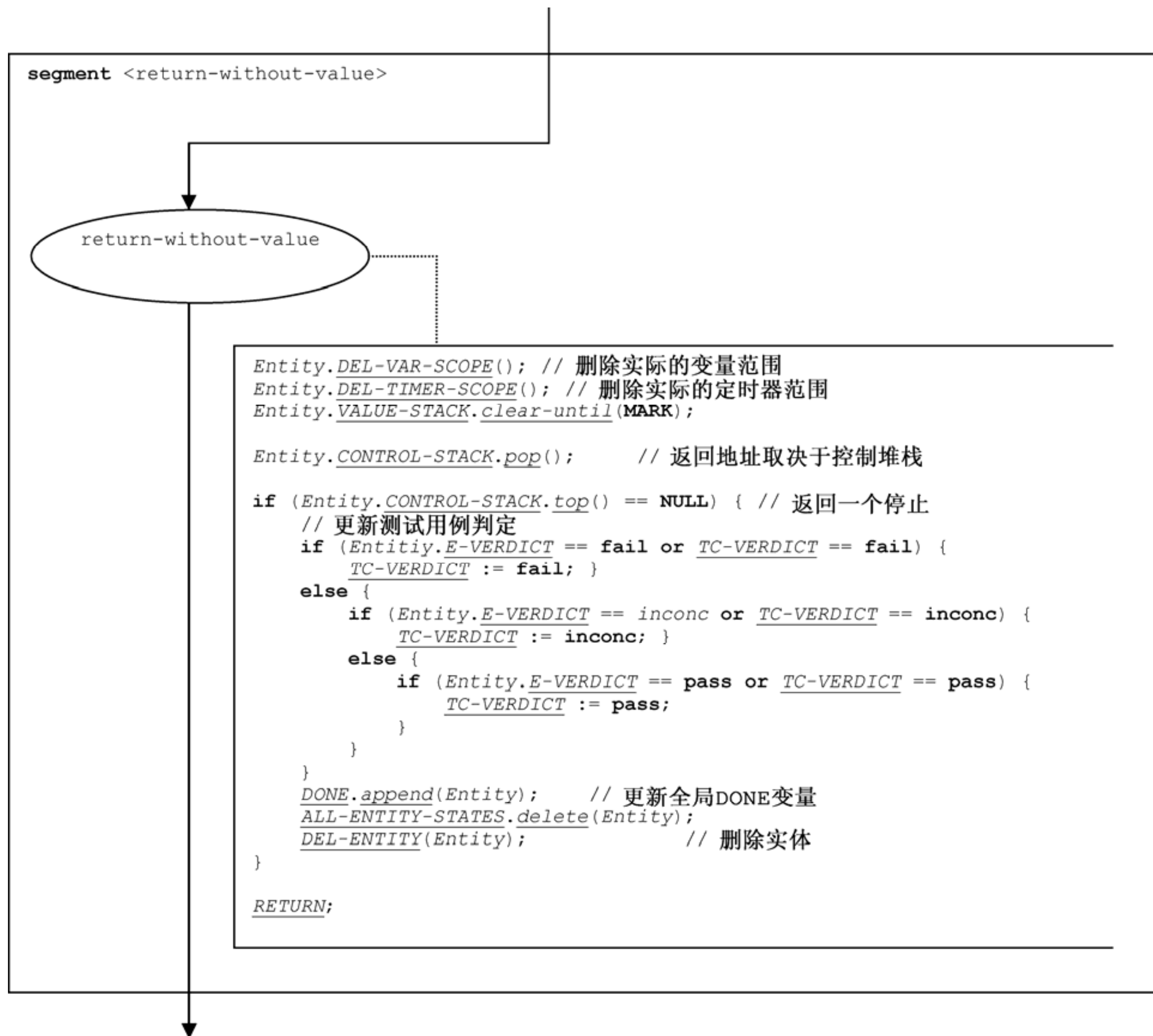


图 110/Z.143－流程图片段<return-without-value>

9.41 Running部件操作

running 部件操作的语法结构是：

```
<component-expression>.running
```

running 部件操作检查部件是正在运行还是已经停止。使用一个部件引用来确定要检查的部件，它可以以一个变量或值返回函数的形式来提供，也就是说，它是一个表达式。为简化起见，认为关键字'**all component**'和'**any component**'是特殊的表达式。

running 部件操作区分其在 **alt** 语句或阻塞 **call** 操作的布尔防卫中的用法与在所有其他情况中的用法之间的差别。如果用在布尔防卫中，那么 **running** 部件操作的结果基于实际的快照，而在所有其他情况下，则直接计算状态信息。

将 **running** 部件操作的结果推入实体的值堆栈，称之为操作。

图 111 中的流程图片段<running-component-op>对 **running** 部件操作的执行过程进行定义：

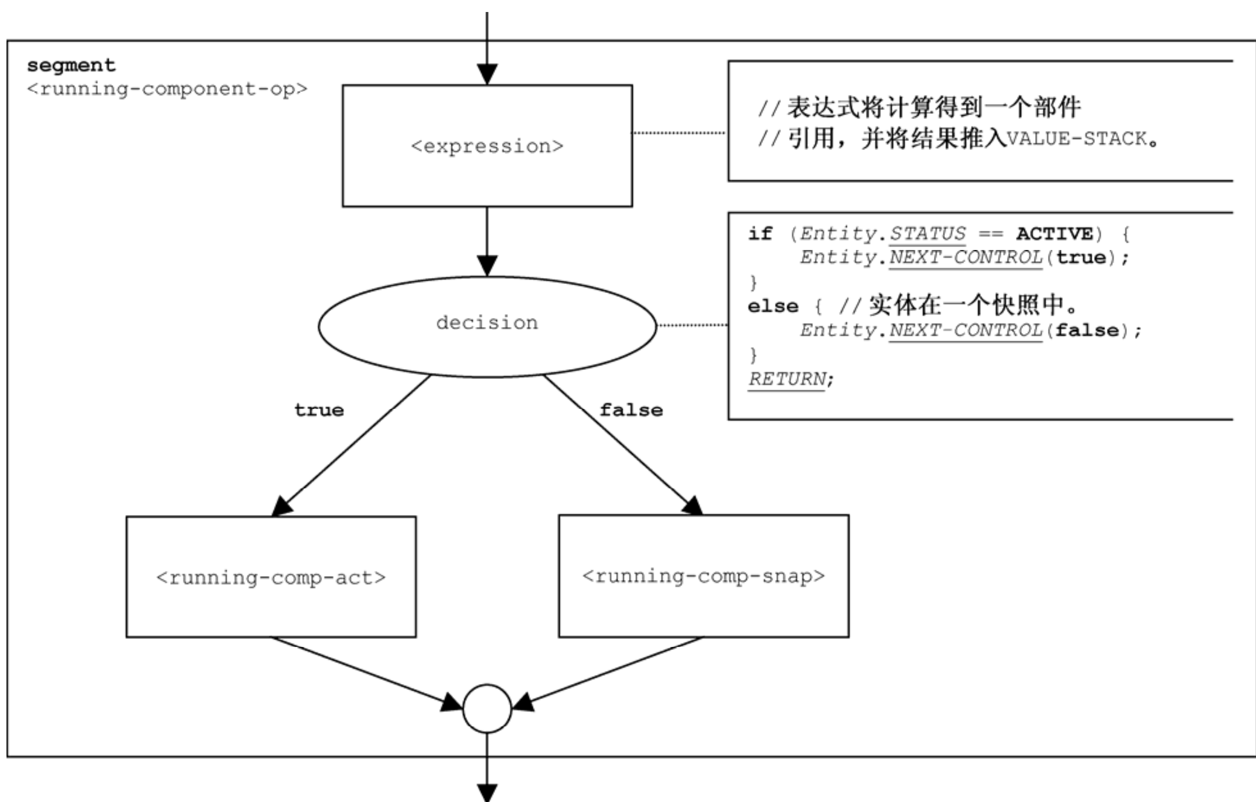


图 111/Z.143－流程图片段<running-component-op>

9.41.1 流程图片段<running-comp-act>

图 112 中的流程图片段<running-comp-act>对快照之外的 **running** 部件操作的执行过程进行定义，也就是说，实体处于 **ACTIVE** 状态。

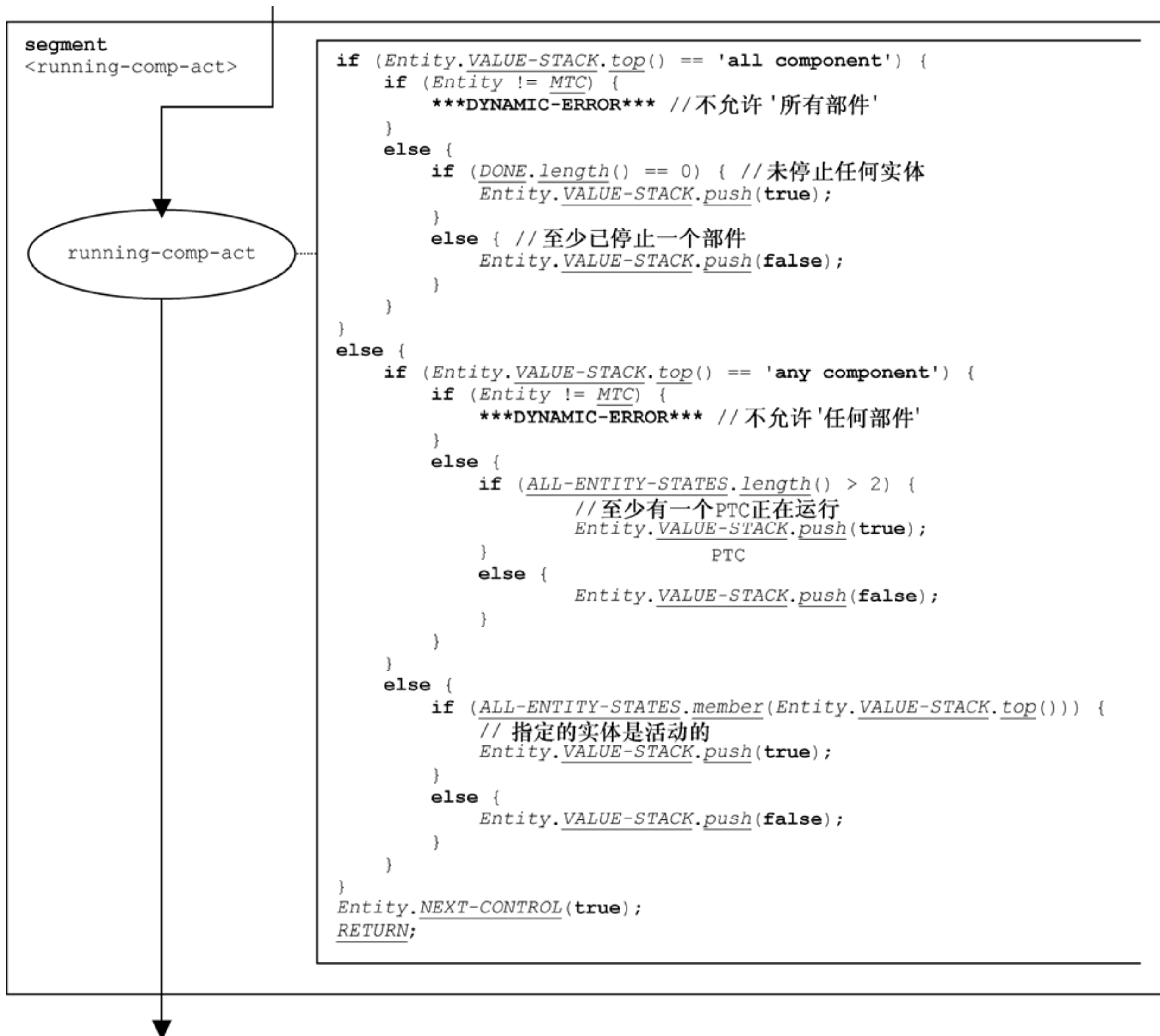


图 112/Z.143－流程图片段<running-comp-act>

9.41.2 流程图片段<running-comp-snap>

图 113 中的流程图片段<running-comp-snap>对快照计算期间的 **running** 部件操作的执行过程进行定义，也就是说，实体处于 **SNAPSHOT** 状态。

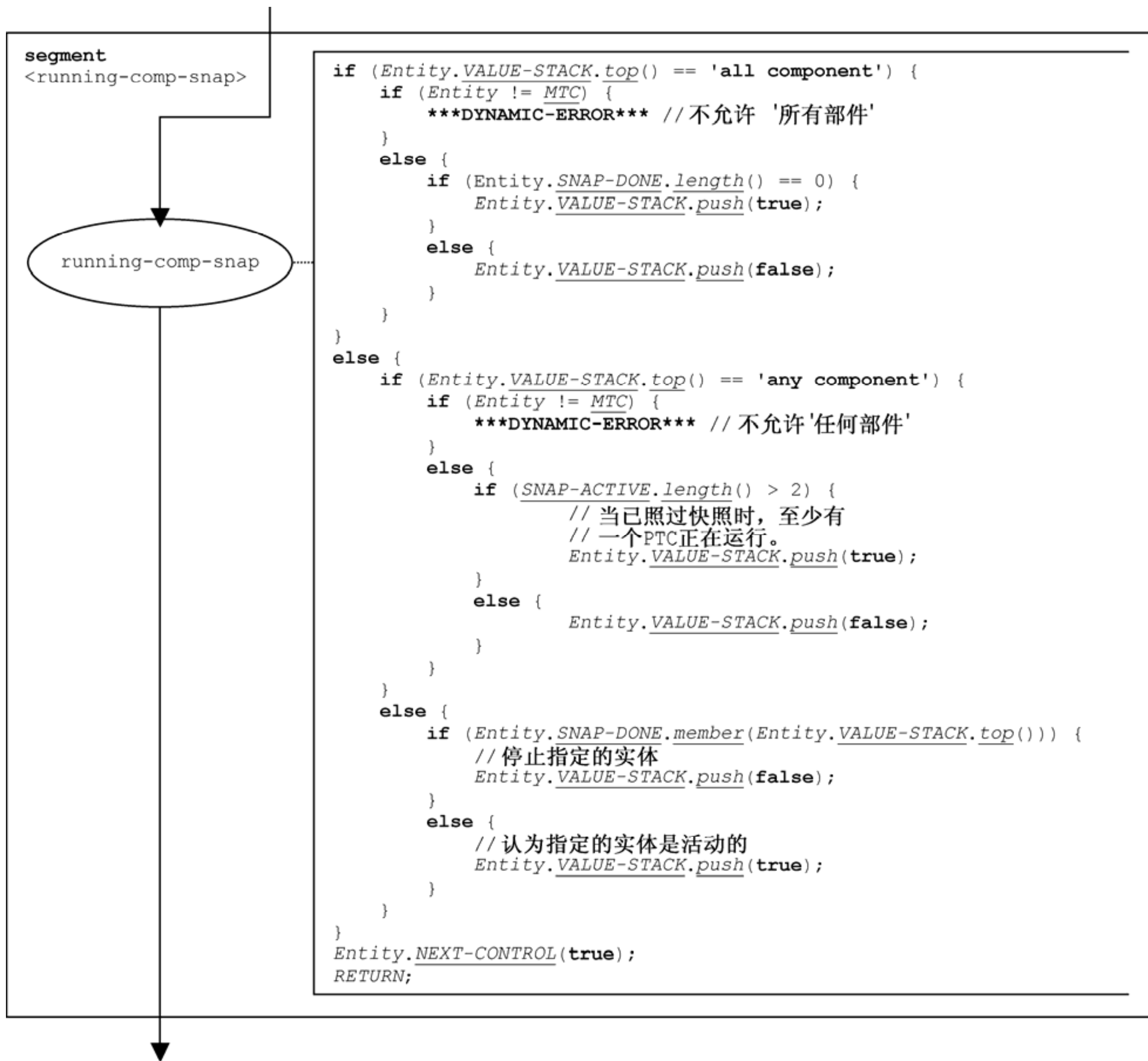


图 113/Z.143－流程图片段<running-comp-snap>

9.42 Running 定时器操作

running 定时器操作的语法结构是：

```
<timerId>.running
```

图 114 中的流程图片段<running-timer-op>对 **running** 定时器操作的执行过程进行定义。

Running 定时器操作区分其在 **alt** 语句或阻塞 **call** 操作的布尔防卫中的用法与在所有其它情况中的用法之间的差别。如果用在布尔防卫中，那么 **running** 定时器操作的结果基于实际的快照，即定时器绑定的 *SNAP-STATUS* 条目，而在所有其它情况下，为用于确定操作结果的、定时器绑定的 *STATUS* 条目。

关键字 **any** 当作 timerId 的一个特殊值来处理。

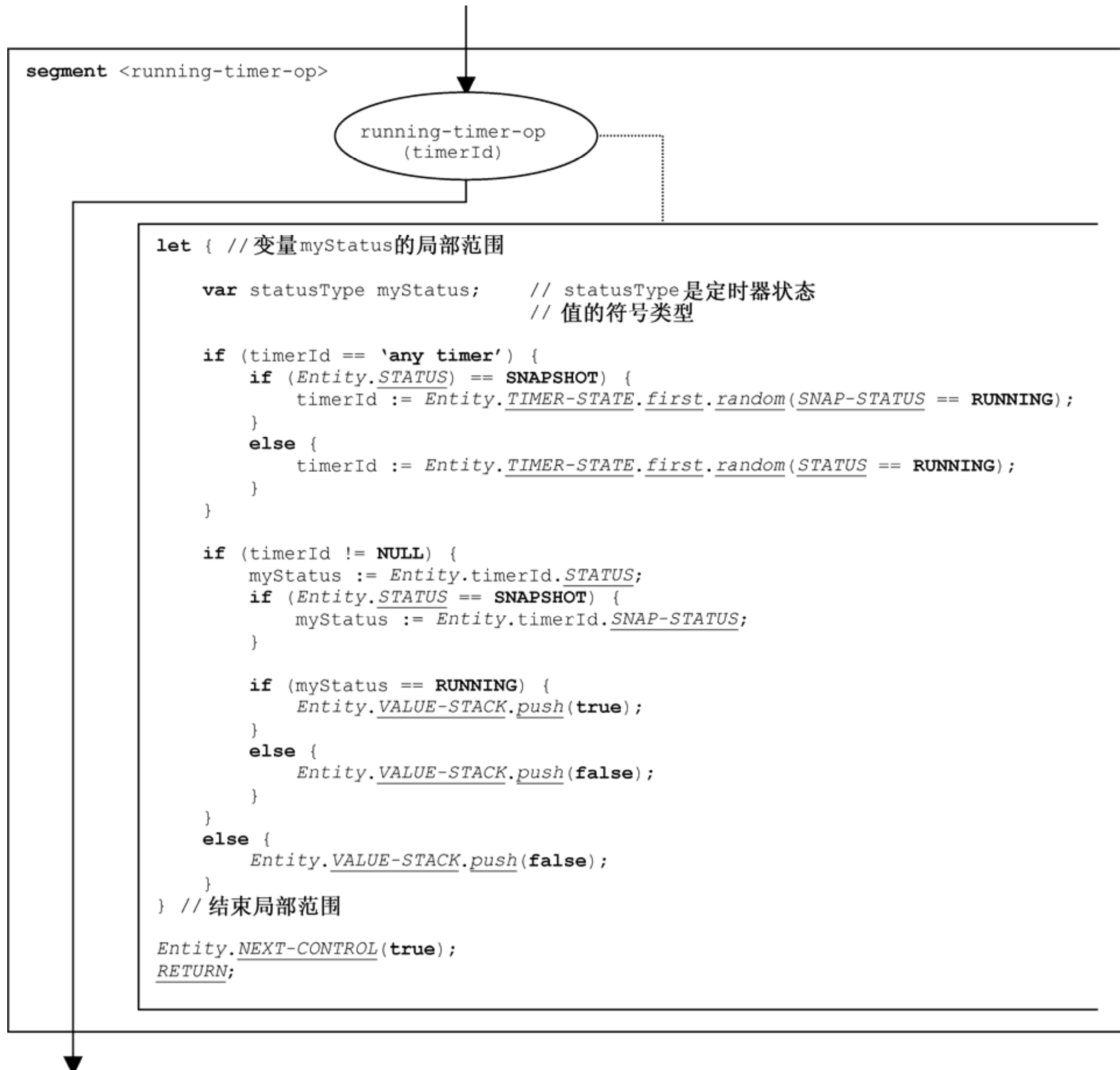


图 114/Z.143－流程图片段<running-timer-op>

9.43 Self 操作

self 操作的语法结构是：

```
self
```

图 115 中的流程图片段<self-op>对 **self** 操作的执行过程进行定义：

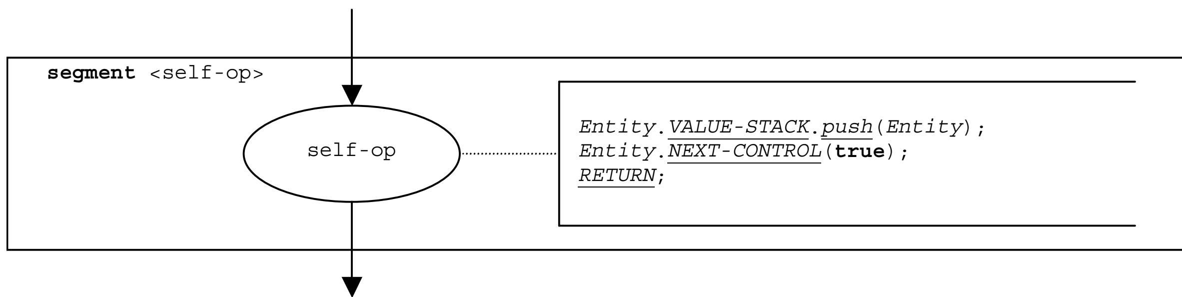


图 115/Z.143－流程图片段<self-op>

9.44 Send 操作

send 操作的语法结构是：

```
<portId>.send (<send-spec>) [to <component-expression>]
```

to 子句中的可选<component-expression>指的是接收方实体。它可以以一个变量值的形式来提供，也可以以一个函数返回值的形式来提供。

图 116 中的流程图片段<send-op>对 **send** 操作的执行过程进行定义：

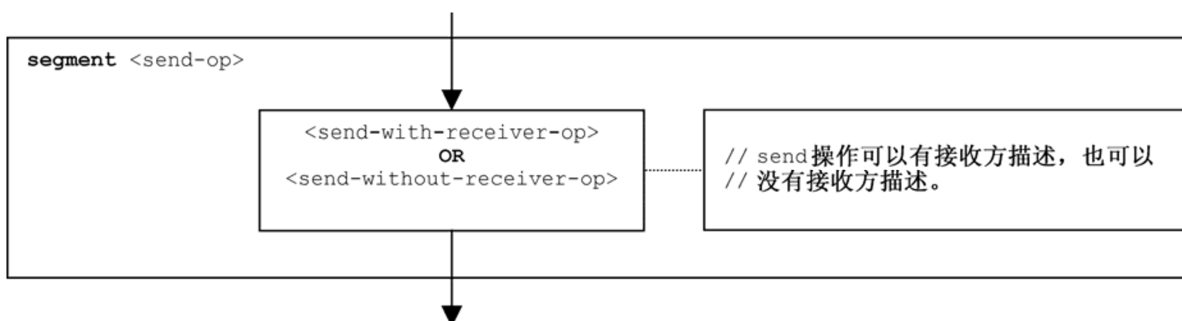


图 116/Z.143－流程图片段<send-op>

9.44.1 流程图片段<send-with-receiver-op>

图 117 中的流程图片段<send-with-receiver-op>对 **send** 操作的执行过程进行定义，当中的接收方以一个表达式的形式来描述。

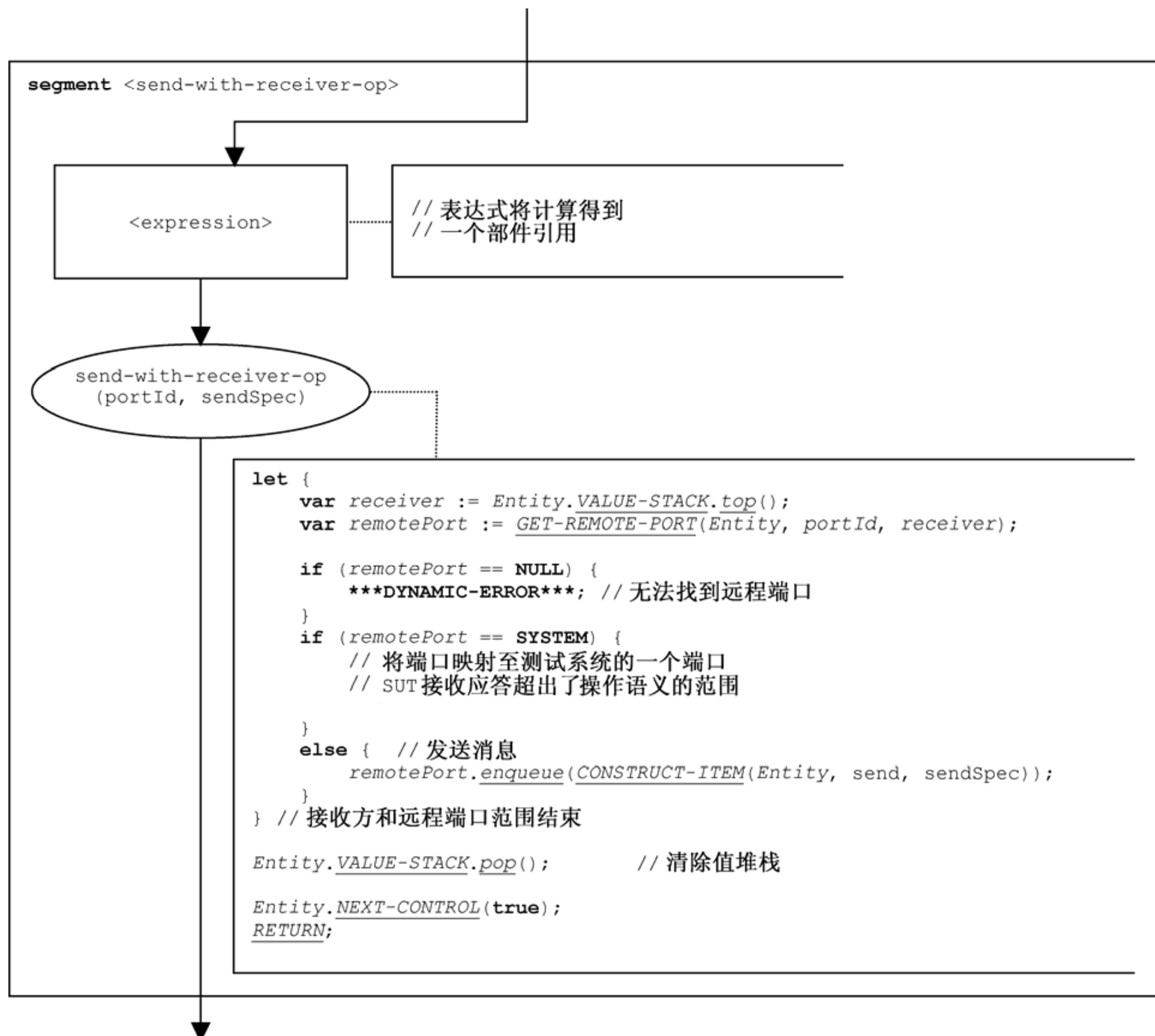


图 117/Z.143－流程图片段<send-with-receiver-op>

9.44.2 流程图片段<send-without-receiver-op>

图 118 中的流程图片段<send-without-receiver-op>对不带 **to** 子句的 **send** 操作的执行过程进行定义：

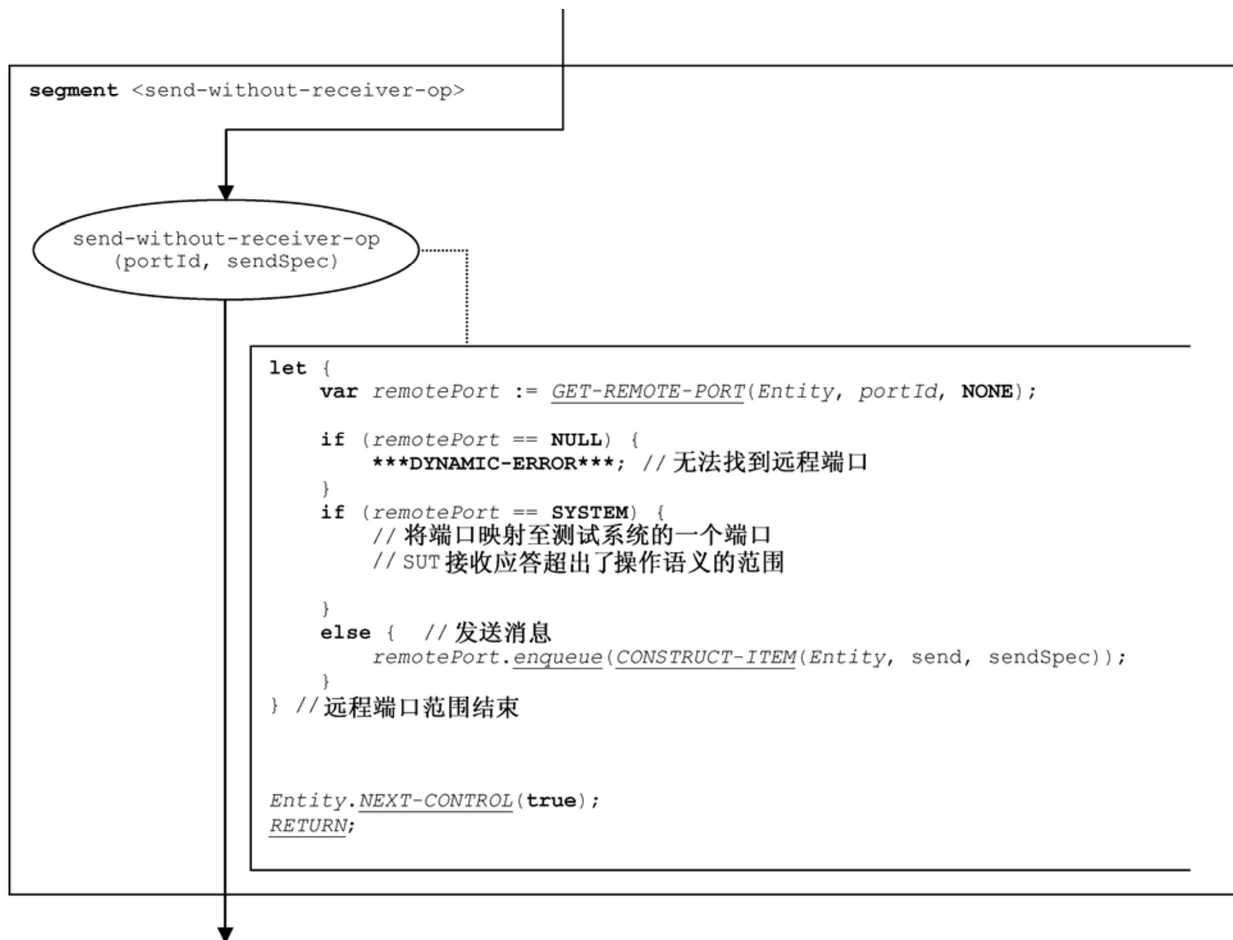


图 118/Z.143－流程图片段<send-without-receiver-op>

9.45 Setverdict 操作

setverdict 操作的语法结构是：

```
setverdict(<verdicttype-expression>)
```

setverdict 操作的<verdicttype-expression>参数是一个表达式，它将计算得到一个 **verdicttype** 类型的值，即 **none**、**pass**、**inconc** 或 **fail**。在应用 **setverdict** 操作之前对表达式进行计算。

图 119 中的流程图片段<setverdict-op>对 **setverdict** 操作的执行过程进行定义：

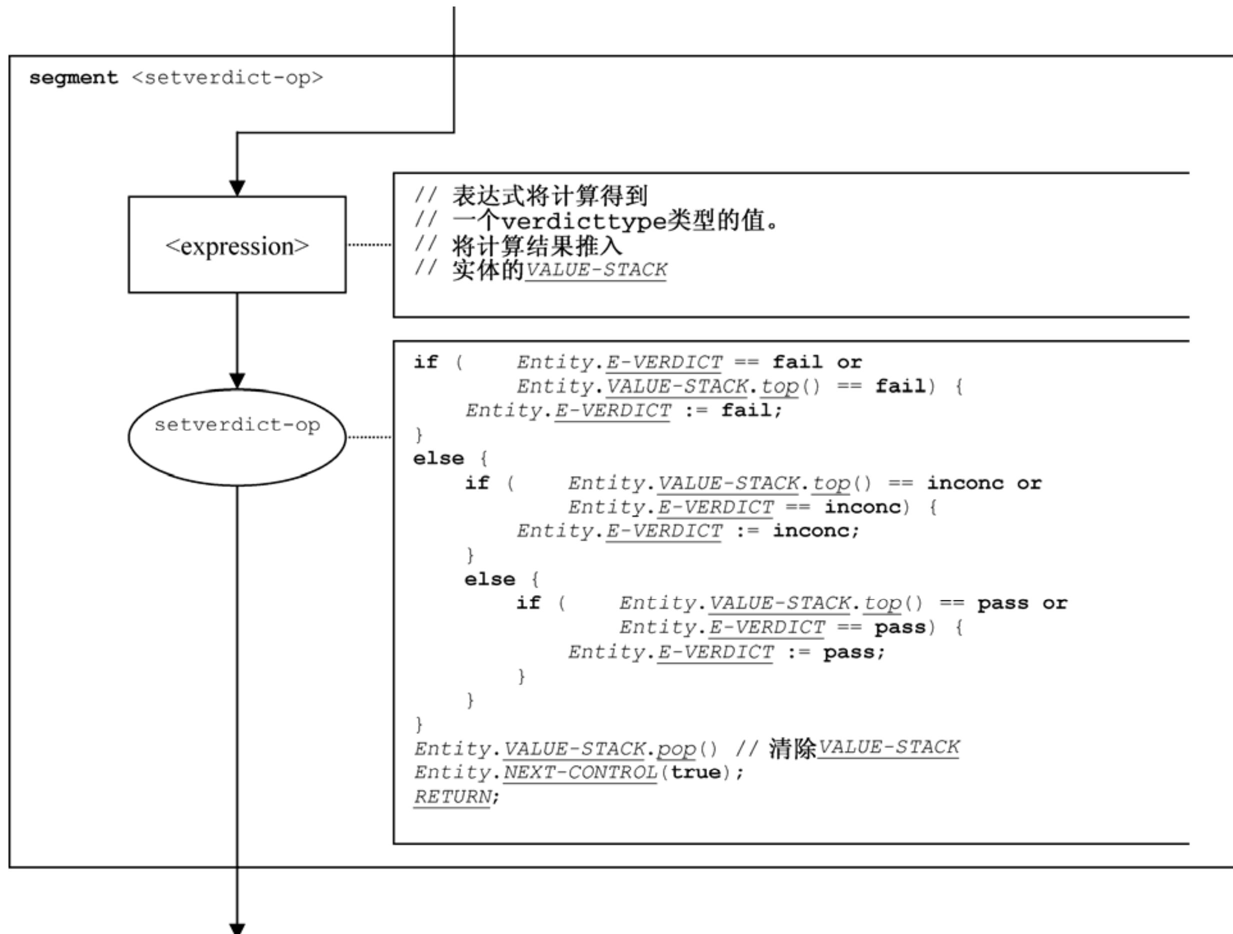


图 119/Z.143－流程图片段<setverdict-op>

9.46 Start部件操作

start 部件操作的语法结构是：

```
<component-expression>.start(<function-name>(<act-par-desc1>,..., <act-par-descn>))
```

start 部件操作开始一个新创建的部件。使用部件引用来确定要开始的部件。引用可以保存在一个变量中或由一个函数来返回，也就是说，它是一个表达式，将计算到一个部件引用。

<function-name>表示用于定义新部件行为的函数名称，<act-par-descr$_1$>,...,<act-par-descr$_n$>用于描述<function-name>的实际的参数值。在 **start** 部件操作引用的函数中，只允许值参数。对实参的描述以表达式的形式来提供，在执行调用之前，需要对表达式进行计算。对形式值参数和实际值参数的处理等同于函数调用中对它们的处理（见第 9.24 节）。

图 120 中的流程图片段<start-component-op>对 **start** 部件操作的执行过程进行定义。**start** 部件操作分四个步骤执行。在第一个步骤中，创建一个调用记录。在第二个步骤中，计算实际的参数值。在第三个步骤中，获取要开始的部件引用。在第四个步骤中，将控制和调用记录提供给新的部件。

注 — 图120中的流程图片段包括对引用参数（<ref-var-par-calc>）的处理。需要引用参数来解释测试用例的引用参数。操作语义假设由MTC来处理这些参数。

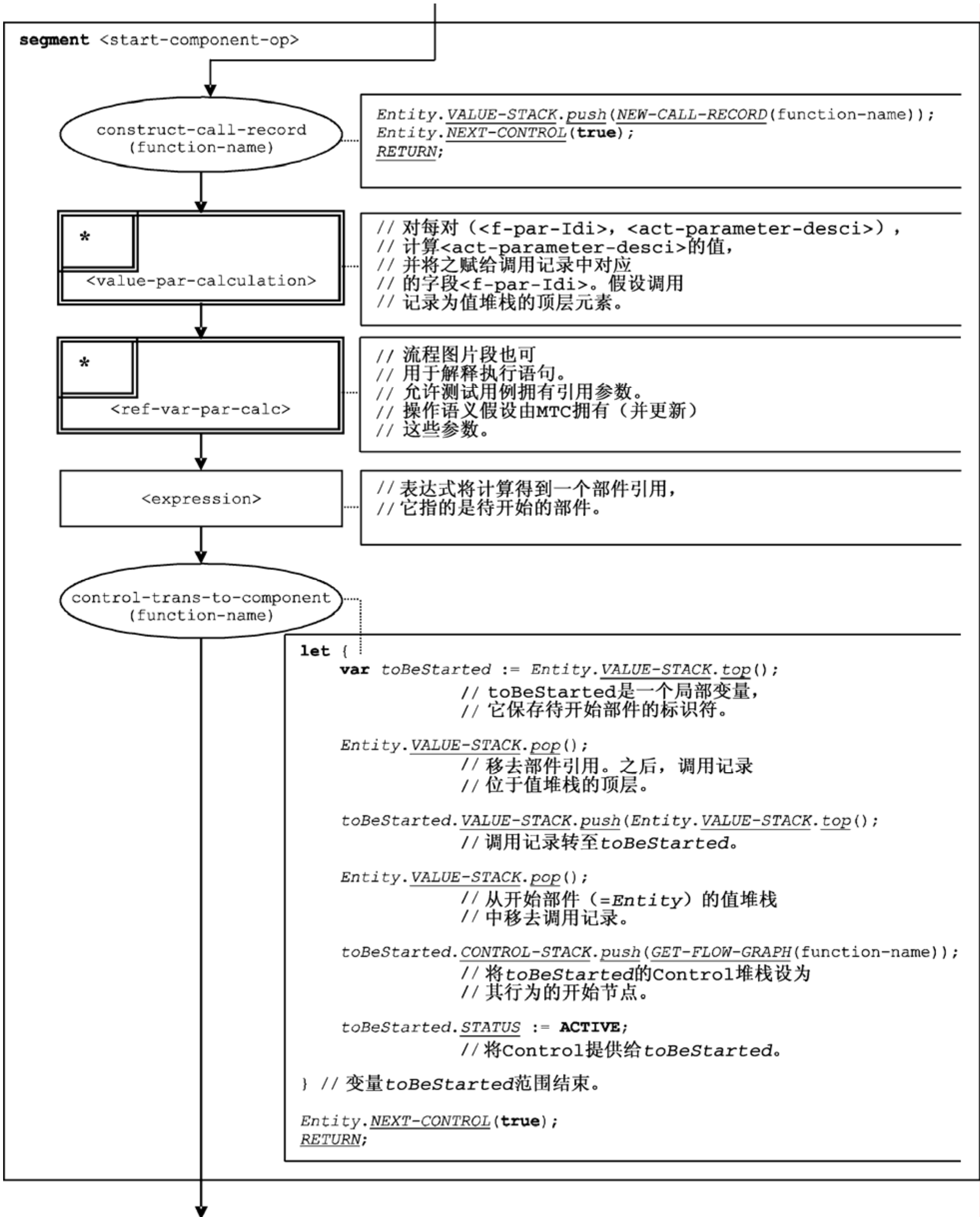


图 120/Z.143－流程图片段<start-component-op>

9.47 Start端口操作

start 端口操作的语法结构是：

```
<portId>.start
```

图 121 中的流程图片段<start-port-op>对 **start** 端口操作的执行过程进行定义：

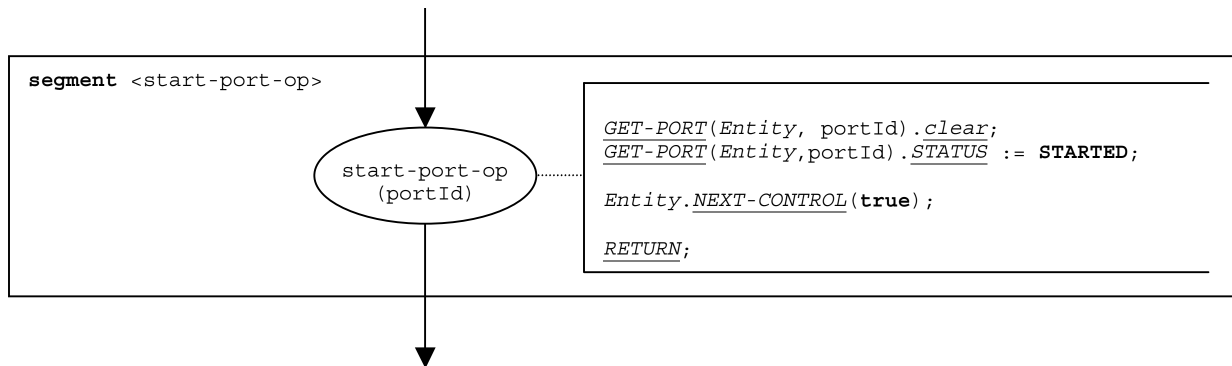


图 121/Z.143－流程图片段<start-port-op>

9.48 Start定时器操作

start 定时器操作的语法结构是：

```
<timerId>.start [(<float-expression>)]
```

定时器 **start** 操作的可选<float-expression>参数表示定时器实际的持续时间。如果它未提供，那么 **start** 操作将使用缺省的持续时间。表达式将计算得到一个 **float** 类型的值。如果它提供，那么在应用 **start** 操作之前对表达式进行计算。将计算结果推入实体的 *VALUE-STACK* 中。

图 122 中的流程图片段<start-timer-op>对 **start** 定时器操作的执行过程进行定义：

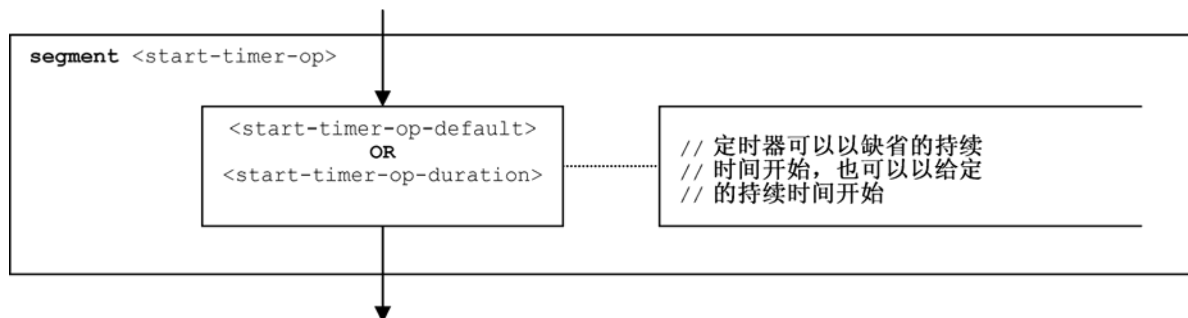


图 122/Z.143－流程图片段<start-timer-op>

9.48.1 流程图片段<start-timer-op-default>

图 123 中的流程图片段`<start-timer-op-default>`对带缺省值的 **start** 定时器操作的执行过程进行定义：

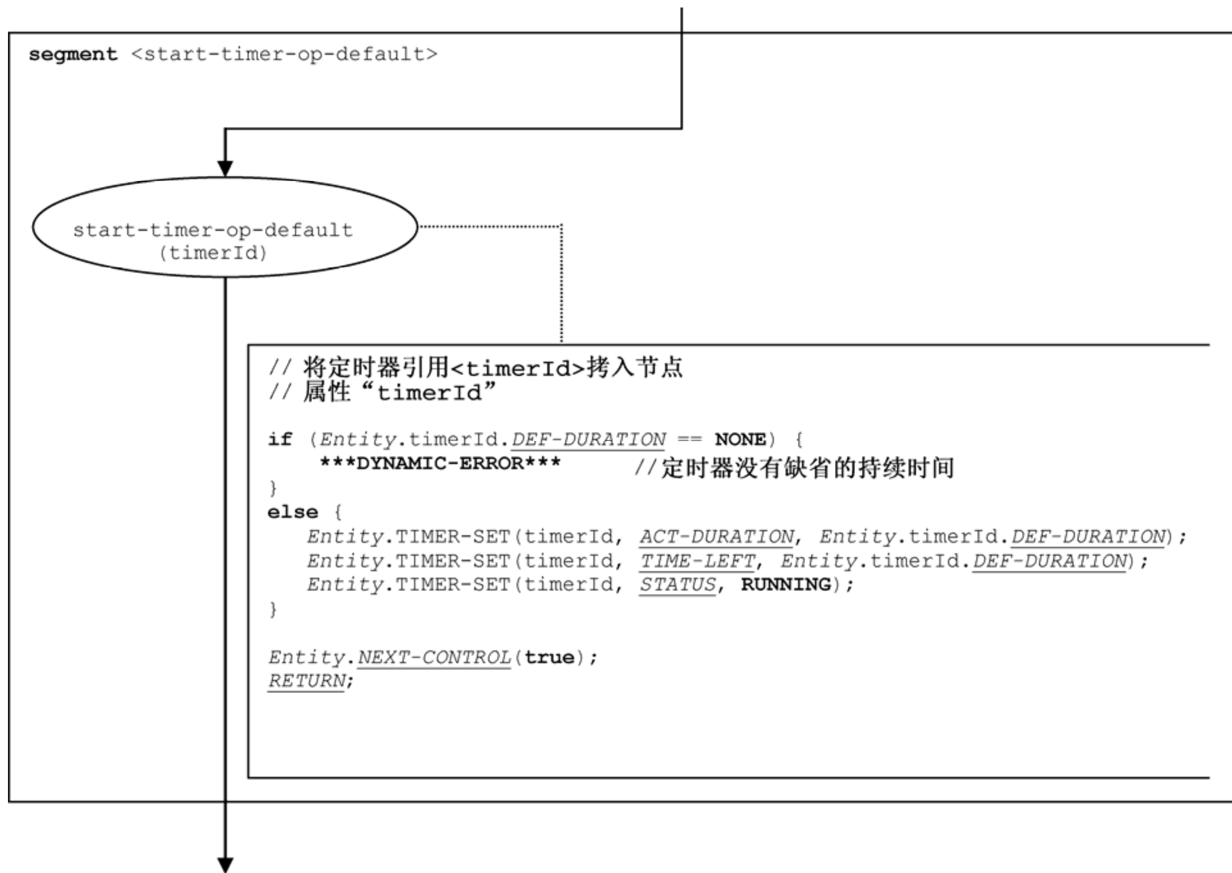


图 123/Z.143－流程图片段<start-timer-op-default>

9.48.2 流程图片段<start-timer-op-duration>

图 124 中的流程图片段<start-timer-op-duration>对带指定持续时间的 **start** 定时器操作的执行过程进行定义：

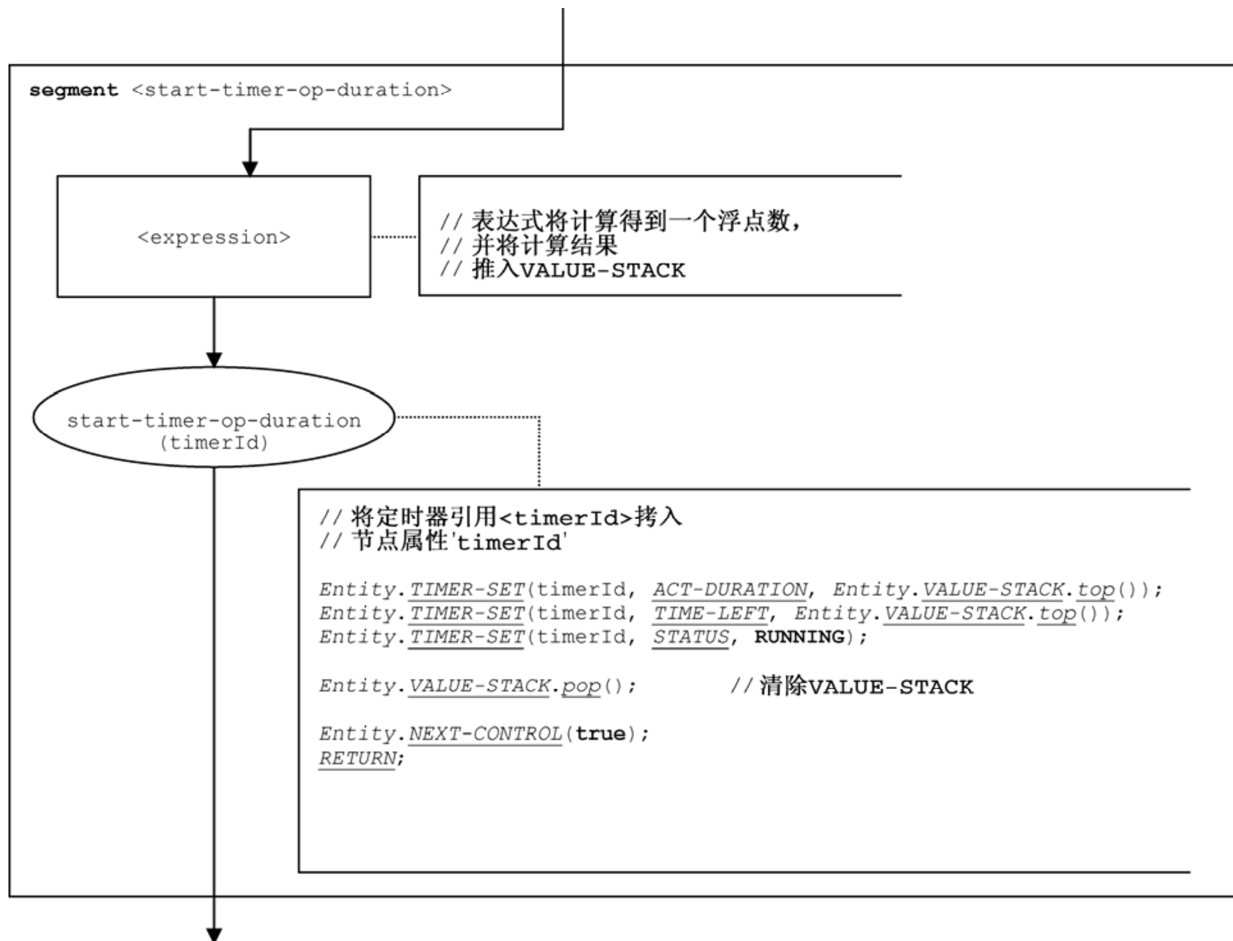


图 124/Z.143－流程图片段<start-timer-op-duration>

9.49 Stop部件操作

stop 部件操作的语法结构是：

<component-expression>.**stop**

stop 部件操作停止指定的部件。将停止所有的测试部件，也就是说，如果停止 MTC（如 **mtc.stop**）或停止其自身（如 **self.stop**），那么终止测试部件。通过使用关键字 **all**，即 **all component.stop**，MTC 可以停止所有的并行测试部件。

通过表达式指定的一个部件引用来确定要停止的部件，如一个值或值返回函数。为简化起见，认为关键字 '**all component**'是的<component-expression>特殊值。依据第 9.33 节和第 9.43 节对 **mtc** 和 **self** 操作进行计算。

图 125 中的流程图片段<stop-component-op>对 **stop** 部件操作的执行过程进行定义：

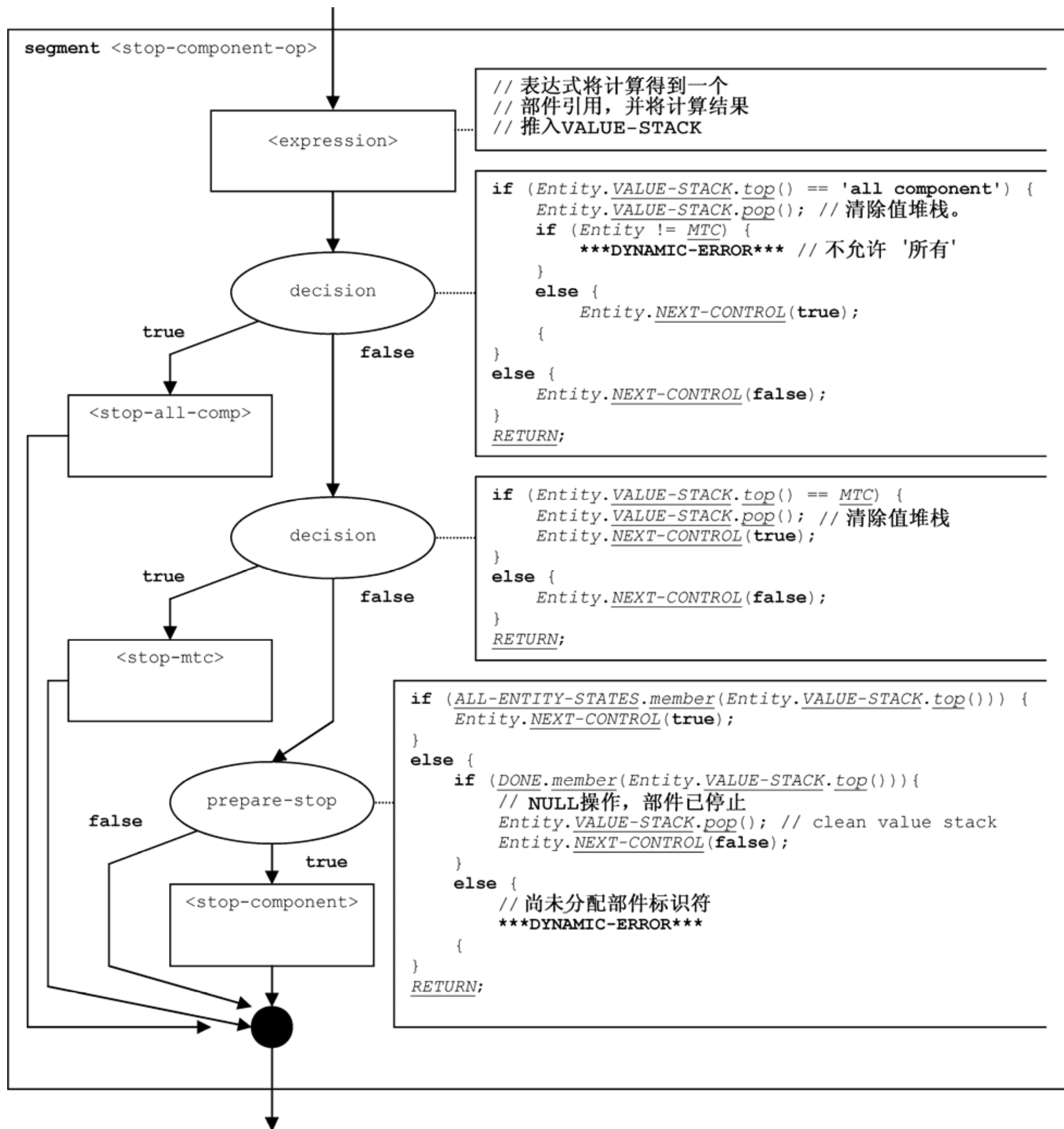


图 125/Z.143－流程图片段<stop-component-op>

9.49.1 流程图片段<stop-mtc>

图 126 中的流程图片段<stop-mtc>对测试用例的 MTC 的停止进行描述。结果是测试用例终止，也就是说，计算最终的判定，并将之推入模块控制的值堆栈，释放所有的资源，清空模块状态的 DONE 清单，终止包括 MTC 在内的所有测试部件。

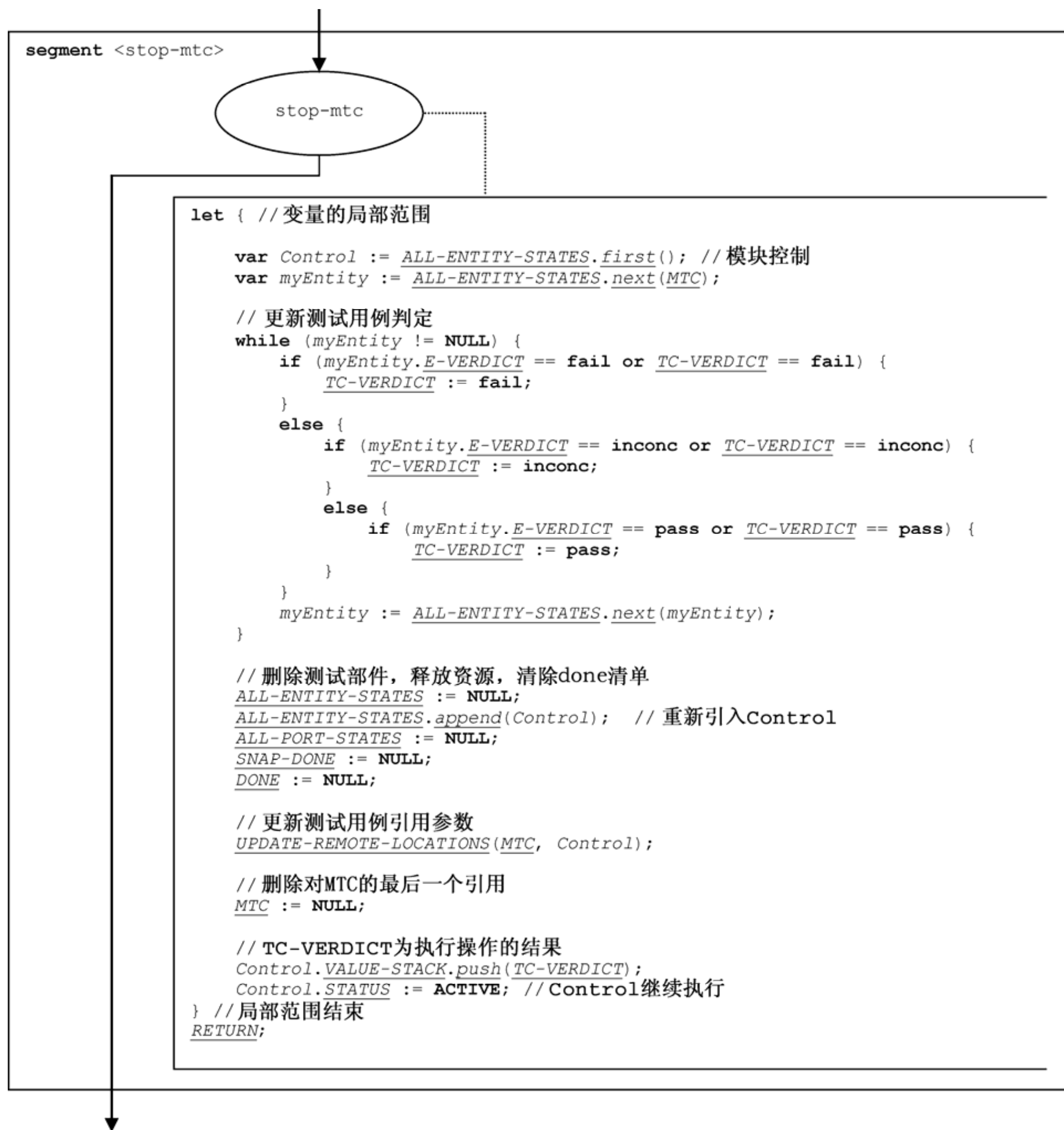


图 126/Z.143－流程图片段<stop-mtc-op>

9.49.2 流程图片段<stop-component>

图 127 中的流程图片段<stop-component>对并行测试部件的停止进行描述，也就是说，不是 MTC 或模块控制。结果是更新测试用例判定 *TC-VERDICT* 以及已终止测试部件（*DONE*）的清单，从模块状态删除该部件。<stop-component>流程图假设待停止部件的标识符位于执行该片段的部件值堆栈的顶层。

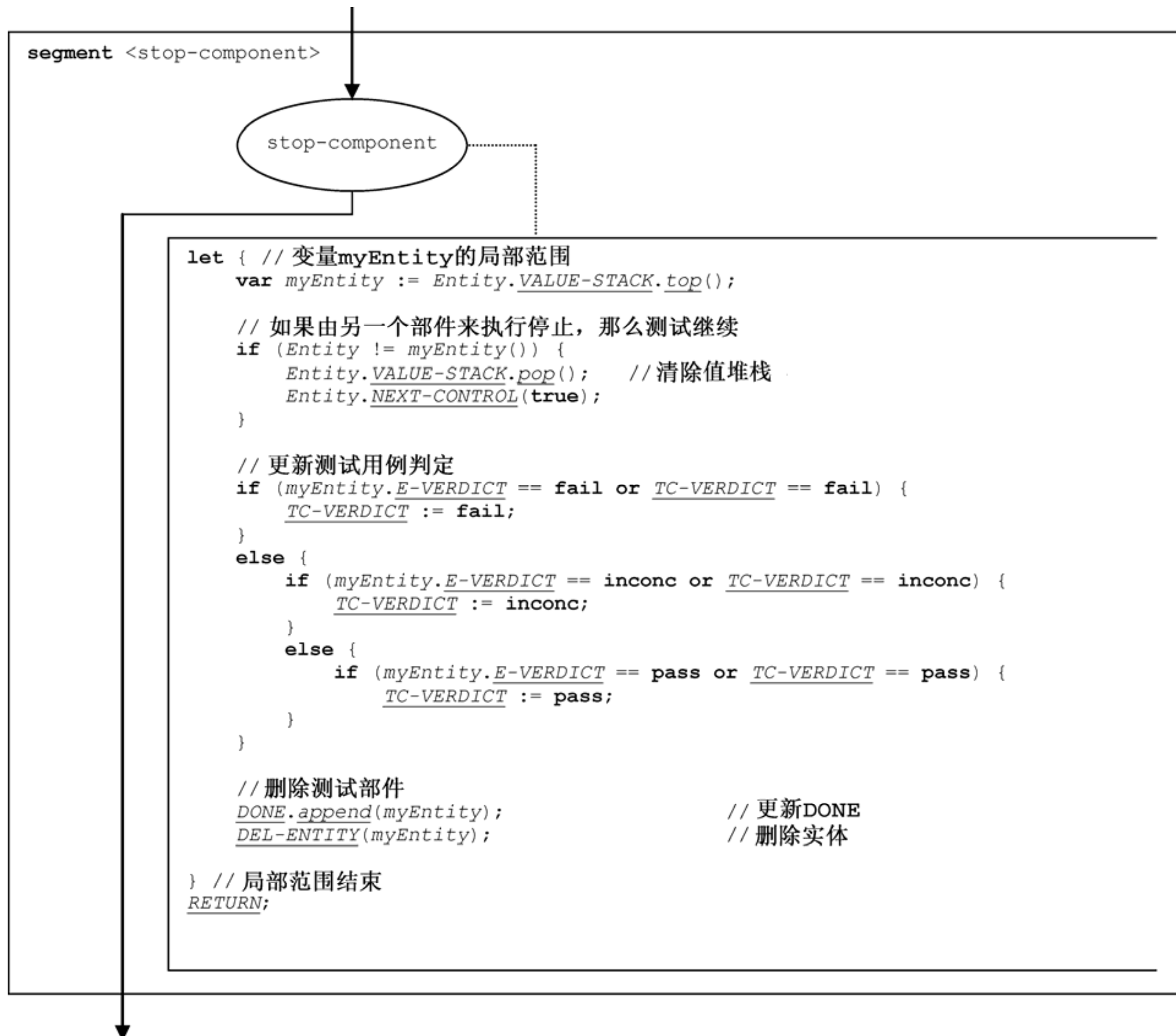


图 127/Z.143－流程图片段<stop-component>

9.49.3 流程图片段<stop-all-comp>

图 128 中的流程图片段`<stop-all-comp>`对测试用例所有并行测试部件的停止进行描述：

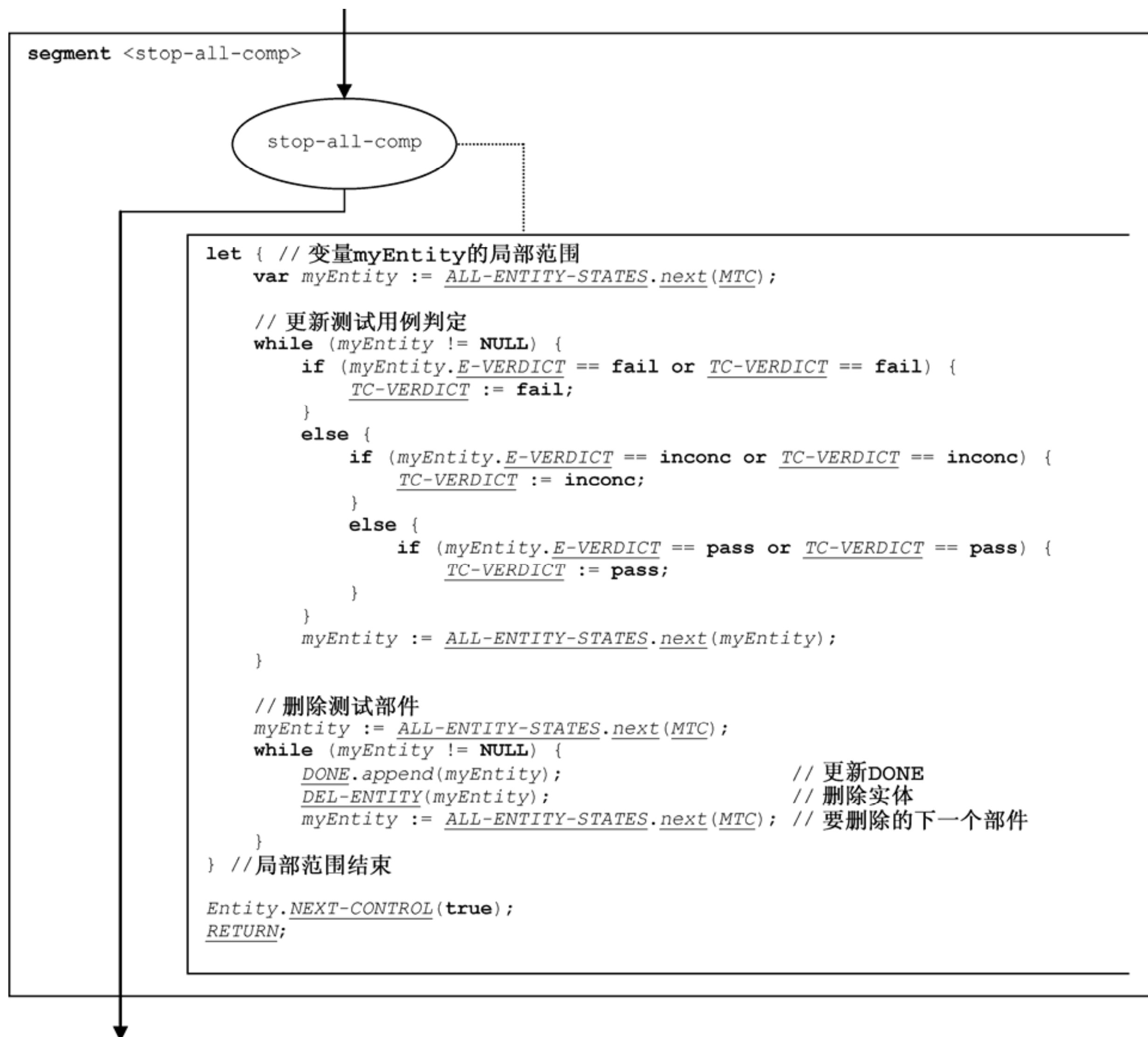


图 128/Z.143－流程图片段<stop-all-comp>

9.50 Stop执行语句

`stop` 执行语句的语法结构是：

```
stop
```

`stop` 执行语句的作用依赖于执行 **`stop`** 执行语句的实体：

a） 如果由模块控制来执行**`stop`**，那么测试过程结束，也就是说，所有测试部件和模块控制从模块状态消失。

b） 如果由MTC来执行**`stop`**，那么所有并行测试部件和MTC停止执行。对全局测试用例判定进行更新，并将之推入模块控制的值堆栈中。最后，将控制还给模块控制，MTC终止。

c） 如果由测试部件来执行**`stop`**，那么对全局测试用例判定*TC-VERDICT*和全局*DONE*进行更新。然后部件彻底从模块中消失。

图 129 中的流程图片段<stop-exec-stmt>对 **stop** 语句的执行过程进行描述：

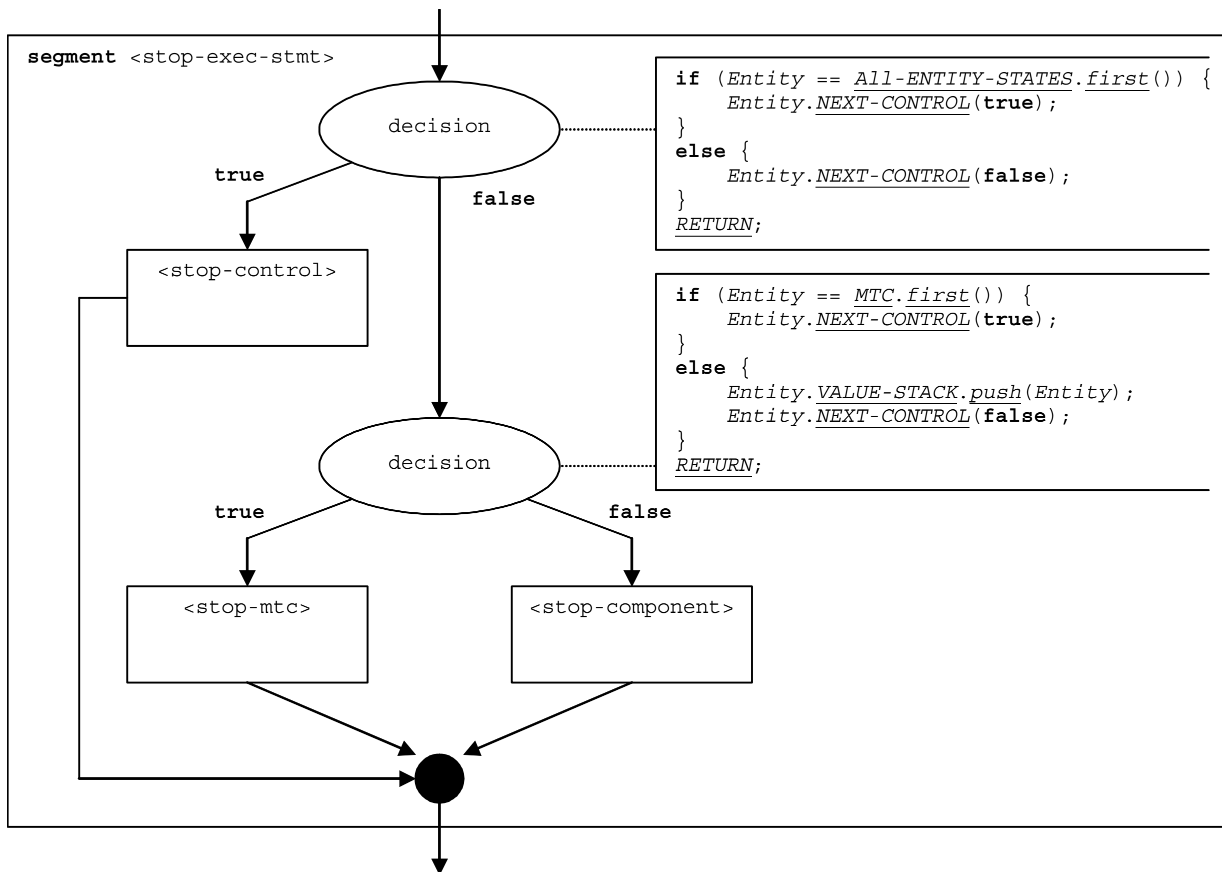


图 129/Z.143－流程图片段<stop-exec-stmt>

9.50.1 流程图片段<stop-control>

图 130 中的流程图片段<stop-control>描述模块控制的停止。作用是将 *ALL-ENTITY-STATES* 设为 **NULL**，也就是说，满足模块计算程序（见第 8.6 节）的终止条件。

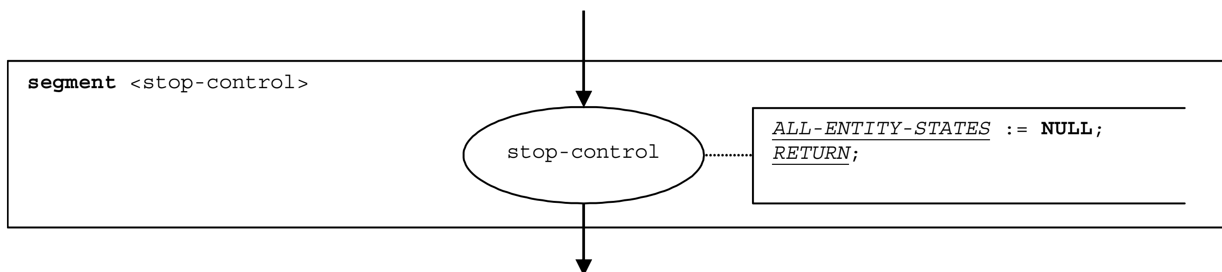


图 130/Z.143－流程图片段<stop-control>

9.51 Stop端口操作

stop 端口操作的语法结构是：

```
<portId>.stop
```

图 131 中的流程图片段<stop-port-op>对 **stop** 端口操作的执行过程进行定义：

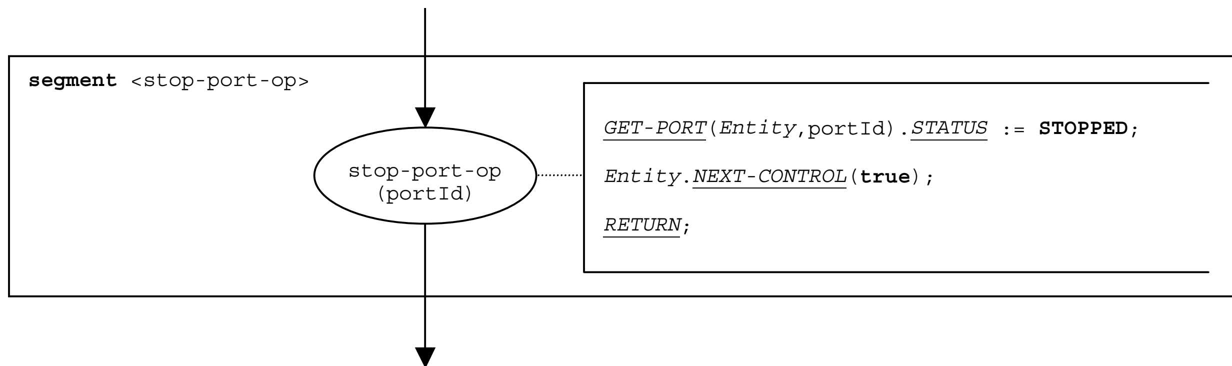


图 131/Z.143－流程图片段<stop-port-op>

9.52 Stop定时器操作

stop 定时器操作的语法结构是：

```
<timerId>.stop
```

图 132 中的流程图片段<stop-timer-op>对 **stop** 定时器操作的执行过程进行定义。

将关键字 **all** 作为 timerId 的一个特殊值来处理。

图 132/Z.143－流程图片段<stop-timer-op>

9.53 System 操作

system 操作的语法结构是：

```
system
```

图 133 中的流程图片段<system-op>对 **system** 操作的执行过程进行定义：

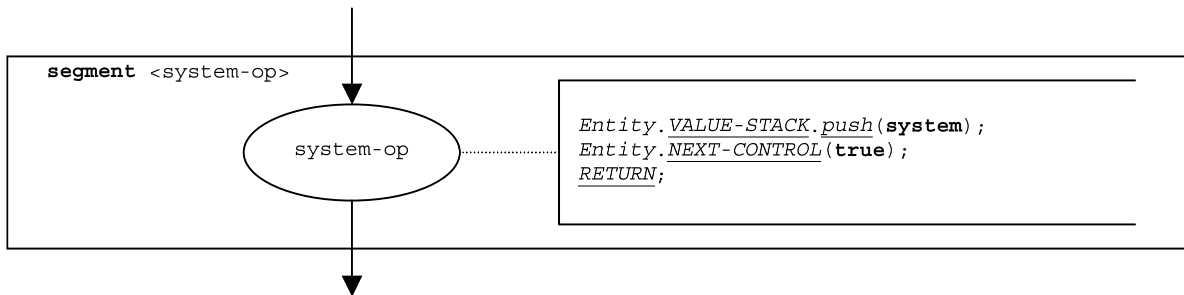


图 133/Z.143－流程图片段<system-op>

9.54 Timer 声明

timer 声明的语法结构是：

```
timer <timerId> [:= <float-expression>]
```

定时器声明的作用是创建一个新的定时器绑定。缺省持续时间的声明是可选的。认为缺省值是一个表达式，它计算得到一个 **float** 类型的值。

图 134 中的流程图片段<timer-declaration>对定时器声明的执行过程进行定义：

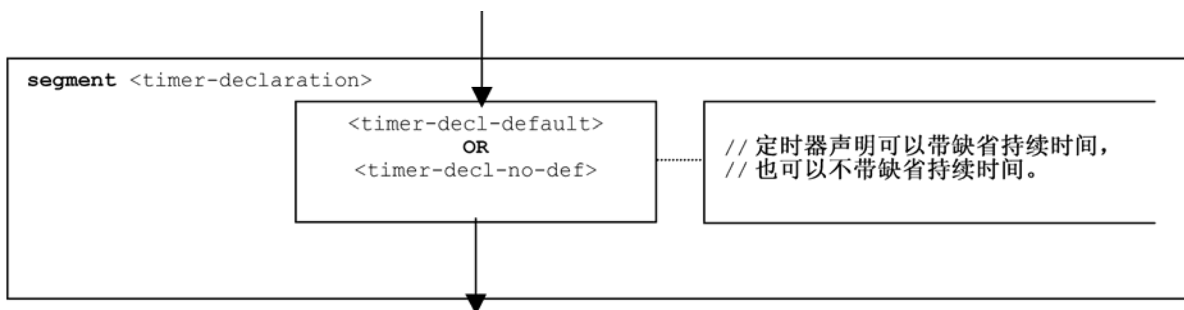


图 134/Z.143－流程图片段<timer-declaration>

9.54.1 流程图片段<timer-decl-default>

图 135 中的流程图片段`<timer-decl-default>`对以表达式形式提供了缺省持续时间的定时器声明的执行过程进行定义：

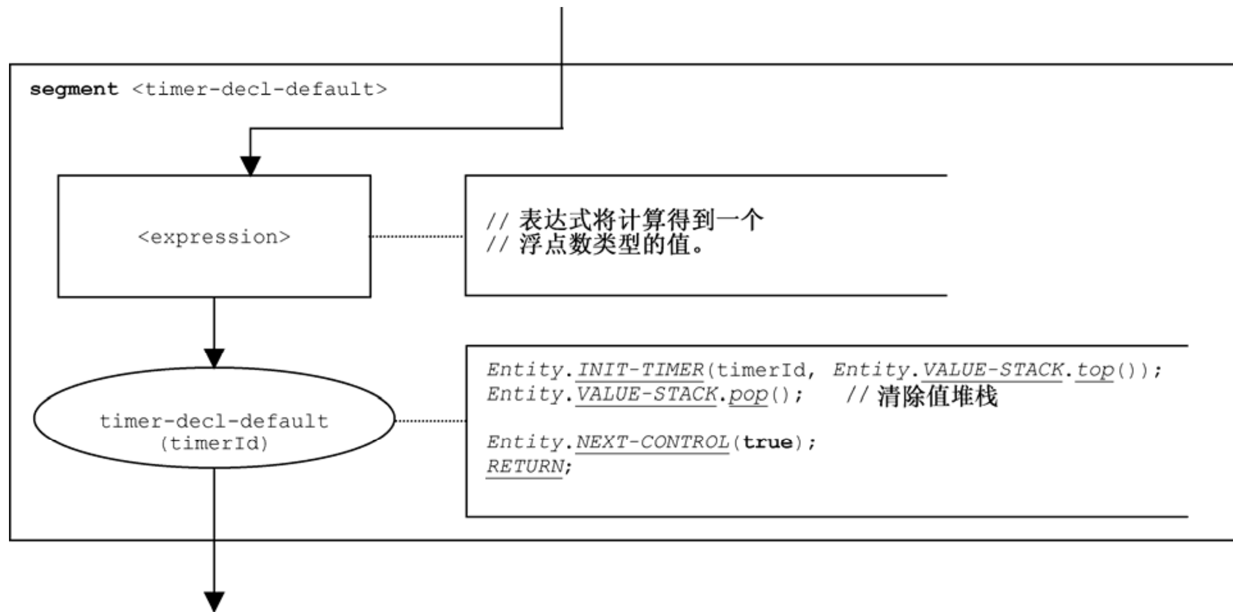


图 135/Z.143－流程图片段<timer-decl-default>

9.54.2 流程图片段<timer-decl-no-def>

图 136 中的流程图片段`<timer-decl-no-def>`对未提供缺省持续时间的定时器声明的执行过程进行定义，也就是说，未定义定时器的缺省持续时间。

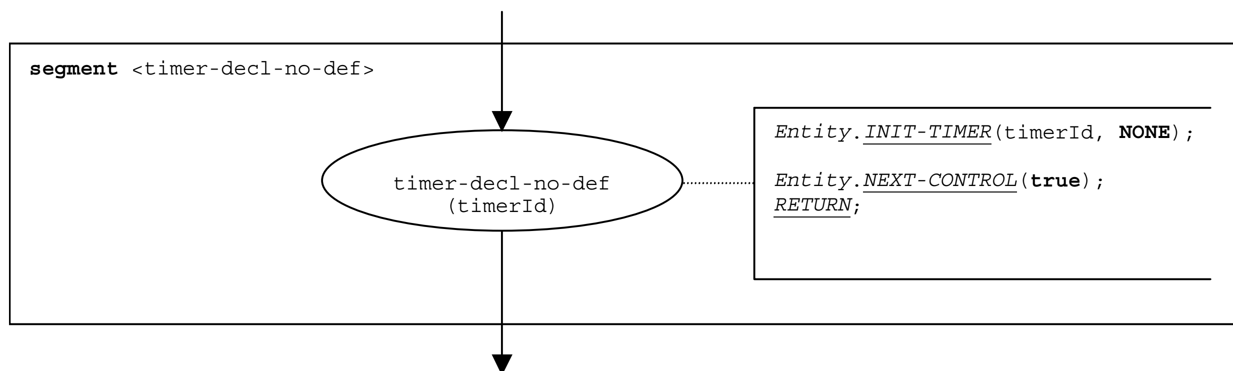


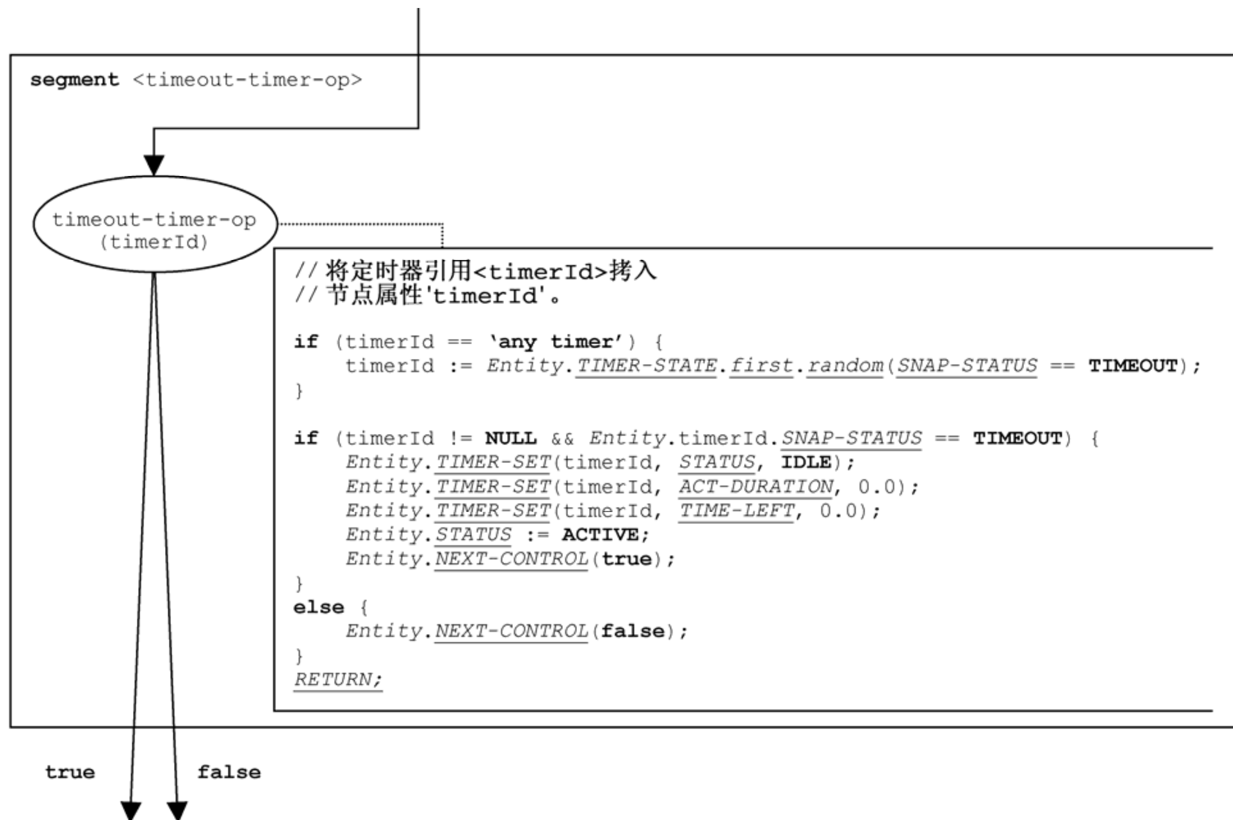
图 136/Z.143－流程图片段<timer-decl-no-def>

9.55 Timeout定时器操作

timeout 定时器操作的语法结构是：

```
<timerId>.timeout
```

图 137 中的流程图片段<timeout-timer-op>对 **timeout** 定时器操作的执行过程进行定义：



注1－timeout操作嵌入在一个alt语句中。其计算结果基于实际的快照，也就是说，依据定时器绑定中的*SNAP-STATUS*来做决定。如果timeout操作成功，也就是说*SNAP-STATUS* == TIMEOUT，那么定时器设为IDLE状态，并且部件状态从SNAPSHOT改为ACTIVE。

注2－当timeout计算得到true 或 false时，语句继续执行，后跟timeout操作（true分支），或者后跟待检查的alt语句中的下一个选项（false分支）。

注3－像对待timerId的特殊值一样来处理关键字any。

图 137/Z.143－流程图片段<timeout-timer-op>

9.56 Unmap 操作

unmap 操作的语法结构是：

```
unmap(<component_expression>:<portId1>,system:<portId2>)
```

认为标识符<portId1> 和 <portId2>是对应测试部件和测试系统接口的端口标识符。通过部件引用<component-expression>的方式来引用<portId1>所属的部件。引用可以保存在变量中或通过函数返回，也就是说，它是一个表达式，将计算得到一个部件引用。值堆栈用于保存部件引用。

注 — **unmap**操作不在乎**system**:<portId>语句是作为第一个参数还是第二个参数出现。为简化起见，假设它总为第二个参数。

图 138 中的流程图片段<unmap-op>对 **unmap** 操作的执行过程进行定义：

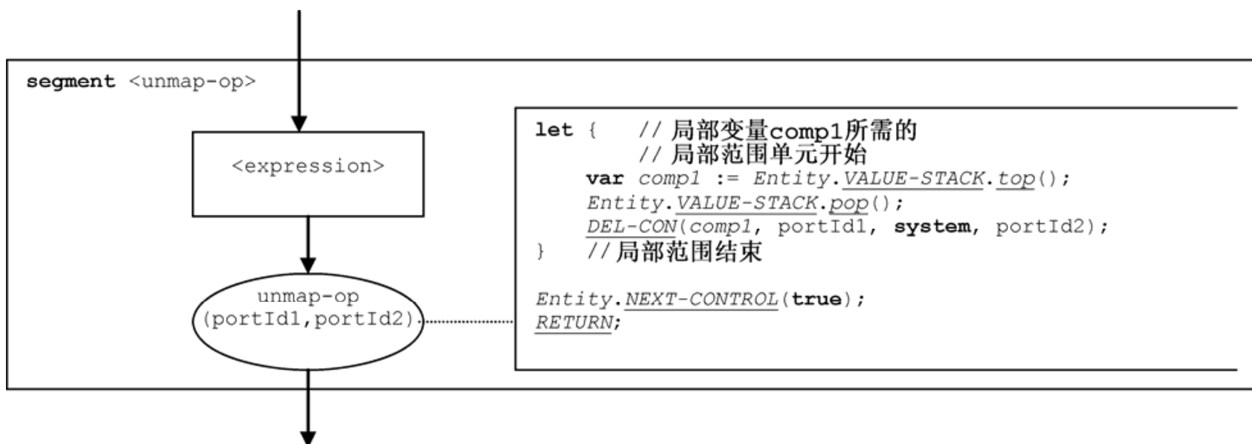


图 138/Z.143－流程图片段<unmap-op>

9.57 Variable 声明

Variable 声明的语法结构是：

```
var <varType> <varId> [:= <varType-expression>]
```

通过提供一个初始值（以表达式形式）来对一个变量进行初始化是可选的。认为初始值是一个表达式，它计算得到一个变量类型的值。

图 139 中的流程图片段<variable-declaration>对变量声明的执行过程进行定义：

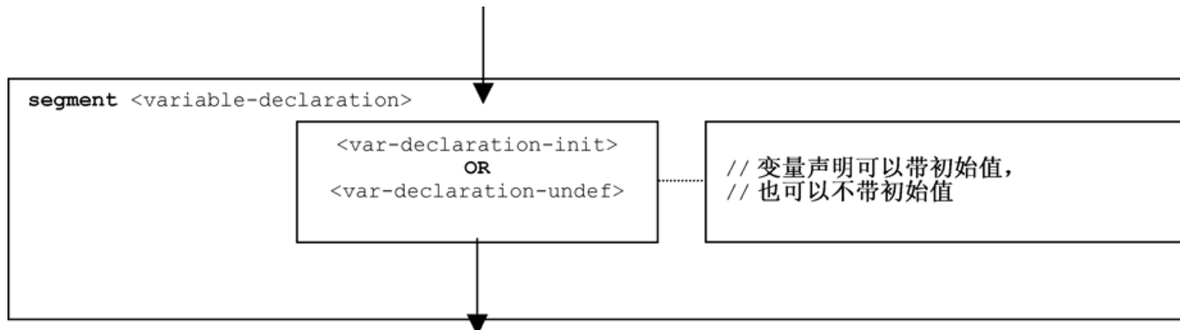


图 139/Z.143－流程图片段<variable-declaration>

9.57.1 流程图片段<var-declaration-init>

图 140 中的流程图片段<var-declaration-init>对变量声明的执行过程进行定义，其中的初始值以表达式的形式提供。

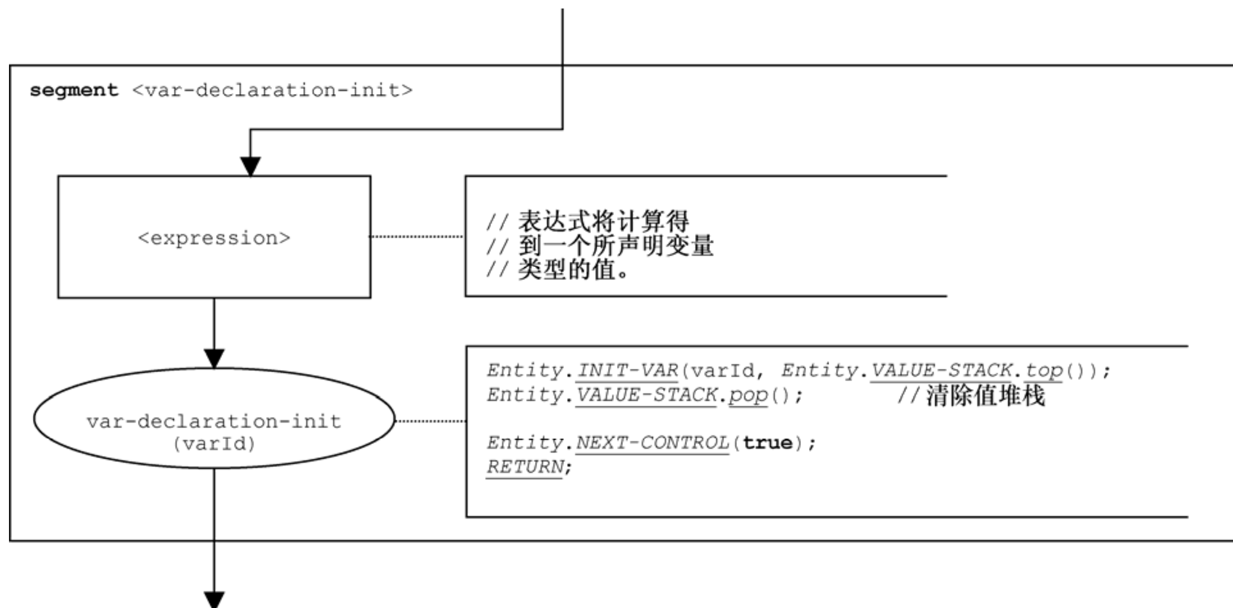


图 140/Z.143－流程图片段<var-declaration-init>

9.57.2 流程图片段<var-declaration-undef>

图 141 中的流程图片段<var-declaration-undef>对未提供初始值的变量声明的执行过程进行定义，也就是说，未定义变量的值。

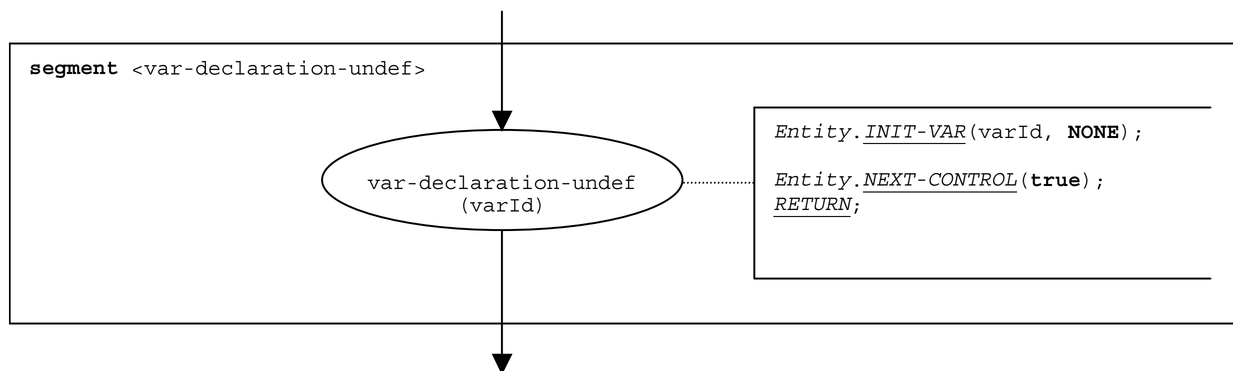


图 141/Z.143－流程图片段<var-declaration-undef>

9.58 While语句

while 语句的语法结构是：

```
while (<boolean-expression>) <statement-block>
```

通过图 142 中的流程图片段<while-stmt>对 **while** 语句的执行过程进行定义：

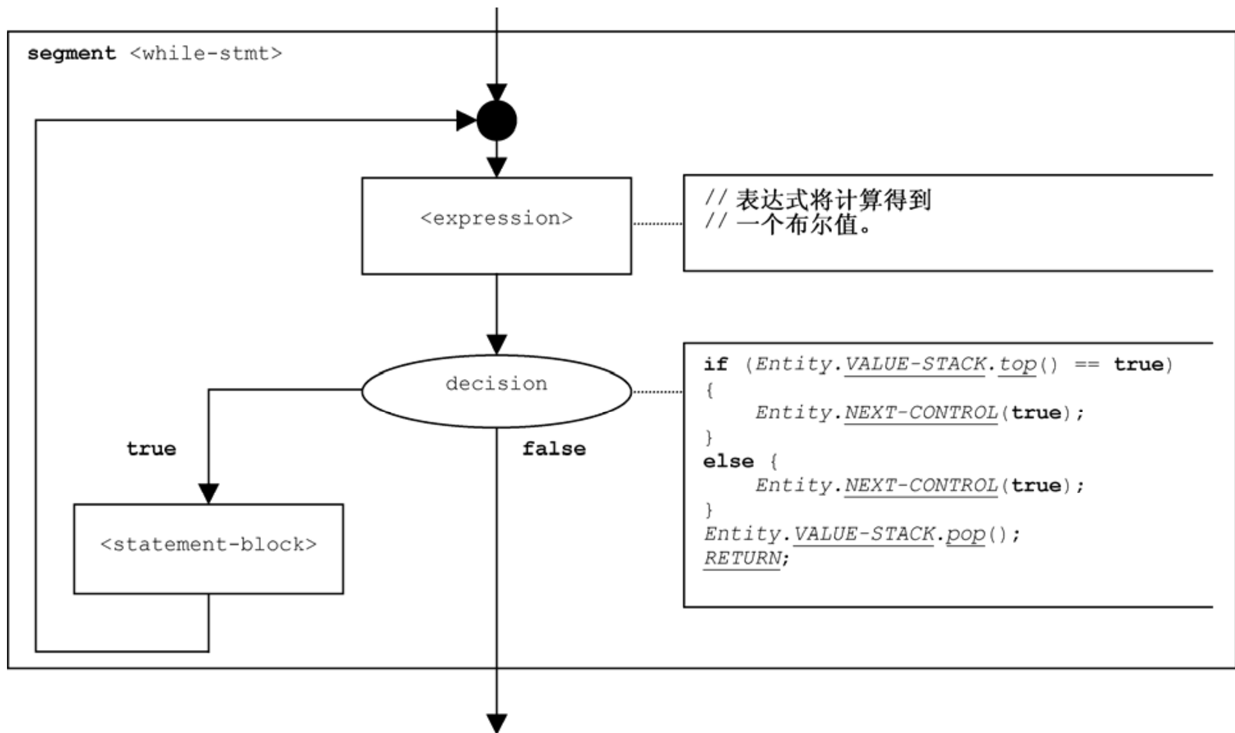


图 142/Z.143－流程图片段<while-stmt>

10 操作语义部件列表

10.1 功能和状态

名　称	描　述	节
ACT-DURATION	活动定时器处于启动状态的持续时间	8.3.2.4
增加	列表操作：增加一个条目，作为列表的第一个元素。	8.3.1.1
ADD-CON	增加一个到端口状态的连接	8.3.3.2
ALL-ENTITY-STATES	处于模块状态的部件状态	8.3.1
ALL-PORT-STATES	处于模块状态的端口状态	8.3.1
添加	列表操作：添加一个条目，作为列表的最后一个元素。	8.3.1.1
APPLY-OPERATOR	应用如＋、－ 或 /这样的运算符	8.6.2
修改	列表操作：修改列表的所有元素	8.3.1.1
清除	堆栈操作“清除”：清除一个堆栈	8.3.2.1
清除	队列操作“清除”：从一个队列中移去所有元素	8.3.3.2
清除－直至	堆栈操作“清除－直至”：弹出条目，直至特定条目为堆栈中的顶层元素	8.3.2.1
CONNECTIONS-LIST	列出一个端口的所有连接	8.3.3
CONSTRUCT-ITEM	构造一个待发送的条目	8.4.4
CONTINUE-COMPONENT	实际部件继续其执行过程	8.6.2
CONTROL-STACK	流程图节点堆栈，表示一个实体的实际控制状态	8.3.2
DATA-STATE	处于实体状态的数据状态	8.3.2

名　　称	描　　述	节
DEF-DURATION	定时器的缺省持续时间	8.3.2.4
DEFAULT-LIST	列出处于实体状态的所有活动的缺省状态	8.3.2
DEFAULT-POINTER	在计算缺省值期间指向实际的缺省值	8.3.2
DEL-CON	从一个端口状态删除一个连接	8.3.3.2
DEL-ENTITY	从一个模块状态删除一个实体	8.3.4
DEL-TIMER-SCOPE	删除一个定时器范围	8.3.2.5
DEL-VAR-SCOPE	删除一个变量范围	8.3.2.3
删除	列表操作：从一个列表删除一个条目	8.3.1.1
出列	队列操作：从一个队列删除第一个元素	8.3.3.2
DONE	已终止测试部件的标识符（模块状态的一部分）	8.3.1
E-VERDICT	测试部件的局部测试判定	8.3.2
入列	队列操作：将一个条目放入队列，作为最后元素。	8.3.3.2
第一个	队列操作“第一个”：返回队列的第一个元素	8.3.3.2
第一个	列表操作：返回列表的第一个元素	8.3.1.1
GET-FLOW-GRAPH	获取一个流程图的起始节点	8.2.7
GET-PORT	获取一个端口引用	8.3.3.2
GET-REMOTE-PORT	获取一个远程端口的引用	8.3.3.2
GET-TIMER-LOC	获取一个定时器的位置	8.3.2.5
GET-UNIQUE-ID	当调用时，返回一个新的、唯一的标识符。	8.6.2
GET-VAR-LOC	获取一个变量的位置	8.3.2.3
INIT-CALL-RECORD	在测试部件的实际范围单元，为基于过程的通信的参数初始化变量。	8.5.1
INIT-FLOW-GRAPHS	初始化流程图处理	8.6.2
INIT-TIMER	创建一个新的定时器绑定	8.3.2.5
INIT-TIMER-LOC	以一个现有的位置创建一个新的定时器绑定	8.3.2.5
INIT-TIMER-SCOPE	初始化一个新的定时器范围	8.3.2.5
INIT-VAR	创建一个新的变量绑定	8.3.2.3
INIT-VAR-LOC	以一个现有的位置创建一个新的变量绑定	8.3.2.3
INIT-VAR-SCOPE	初始化一个新的变量范围	8.3.2.3
长度	列表操作：返回一个列表的长度	8.3.1.1
M-CONTROL	处于模块状态的模块控制标识符	8.3.1
MATCH-ITEM	检查一个接收到的消息、调用、应答或异常是否匹配一个接收操作	8.4.5
成员	列表操作：检查一个条目是否为一个列表的元素	8.3.1.1
MTC	引用处于模块状态的 MTC	8.3.1
NEW-CALL-RECORD	为函数调用创建一个调用记录	8.5.1
NEW-ENTITY	创建一个新的实体状态	8.3.2.1
NEW-PORT	创建一个新的端口	8.3.3.2
NEXT	获取流程图中一个给定节点的后续节点	8.1.6
下一个	列表操作：返回列表中的下一个元素	8.3.1.1
NEXT-CONTROL	从控制堆栈弹出顶层流程图节点，并将下一个流程图节点推入控制堆栈。	8.3.2.1
OWNER	端口的所有者	8.3.3
弹出	堆栈操作“弹出”：从堆栈弹出一个条目	8.3.2.1
PORT-NAME	端口名称	8.3.3
推入	堆栈操作“推入”：将一个条目推入堆栈	8.3.2.1
随机	列表操作：随机返回列表的一个元素	8.3.1.1
REMOTE-ENTITY	处于端口状态的一个连接中的远程实体	8.3.3.1
REMOTE-PORT-NAME	处于端口状态的一个连接中的端口名称	8.3.3.1
RETRIEVE-INFO	从一个接收到的消息、调用、应答或异常中获取信息	8.4.6
RETURN	向模块估算过程返回控制	8.6.2

名　　称	描　　述	节
SNAP-ACTIVE	当 MTC 照快照（模块状态的一部分）时的活动测试部件编号	8.3.1
SNAP-DONE	当照快照时列出已终止的测试部件	8.3.2
SNAP-PORTS	提供快照功能，即更新 SNAP-VALUE。	8.3.3.2
SNAP-STATUS	一个定时器的快照状态	8.3.2.4
SNAP-TIMER	提供快照功能，并更新 SNAP-VALUE 和 SNAP-STATUS。	8.3.2.5
SNAP-VALUE	一个定时器的快照值	8.3.2.4
SNAP-VALUE	对快照语义，当照快照时予以更新。	8.3.3
STATUS	模块控制或一个测试部件的（**ACTIVE**、**SNAPSHOT**、**REPEAT** 或 **BLOCKED**）状态	8.3.2
STATUS	一个定时器的状态（**IDLE**、**RUNNING** 或 **TIMEOUT**）	8.3.2.4
STATUS	一个端口的状态（**STARTED** 或 **STOPPED**）	8.3.3
TC-VERDICT	处于模块状态的测试用例判定	8.3.1
TIME-LEFT	在运行定时器到期之前，对留下来用于运行的运行定时器进行计时	8.3.2.4
TIMER-GUARD	用于防卫 `execute` 语句和 `call` 操作的定时器	8.3.2
TIMER-NAME	一个定时器的名称	8.3.2.4
TIMER-SET	设置一个定时器的值	8.3.2.5
TIMER-STATE	处于实体状态的定时器状态	8.3.2
顶层	堆栈操作"顶层"：从一个堆栈返回顶层条目	8.3.2.1
UPDATE-REMOTE-REFERENCES	以不同实体中的相同位置将定时器和变量更新为相同的值	8.3.4
VALUE	一个变量的值	8.3.2.2
VALUE-QUEUE	端口队列	8.3.3
VALUE-STACK	用于保存表达式、操作数、操作和函数结果的堆栈值	8.3.2
VAR-NAME	一个变量的名称	8.3.2.2
VAR-SET	设置一个变量的值	8.3.2.3
动态错误	描述一个动态错误的发生	8.6.2
<标识符>	一个测试部件的唯一标识符	8.3.2
<位置>	支持范围单元、引用和定时器参数。表示定时器和变量的保存位置。	8.3.2.2, 8.3.2.4

10.2 特殊关键字

关　键　字	描　　述	节
ACTIVE	实体状态的 *STATUS*	8.3.2
BLOCKED	实体状态的 *STATUS*	8.3.2
IDLE	定时器状态的 *STATUS*	8.3.2.4
MARK	用作 *VALUE-STACK* 的标记	8.3.2
NONE	用于描述一个未定义的值	8.3.2.3, 8.3.2.5, 8.3.3.2
NULL	指针和指针类类型的符号值，指示未对任何寻址。	8.3.1.1, 8.3.2.1, 8.3.3, 8.3.3.2, 8.6.1.1
REPEAT	实体状态的 *STATUS*	8.3.2
RUNNING	定时器状态的 *STATUS*	8.3.2.4
SNAPSHOT	实体状态的 *STATUS*	8.3.2
STARTED	端口的 *STATUS*	8.3.3
STOPPED	端口的 *STATUS*	8.3.3
TIMEOUT	定时器状态的 *STATUS*	8.3.2.4

10.3 TTCN-3行为描述的流程图

	引　用	
	图	节
模块控制	18	8.2.2
测试用例	19	8.2.3
函数	20	8.2.4
可选步骤	21	8.2.5
部件类型定义	22	8.2.6

10.4 流程图片段

标　识　符	相关的TTCN-3构件	引　用	
		图	节
<action-stmt>	**action** 语句	36	9.1
<activate-stmt>	**activate** 语句	37	9.2
<alt-stmt>	**alt** 语句	38	9.3
<altstep-call>	调用一个可选步骤	44	9.4
<altstep-call-branch>	**alt** 语句	41	9.3.3
<assignment-stmt>	赋值	45	9.5
<b-call-with-duration>	**call** 操作	52	9.6.4
<b-call-without-duration>	**call** 操作	51	9.6.3
<blocking-call-op>	**call** 操作	47	9.6
<call-op>	**call** 操作	46	9.6
<call-reception-part>	**call** 操作	53	9.6.5
<catch-op>	**catch** 操作	55	9.7
<catch-timeout-exception>	**call** 操作	54	9.6.6
<check-op>	**check** 操作	56	9.8
<check-with-sender>	**check** 操作	57	9.8.1
<check-without-sender>	**check** 操作	58	9.8.2
<clear-port-op>	**clear** 端口操作	59	9.9
<connect-op>	**connect** 操作	60	9.10
<constant-definition>	**constant** 定义	61	9.11
<create-op>	**create** 操作	62	9.12
<deactivate-stmt>	**deactivate** 语句	63	9.13
<default-evocation>	**alt** 语句	43	9.3.5
<disconnect-op>	**disconnect** 操作	64	9.14
<do-while-stmt>	**do-while** 语句	65	9.15
<done-component-op>	**done** 部件操作	66	9.16
<else-branch>	**alt** 语句	42	9.3.4
<execute-stmt>	**execute** 语句	67	9.17
<execute-timeout>	**execute** 语句	69	9.17.2
<execute-without-timeout>	**execute** 语句	68	9.17.1
<expression>	表达式	70	9.18
<finalize-component-init>	用在部件类型定义中	75	9.19
<for-stmt>	**for** 语句	79	9.23
<func-op-call>	表达式	73	9.18.3
<function-call>	调用一个函数	80	9.24
<getcall-op>	**getcall** 操作	86	9.25
<getreply-op>	**getreply** 操作	87	9.26
<getverdict-op>	**getverdict** 操作	88	9.27

标 识 符	相关的TTCN-3构件	引 用	
		图	节
<goto-stmt>	**goto** 语句	89	9.28
<if-else-stmt>	**if-else** 语句	90	9.29
<init-component-scope>	用在部件类型定义中	76	9.20
<label-stmt>	**label** 语句	91	9.30
<lit-value>	表达式	71	9.18.1
<log-stmt>	**log** 语句	92	9.31
<map-op>	**map** 操作	93	9.32
<mtc-op>	**mtc** 操作	94	9.33
<nb-call-without-receiver>	**call** 操作	50	9.6.2
<nb-call-with-receiver>	**call** 操作	49	9.6.1
<non-blocking-call-op>	**call** 操作	48	9.6
<operator-appl>	表达式	74	9.18.4
<parameter-handling>	处理函数、可选步骤和测试用例的参数	77	9.21
<port-declaration>	端口声明	95	9.34
<predef-ext-func-call>	调用一个函数（调用一个预定义或外部函数）	85	9.24.5
<raise-op>	**raise** 操作	96	9.35
<raise-with-receiver-op>	**raise** 操作	97	9.35.1
<raise-without-receiver-op>	**raise** 操作	98	9.35.2
<read-timer-op>	**read** 定时器操作	99	9.36
<receive-assignment>	**receive** 操作	103	9.37.3
<receive-op>	**receive** 操作	100	9.37
<receive-with-sender>	**receive** 操作	101	9.37.1
<receive-without-sender>	**receive** 操作	102	9.37.2
<receiving-branch>	**alt** 语句	40	9.3.2
<ref-par-var-calc>	调用一个函数（处理一个引用参数）	82	9.24.2
<ref-par-timer-calc>	调用一个函数（处理定时器参数）	83	9.24.3
<repeat-stmt>	**repeat** 语句	104	9.38
<reply-op>	**reply** 操作	105	9.39
<reply-with-receiver-op>	**reply** 操作	106	9.39.1
<reply-without-receiver-op>	**reply** 操作	107	9.39.2
<return-stmt>	**return** 语句	108	9.40
<return-with-value>	**return** 语句	109	9.40.1
<return-without-value>	**return** 语句	110	9.40.2
<running-component-op>	部件 **running** 操作	111	9.41
<running-comp-act>	部件 **running** 操作	112	9.41.1
<running-comp-snap>	部件 **running** 操作	113	9.41.2
<running-timer-op>	定时器 **running** 操作	114	9.42
<self-op>	**self** 操作	115	9.43
<send-op>	**send** 操作	116	9.44
<send-with-receiver-op>	**send** 操作	117	9.44.1
<send-without-receiver-op>	**send** 操作	118	9.44.2
<setverdict-op>	**setverdict** 操作	119	9.45
<start-component-op>	**start** 部件操作	120	9.46
<start-port-op>	**start** 端口操作	121	9.47
<start-timer-op>	**start** 定时器操作	122	9.48
<start-timer-op-default>	**start** 定时器操作	123	9.48.1
<start-timer-op-duration>	**start** 定时器操作	124	9.48.2
<statement-block>	复合语句中的语句块	78	9.22
<stop-component-op>	**stop** 部件操作	125	9.49

标 识 符	相关的TTCN-3构件	引 用	
		图	节
<stop-mtc>	**stop** 部件操作（停止 MTC）	126	9.49.1
<stop-component>	**stop** 部件操作（停止单个测试部件）	127	9.49.2
<stop-all-comp>	**stop** 部件操作（停止所有部件）	128	9.49.3
<stop-exec-stmt>	**Stop** 执行语句	129	9.50
<stop-control>	**stop** 执行语句（停止模块控制）	130	9.50.1
<stop-port-op>	**stop** 端口操作	131	9.51
<stop-timer-op>	**stop** 定时器操作	132	9.52
<system-op>	**system** 操作	133	9.53
<take-snapshot>	**alt** 语句	39	9.3.1
<timeout-timer-op>	**timeout** 操作	137	9.55
<timer-declaration>	定时器声明	134	9.54
<timer-decl-default>	定时器声明	135	9.54.1
<timer-decl-no-def>	定时器声明	136	9.54.2
<timeout-timer-op>	**timeout** 操作	137	9.55
<unmap-op>	**unmap** 操作	138	9.56
<user-def-func-call>	调用一个函数（调用一个用户定义的函数）	84	9.24.4
<value-par-calculation>	调用一个函数（处理值参数）	81	9.24.1
<var-declaration-init>	变量声明	140	9.57.1
<var-declaration-undef>	变量声明	141	9.57.2
<var-value>	表达式	72	9.18.2
<variable-declaration>	变量声明	139	9.57
<while-stmt>	**while** 语句	140	9.58

ITU-T 系列建议书

A系列	ITU-T工作的组织
D系列	一般资费原则
E系列	综合网络运行、电话业务、业务运行和人为因素
F系列	非话电信业务
G系列	传输系统和媒质、数字系统和网络
H系列	视听及多媒体系统
I系列	综合业务数字网
J系列	有线网络和电视、声音节目及其它多媒体信号的传输
K系列	干扰的防护
L系列	电缆和外部设备其它组件的结构、安装和保护
M系列	电信管理，包括TMN和网络维护
N系列	维护：国际声音节目和电视传输电路
O系列	测量设备的技术规范
P系列	电话传输质量、电话设施及本地线路网络
Q系列	交换和信令
R系列	电报传输
S系列	电报业务终端设备
T系列	远程信息处理业务的终端设备
U系列	电报交换
V系列	电话网上的数据通信
X系列	数据网、开放系统通信和安全性
Y系列	全球信息基础设施、互联网协议问题和下一代网络
Z系列	**电信系统使用的语言和一般性软件情况**

瑞士印刷
2006年，日内瓦