国 际 电 信 联 盟

ITU-T Z.140

国际电信联盟
电信标准化部门

(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）：核心语言

ITU-T Z.140建议书

国际电信联盟

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	
分布式处理环境	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.140建议书

测试和测试控制记法第三版（TTCN-3）：核心语言

摘　要

本建议书定义了测试和测试控制记法第三版（TTCN-3），旨在规范独立于平台、测试方法、协议层和协议的测试套件。TTCN-3 可用于规范各种各样通信端口上所有类型的响应系统测试。典型的应用领域是协议测试（包括移动协议和国际互联网协议）、服务测试（包括增补的服务）、模块测试、基于 CORBA 平台和 API 的测试。有关物理层协议的测试套件规范超出了本建议书的讨论范围。

TTCN-3 核心语言可用多种表示格式进行描述。本建议书定义了核心语言，ITU-T Z.141 建议书定义了 TTCN 的表格表示格式（TFT），ITU-T Z.142 建议书定义了 TTCN 的图形表示格式（GFT）。有关这些格式的规范超出了本建议书的讨论范围。核心语言有以下三个作用：

1） 作为基于文本的通用测试语言；

2） 作为TTCN测试套件和TTCN工具之间的标准交换格式；

3） 作为各种各样表示格式的语义基础（相关的话，还可作为句法基础）。

核心语言可独立于表示格式使用。不过，无论是表格表示格式，还是图形表示格式，都不能脱离核心语言单独使用。这些表示格式的使用和实现都应基于核心语言。

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.140 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/。

目 录

ITU-T Z.140建议书

测试和测试控制记法第三版（TTCN-3）：核心语言

1 范围

本建议书定义了测试和测试控制记法第三版（TTCN-3）的核心语言。TTCN-3 可用于规范各种各样通信端口上所有类型的响应系统测试。典型的应用领域是协议测试（包括移动协议和国际互联网协议）、服务测试（包括增补的服务）、模块测试、基于 CORBA 平台和 API 等的测试。TTCN-3 并不限于一致性测试，它可用于众多其它类型的测试，包括互操作性测试、健壮性测试、回归测试、系统和集成测试。有关物理层协议的测试套件规范超出了本建议书的讨论范围。

TTCN-3 旨在规范独立于测试方法、协议层和协议的测试套件。为 TTCN-3 定义了各种各样的表示格式，如表格表示格式（ITU-T Z.141 建议书 [1]）和图形表示格式（ITU-T Z.142 建议书 [2]）。有关这些格式的规范超出了本建议书的讨论范围。

TTCN-3 的设计已考虑到 TTCN-3 转换器和编译器的最终实现，但有关从抽象测试套件（ATS）到可执行测试套件（ETS）的实现方法超出了本建议书的讨论范围。

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。所有的建议书和其它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它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.141 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Tabular presentation format (TFT).*

[2] ITU-T Recommendation Z.142 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Graphical presentation format (GFT).*

[3] ITU-T Recommendation Z.143 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Operational semantics.*

[4] ITU-T Recommendation Z.144 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Runtime interface (TRI).*

[5] ITU-T Recommendation Z.145 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Control interface (TCI).*

[6] ITU-T Recommendation Z.146 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Using ASN.1 with TTCN-3.*

[7] ITU-T Recommendation X.290 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts.*
ISO/IEC 9646-1:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts.*

[8] ITU-T Recommendation X.292 (2002), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN).*
ISO/IEC 9646-3:1998, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN).*

[9] ITU-T Recommendation T.50 (1992), *International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) – Information technology – 7-bit coded character set for information interchange.*
ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange.*

[10] ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS).*

[11] ISO/IEC 6429:1992, *Information technology – Control functions for coded character sets.*

3 定义和缩写

3.1 定义

就本建议书而言，ITU-T X.290 建议书[7]、ITU-T X.292 建议书[8]中给出的术语和定义以及下面给出的术语和定义适用：

3.1.1 actual parameter 实参：在调用位置上定义的、作为参数传递给被调用实体（函数、测试用例、可选步骤等）的值、表达式、模板或名称引用（标识符）。

注 — 传递给一个调用的所有实参个数、次序和类型都应与被调用实体中定义的形参列表相一致。

3.1.2 basic types 基本类型：第 6.1.0 节和第 6.1.1 节中描述的预定义 TTCN-3 类型集。

注 — 通过其名称来对基本类型进行引用。

3.1.3 compatible type 兼容类型：TTCN-3 不是强类型分类，但语言需要类型兼容性。

注 — 如果满足第6.7节中的条件，那么变量、常量、模板等都有兼容的类型。

3.1.4 communication port 通信端口：以利测试部件之间通信的抽象机制。

注 — 一个通信端口在接收方向上当作一个先入先出（FIFO）队列来建模。端口可以是基于消息的、基于过程的或二者的混合。

3.1.5 data types 数据类型：简单基本类型、基本字符串类型、结构类型、特殊数据类型 **anytype** 以及基于以上类型的所有用户定义类型的共用名称（见表 3）。

3.1.6 defined types (defined TTCN-3 types) 定义的类型（定义的 TTCN-3 类型）：所有预定义 TTCN-3 类型（基本类型、所有结构类型、**anytype** 类型、地址、端口和部件类型以及缺省类型）和所有在模块中声明的或从其它 TTCN-3 模块引入的用户定义类型。

3.1.7 dynamic parameterization 动态参数化：参数化的一种，当中，实参依赖于运行时间事件，如实参的值是一个在运行时间收到的值，或者按照某种逻辑关系取决于所收到的值。

3.1.8 exception 异常：在基于过程的通信的情况下，如果应答实体不能以正常的期望响应来响应远程过程调用，那么它将产生一个异常（如果定义了异常的话）。

3.1.9 formal parameter 形参：类型名称或类型模板引用（标识符），不是在定义一个实体（函数、测试用例、可选步骤等）时被解析，而是在调用它时被解析。

注 — 在形参位置上使用的实际值或模板（或其名称）是从调用实体的位置上传递过来的（也可参见实参定义）。

3.1.10 global visibility 全局可见性：实体（模块参数、常量、模板等）的属性，其标识符可以在定义它的模块中任意位置（包括在同一模块内定义的所有函数、测试用例和可选步骤以及该模块的控制部分）被引用。

3.1.11 Implementation Conformance Statement (ICS) 实现方案一致性声明（ICS）：参见 ITU-T X.290 建议书 [7]。

3.1.12 Implementation eXtra Information for Testing (IXIT) 用于测试的实现方案附加信息（IXIT）：参见 ITU-T X.290 建议书 [7]。

3.1.13 Implementation Under Test (IUT) 接受测试的实现方案（IUT）：参见 ITU-T X.290 建议书 [7]。

3.1.14 known types 已知类型：所有 TTCN-3 预定义类型、TTCN-3 模块中定义的类型以及从其它 TTCN-3 模块或非 TTCN-3 模块引入到该模块中的类型的集合。

3.1.15 left-hand side (of assignment) （赋值）左侧：放在赋值符号（:=）左侧的值、模板变量标识符、结构类型值或模板变量（如果有数组下标，那么包括它）的字段名。

注 — 在声明中与/或被修改的模板定义中，放在赋值符号左侧（:=）的常量、模块参数、定时器、结构类型域名或模板题头（包括模板类型、名称和形参列表），不在本定义范围之内，不是赋值的一部分。

3.1.16 local visibility 局部可见性：实体（常量和变量等）的属性，其标识符仅可在定义它的函数、测试用例或可选步骤（altstep）中被引用。

3.1.17 Main Test Component (MTC) 主测试部件（MTC）：参见 ITU-T X.292 建议书 [8]。

3.1.18 passing parameter by value 通过值传递参数：在进入一个可参数化的实体之前先估计变量值的传递参数方式。

注 — 只传递了变量的值，在被调用实体中变量的变化对调用者可见的实参没有任何影响。

3.1.19 passing parameter by reference 通过引用传递参数：在进入函数、可选步骤等之前不估计变量值的传递参数方式，进入时调用过程（函数、可选步骤等）传递给被调用过程的是参数的引用。

注 — 被调用过程中自变量的所有变化都会影响到调用者所见的实际变量。

3.1.20 Parallel Test Component (PTC) 并发测试部件（PTC）：见 ITU-T X.292 建议书 [8]。

3.1.21 right-hand side (of assignment) （赋值）右侧：放在赋值符号（:=）右侧的表达式、模板引用或特征参数标识符。

注 — 在常量、模块参数、定时器、模板或被修改的模板声明中，放在赋值符号右侧（:=）的表达式和模板引用，不在本定义范围之内，不是赋值的一部分。

3.1.22 root type 源类型：用户定义的 TTCN-3 类型可以追溯到的基本类型、结构类型、特殊数据类型、特殊配置类型或特殊缺省类型。

3.1.23 static parameterization 静态参数化：参数化的一种，当中，实参独立于运行时间事件，即实参值在编译时就已知，如果是模块参数，那么在测试套执行启动时知道模块参数的值（例如，从测试套说明中得知，在此计算引入的定义，或者在执行之前，测试系统已经知道参数的值）。

注 — 编译时，所有类型都是已知的，即静态绑定。

3.1.24 strong typing 强分类：根据类型名称等同性来严格实施的类型兼容性，没有任何异常。

3.1.25 System Under Test (SUT) 接受测试的系统（SUT）：参见 ITU-T X.290 建议书 [7]。

3.1.26 template 模板：TTCN-3 模板是用于测试的特殊数据结构，用于传输一组不同的值或用于检查收到的一组值是否与模板说明相匹配。

3.1.27 test behaviour 测试行为：（或行为）当执行一个 **`execute`** 或 **`start`** 部件语句时，在一个测试部件启动的测试用例或函数，以及称为递归的所有函数和可选步骤。

注 — 在测试用例执行期间，每个测试部件都有其自身的行为，因此，测试系统中可能会并发运行几个测试行为（即一个测试用例可看作是几个测试行为的集合）。

3.1.28 test case 测试用例：参见 ITU-T X.290 建议书 [7]。

3.1.29 test case error 测试用例错误：参见 ITU-T X.290 建议书 [7]。

3.1.30 test suite 测试套件：TTCN-3 模块集，包括测试用例的完整定义集，可选地，可补充一个或多个 TTCN-3 控制部件。

3.1.31 test system 测试系统：参见 ITU-T X.290 建议书 [7]。

3.1.32 test system interface 测试系统接口：提供从（抽象）TTCN-3 测试系统中可用端口到 SUT 提供之可用端口映射的测试部件。

3.1.33 type compatibility 类型兼容性：语言特性，它允许使用一个给定类型的值、表达式或模板作为另一个类型的实际值（如在赋值时，在调用一个函数、引用一个模板等时作为实参，或者作为一个函数的返回值）。

注 — 值、表达式或模板的类型和当前值应兼容于其它类型。

3.1.34 value parameterization 值的参数化：把一个值或模板作为一个实参传递给一个参数化对象的能力。

注 — 而后该实际值参数完成对该对象的描述。

3.1.35 user-defined type 用户定义类型：通过定义一个基本类型的子类型或声明一个结构类型来定义的类型。

注 — 用户定义类型通过其标识符（名称）来引用。

3.1.36 value notation 值记法：使一个标识符和一个特定类型的给定值或作用范围相关联的记法。

注 — 值可以是常量或变量。

3.2 缩写

本建议书采用下列缩写：

API	应用编程接口
ATS	抽象测试套件
BMP	基本多语言平台
BNF	巴科斯范式

CORBA	通用对象请求代理体系结构
ETS	可执行的测试套件
FIFO	先入先出
ICS	实现方案一致性声明
IRV	国际参考版本
IUT	接受测试的实现方案
IXIT	用于测试的实现方案附加信息
MTC	主测试部件
PTC	并发测试部件
SUT	接受测试的系统
TSI	测试系统接口

4 引言

4.0 概述

TTCN-3 是一种灵活而强有力的语言，适用于描述多种通信接口上的各种响应系统测试类型。典型的应用领域包括：协议测试（包括移动和互联网协议）、服务测试（包括补充业务）、模块测试、基于 CORBA 平台的测试、API 测试等。TTCN-3 并不限于一致性测试，它还可用于许多其它种类的测试，包括互操作性测试、健壮性测试、回归测试、系统测试和集成测试。

TTCN-3 包括以下基本特性：

- 描述动态并发测试配置的能力；
- 基于过程的操作和基于消息的通信；
- 描述编码信息和其它属性（包括用户扩展性）的能力；
- 描述数据和带强有力匹配机制的特征模板的能力；
- 值的参数化；
- 赋值和测试判定的处理；
- 测试套件参数化和测试用例选择机制；
- TTCN-3与其它语言的结合使用；
- 具有良好定义的语法、交换格式和静态语义；
- 不同的表示格式（如表格和图形表示格式）；
- 精确的执行算法（操作语义）。

4.1 核心语言和表示格式

TTCN-3 规范分为若干部分：第一部分在本建议书中进行定义，是 TTCN-3 的核心语言；第二部分在 ITU-T Z.141 建议书 [1]中进行定义，是 TTCN-3 的表格表示格式；第三部分在 ITU-T Z.142 建议书 [2]中进行定义，是 TTCN-3 的图形表示格式；第四部分在 ITU-T Z.143 建议书 [3]中进行定义，包含该语言的操作语义；第五部分在 ITU-T Z.144 建议书 [4]中进行定义，定义了 TTCN-3 的运行时间接口（TRI）；第六部分在 ITU-T Z.145 建议书 [5]中进行定义，定义了 TTCN-3 的控制接口（TCI）；第七部分在 ITU-T Z.146 建议书 [6]中进行定义，规定了 TTCN-3 对 ASN.1 定义的使用。

核心语言有三个目的：

a） 作为广义的、基于文本的测试语言，拥有自己的权限；

b） 作为TTCN-3工具之间TTCN-3测试套件的标准化交换格式；

c） 作为各种表示格式的语义基础（以及相关的语法基础）。

核心语言可以独立于表示格式使用。不过，表格格式和图形格式都不能独立于核心语言使用。这些表示格式的使用和实现应基于核心语言。

在预期的不同表示格式集合中，表格格式和图形格式是最先被标准化的，其它格式可以是标准化的表示格式或是 TTCN-3 用户自行定义的私有表示格式。本建议书中不对这些额外的表示格式进行定义。

TTCN-3 可以与其它类型－值记法一起使用，这种情况下，其它语言中的定义可作为另一种数据类型和值语法来使用。本标准的其它部分规定了 TTCN-3 对某些其它语言的使用。对其它语言的支持不限于 ITU-T Z.140 系列建议书中所规定的那些，它还定义了能与 TTCN-3 结合使用的支持语言，应使用本建议书中给出的规则。

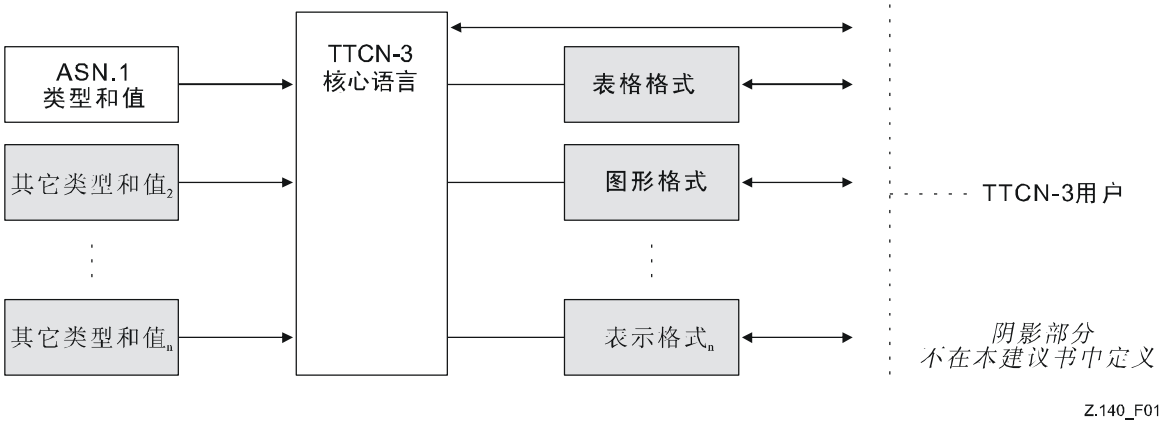


图 1/Z.140－核心语言和各种表示格式的用户视图

通过一整套完整的语法（见附件 A）和操作语义（见 ITU-T Z.143 建议书 [3]）来定义核心语言。出于某些基础应用领域或方法学方面的考虑，它包含不限制语言使用的最小静态语义（在本建议书的主体部分和附件 A 中给出）。

4.2 描述的一致性

在本建议书的主体部分（第 5 节～第 28 节），通过文本描述方式，在本建议书的附件 A，通过形式化方式，从语法和语义角度，对 TTCN-3 语言进行了详细说明。任何情况下，当文本描述不能表达清楚时，可以用形式化描述的方式来完成。如果文本和形式化描述出现矛盾时，应以形式化描述为准。

4.3 一致性

对一个声称与该版本语言一致的实现方案而言，本建议书中规定的所有特征的实现应与本建议书和 ITU-T Z.143 建议书 [3]中给出的需求相一致。

5 基本语言要素

5.0 概述

TTCN-3 的顶层单元是模块。一个模块不能分解为子模块，但它可以从其它模块引入定义。模块可以带有模块参数，以便允许测试套件参数化。

一个模块由一个定义部分和一个控制部分组成。模块的定义部分定义了测试部件、通信端口、数据类型、常量、测试数据模板、函数、端口上的过程调用特征、测试用例等。

模块的控制部分调用测试用例并控制其执行。控制部分也可以声明（局部）变量等，程序语句（如 **`if-else`** 和 **`do-while`**）可用于描述各个测试用例的选择和执行次序。TTCN-3 不支持全局变量的概念。

TTCN-3 有许多预定义的基本数据类型和结构类型，如记录、集合、联合、枚举类型和数组。

模板是一种特殊的数据结构，它为描述在测试端口上发送和接收的测试数据提供了参数化和匹配机制。在这些端口上的操作提供了基于消息和基于过程的通信能力。过程调用可用于非基于消息的测试实现方案。

动态的测试行为表达为测试用例，TTCN-3 程序语句包括强有力的行为描述机制，如通信和定时器事件、交织和缺省行为的选择性接收。TTCN-3 也支持测试判定赋值和日志机制。

最后，可以对 TTCN-3 语言要素赋予属性，如编码信息和显示属性，也可以描述（非标准化的）用户定义的属性。

表 1/Z.140－TTCN-3语言要素一览表

语言要素	相关的关键字	是否在模块定义中描述	是否在模块控制中描述	是否在函数/可选步骤/测试用例中描述	是否在测试部件中描述
TTCN-3 模块定义	**module**				
从其它模块引入定义	**import**	是			
定义分组	**group**	是			
数据类型定义	**type**	是			
通信端口定义	**port**	是			
测试部件定义	**component**	是			
特征定义	**signature**	是			
外部函数/常量定义	**external**	是			
常量定义	**const**	是	是	是	是
数据/特征模板定义	**template**	是	是	是	是
函数定义	**function**	是			
可选步骤定义	**altstep**	是			
测试用例定义	**testcase**	是			
值变量声明	**var**		是	是	是
模板变量声明	**var template**		是	是	是
定时器声明	**timer**		是	是	是
注 — 变量、常量、类型和其它语言要素的“定义”和“声明”记法在本建议书中交替使用。这两个记法之间的区别仅对实现方案有用，如 C 和 C＋＋这类编程语言中那样。在 TTCN-3 这一层次上，这两个记法具有相同的含义。					

5.1 语言要素的次序

通常，声明的次序可以是任意的。在一个语句和声明块中，如一个函数体或 **if-else** 语句的一个分支中，所有的声明（如果有的话）都应在该块的开始处进行声明。

例如：

```
//这是一个合法的TTCN-3声明混合。
 :
var MyVarType MyVar2 := 3;
const integer MyConst:= 1;
if (x > 10)
{
  var integer MyVar1:= 1;
  :
  MyVar1:= MyVar1 + 10;
  :
}

 :
```

模块定义部分中的声明可以以任意次序给出。不过，在模块控制部分、测试用例定义、函数和可选步骤内，所有要求的声明都必须事先给出，这意味着，尤其是局部变量、局部定时器和局部常量在其被声明之前绝不能使用。该规则的唯一例外是标签。在 **goto** 语句中可能会在标签声明之前预先引用标签。

5.2 参数化

5.2.0 静态参数化和动态参数化

TTCN-3 根据以下限制，支持值的参数化：

a） 不能参数化的语言要素有：**const**、**var**、**timer**、**control**、**group** 和 **import**；

b） 语言要素**module**允许静态的值参数化，以支持测试套件参数，也就是说，在编译时，该参数化可以是可解析的也可以是不可解析的，但应在运行开始之时对它进行解析（即在运行时是静态的）。这意味着，在运行时，模块的参数值是全局可见的，但不能改变；

c） 所有用户定义的**type**定义（包括结构化的类型定义，如**record**、**set**等）和特殊的配置类型**address**支持静态的值参数化，即应在编译时对该参数化进行解析；

d） 语言要素**template**、**signature**、**testcase**、**altstep**和**function**支持动态的值参数化（即该参数化在运行时应是可解析的）。

哪些语言要素可以被参数化以及哪些可以传给它们作为参数的汇总情况如表 2 所示。

表 2/Z.140－参数化TTCN-3语言要素一览表

关 键 字	值 参 数 化	在形参/实参列表中允许出现的值的类型
module	在运行开始时，静态	所有基本类型、所有用户定义类型和 **address** 类型的值。
type **（注 1）**	在编译时，静态	所有基本类型、所有用户定义类型和 **address** 类型的值。
template	在运行时，动态	所有基本类型、所有用户定义类型、**address** 类型和 **template** 的值。
function	在运行时，动态	所有基本类型、所有用户定义类型、**address** 类型、**component** 类型、**port** 类型、**default**、**template** 和 **timer** 的值。
altstep	在运行时，动态	所有基本类型、所有用户定义类型、**address** 类型、**component** 类型、**port** 类型、**default**、**template** 和 **timer** 的值。
testcase	在运行时，动态	所有基本类型、所有用户定义类型、**address** 类型和 **template** 的值。
signature	在运行时，动态	所有基本类型、所有用户定义类型、**address** 类型和 **component** 类型的值。
注 1 — **record of**、**set of**、**enumerated**、**port**、**component** 和 **sub-type definitions** 不允许参数化。 注 2 — 不同语言要素中参数化的语法例子和特殊用法在本建议书的相关章节中给出。		

5.2.1 通过传参和传值方式的参数传递

5.2.1.0 概述

缺省情况下，基本类型、基本串类型、用户定义的结构类型、地址类型和部件类型通过传值来传递所有的实参。这可以通过关键字 **in** 选择性地来表示。如果通过传参方式来传递上述类型的参数，那么应使用关键字 **out** 或 **inout**。

定时器和端口总是通过传参的方式来传递参数。通过关键字 **timer** 来表示定时器参数，通过其端口类型来标识端口参数。可以选择性地使用关键字 **inout** 来表示通过传参方式的参数传递。

5.2.1.1 通过传参方式的参数传递

通过传参方式来传递参数有以下限制：

a） 只有可选步骤、函数、特征和测试用例的形参列表可包含通过传参方式传递的参数；

注 — 如何在特征（见第23节）和可选步骤（见第16.2.1节和第21.3.1节）中使用通过传参方式传递的参数，有进一步的限制。

b） 实参只能是值或模板变量、端口或定时器。

例如：

```
function MyFunction(inout boolean MyReferenceParameter){ … };
// 通过传参方式来传递MyReferenceParameter。可以在该函数中读出和设置实参。

function MyFunction(out template boolean MyReferenceParameter){ … };
// 通过传参方式来传递MyReferenceParameter。仅可以在该函数中设置实参。
```

5.2.1.2 通过传值方式的参数传递

通过传值方式传递的参数可以是变量，也可以是常量、模板等。

```
function MyFunction(in template MyTemplateType MyValueParameter){ … };
// 通过传值方式来传递MyValueParameter，关键字in是可选的。
```

5.2.2 形参和实参列表

在实参列表中出现的语言要素数目和次序应与其在相应形参列表中出现的元素数目和次序相同。另外，每个实参的类型都应兼容于相应的形参类型。

例如：

```
// 带有形参列表的函数定义。
function MyFunction(integer FormalPar1, boolean FormalPar2, bitstring FormalPar3) { … }

// 带有实参列表的函数调用。
MyFunction(123, true,'1100'B);
```

5.2.3 空形参列表

如果 TTCN-3 语言要素 **function**、**testcase**、**signature**、**altstep** 或 **external function** 的形参列表是空的，那么在该元素的声明和调用中都应包括空括号。在所有其它情况下，都可以省略空括号。

例如：

```
// 带有空参数列表的函数定义可以写为：
function MyFunction(){ … }

// 带有空参数列表的记录定义可以写为：
type record MyRecord { … }
```

5.2.4 嵌套式参数列表

被描述为实参的所有参数化实体都应在顶层实参列表中解析其自身的参数。

例如：

```
//给定的消息定义
type record MyMessageType
{
  integer field1,
  charstring  field2,
  boolean  field3
}

// 消息模板可以是：
template MyMessageType MyTemplate(integer MyValue) :=
```

```
{
  field1 := MyValue,
  field2 := pattern "abc*xyz",
  field3 := true
}

// 利用模板参数化的测试用例可以是:
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1 {
:
MyPCO.receive(RxMsg);
}

// 当测试用例在控制部分中被调用且参数化的模板用作实参时，必须提供模板的实参。
control
{ :
  execute( TC001(MyTemplate(7)));
  :
}
```

5.2.5 模板类型的形参

5.2.5.1 模板的参数化和匹配属性

为了能使模板或匹配符号作为实参传递，在相应的形参类型字段之前应添加额外的关键字 **template**。这样可使参数成为模板类型，使得相关类型的允许参数扩展至：既包括匹配属性的适当集（见附件 B），又包括值的普通集。模板参数字段不得通过传参方式进行调用。

例如：

```
// 模板
template MyMessageType MyTemplate (template integer MyFormalParam):=
{ field1 := MyFormalParam optional,
  field2 := pattern "abc*xyz",
  field3 := true
}

// 可按以下方式使用:
pco1.receive(MyTemplate(?));
// 或按以下方式使用:
pco1.receive(MyTemplate(omit));
```

5.2.5.2 使用模板类型参数的语言要素

只有 **function**、 **testcase**、**altstep** 和 **template** 定义可以有模板类型的形参。

例如：

```
function MyBehaviour(template MyMsgType MyFormalParameter)
runs on MyComponentType
{  :
  pco1.receive(MyFormalParameter);
   :
}
```

5.3 作用范围规则

5.3.0 概述

TTCN-3 提供了七个基本的作用范围单位：

a） 模块定义部分；
b） 模块的控制部分；
c） 部件类型；
d） 函数；
e） 可选步骤；
f） 测试用例；
g） 复合语句中的“声明和语句块”。

注 1 — 用于组的附加作用范围规则，在第7.3.1节中给出。

注 2 — 用于**for**循环中计数的附加作用范围规则，在第19.7节中给出。

作用范围的每个单位由声明（可选的）组成。可以通过使用 TTCN-3 程序语句和操作，用作用范围单位 — 模块的控制部分、函数、测试用例、可选步骤和复合语句中的“声明和语句块”，来额外描述行为的某种形式（见第 18 节）。

在模块定义部分中但在其它作用范围单元外的定义是全局可见的，也就是说，它们可以用在模块的任意位置，包括在模块中定义的所有函数、测试用例和可选步骤以及控制部分。从其它模块引入的标识符对整个引入模块来说也是全局可见的。

模块控制部分中的定义具有局部可见性，即只能用在控制部分中。

在测试部件类型中所做的定义只能在通过 **runs on** 子句调用该部件类型或一致的测试部件类型的函数、测试用例和可选步骤中使用（见第 16.3 节）。

测试用例、可选步骤和函数是单独的作用范围单位，它们之间没有任何层次结构，即其主体开始处所做的声明具有局部可见性，应只用在给定的测试用例、可选步骤或函数中（例如，在测试用例中所做的声明对该测试用例调用的函数或该测试用例使用的可选步骤是不可见的）。

复合语句包括“语句和声明块”，如 **if-else-**、**while-**、**do-while-** 或 **alt-**语句。它们可以用在一个模块的控制部分、测试用例、可选步骤、函数中，或者嵌入在其它复合语句中，例如，在一个 **while-**循环中使用 **if-else-**语句。

复合语句和嵌入式复合语句中的“语句和声明块”对包括给定“语句和声明块”的作用范围单元和任何嵌入式“语句和声明块”来说，都具有层次结构。在一个“语句和声明块”中所做的定义具有局部可见性。

作用范围单元的层次结构如图 2 所示。较高层次的作用范围单元声明对层次结构同一分支中较低层次的所有单元来说都是可见的。层次结构较低层次的作用范围单元声明对较高层次上的那些单元来说都是不可见的。

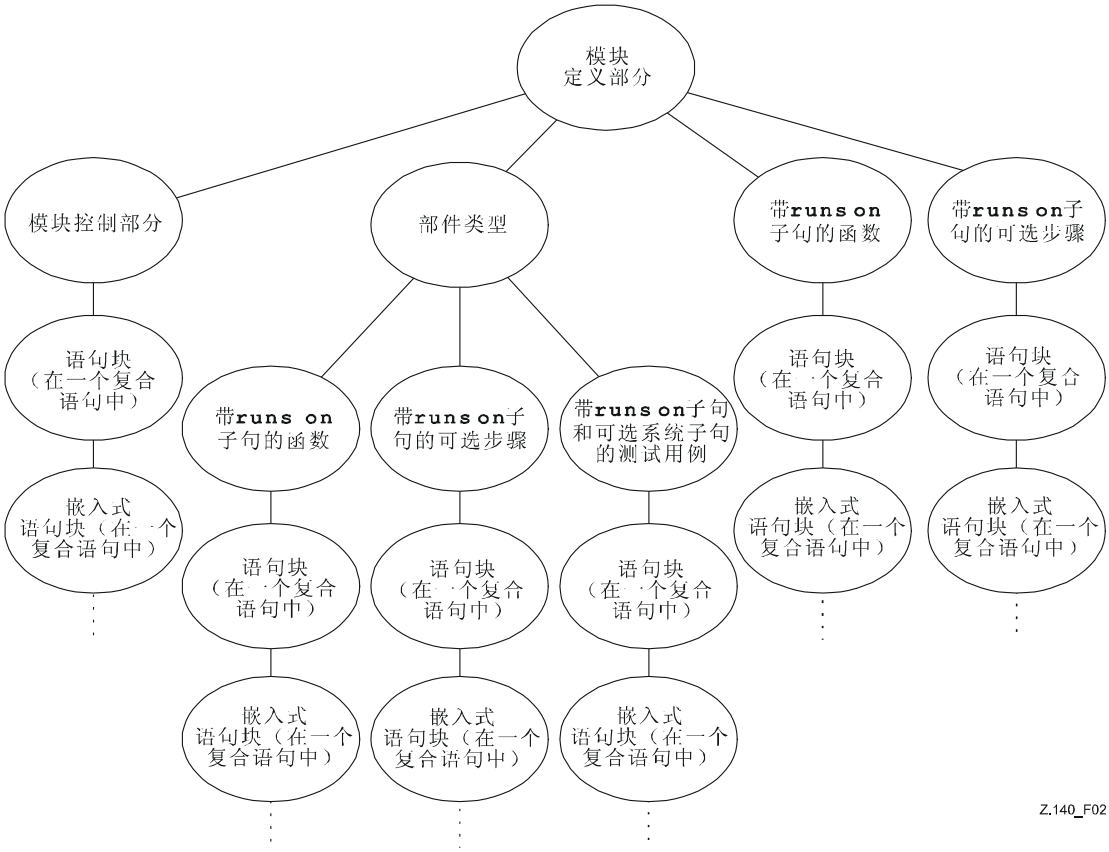


图 2/Z.140－作用范围单元的层次结构

例如：

```
module MyModule
{  :
   const integer MyConst := 0;
                                  // 对MyBehaviourA 和 MyBehaviourB，MyConst是可见的。
   :
   function MyBehaviourA()
   {   :
       const integer A := 1;
                                // 常量A仅对MyBehaviourA是可见的。
       :
   }

   function MyBehaviourB()
   {   :
       const integer B := 1;
                                // 常量B仅对MyBehaviourB是可见的。
       :

   }
}
```

5.3.1 形参的作用范围

在一个参数化语言要素中（如在一个函数调用中），形参的作用范围应限定于这些参数出现的定义中以及相同作用范围层次中的较低层次作用范围中，也就是说，应遵循正常的作用范围规则（见第 5.3.0 节）。

5.3.2 标识符的唯一性

TTCN-3 要求标识符具有唯一性，即在相同作用范围层次中的所有标识符应互不相同。这意味着，在同一作用范围层次分支中，较低层次作用范围中的声明不应重复使用与较高层次作用范围中声明相同的标识符。结构类型字段、枚举值和组的标识符不必全局唯一；不过，在枚举值的情况下，标识符应只能被其它枚举类型中的枚举值重复使用。标识符唯一性规则也应适用于形参标识符。

例如：

```
module MyModule
{  :
   const integer A := 1;
   :
   function MyBehaviourA()
   {   :
       const integer A := 1; // 不允许
       :
       if(…)
       {    :
        const boolean A := true; // 不允许
        :
       }
   }
}

// 下面不在相同作用范围层次中声明的常量是允许的（假设在模块的头部没有A的声明）。
function MyBehaviourA()
{  :
   const integer A := 1;
   :
}

function MyBehaviourB()
{  :
   const integer A := 1;
   :
}
```

5.4 标识符和关键字

TTCN-3 标识符大小写敏感。TTCN-3 关键字的所有字母都应小写（见附件 A）。TTCN-3 关键字既不能用作 TTCN-3 对象的标识符，也不能用作从其它语言的模块中引入的对象的标识符。

6 类型和值

6.0 概述

TTCN-3 支持许多预定义的基本类型。这些基本类型包括与程序语言正常关联的基本类型，如 **integer**、**boolean** 和 string 类型，以及一些 TTCN-3 特定的类型，如 **verdicttype**。可以自这些基本类型来构造结构类型，如 **record** 类型、**set** 类型和 **enumerated** 类型。

特殊的数据类型 **anytype** 定义为是模块中所有已知数据类型和地址类型的联合。

与测试配置相关的特殊类型，如 **address**、**port** 和 **component** 可用于定义测试系统的体系结构（见第 22 节）。

特殊类型 **default** 可以用于缺省处理（见第 21 节）。

TTCN-3 类型汇总情况如表 3 所示：

表 3/Z.140－TTCN-3类型一览表

类型分类	关 键 字	子 类 型
简单基本类型	**integer**	值域、列表
	float	值域、列表
	boolean	列表
	objid	列表
	verdicttype	列表
基本串类型	**bitstring**	列表、长度
	hexstring	列表、长度
	octetstring	列表、长度
	charstring	值域、列表、长度、样式
	universal charstring	值域、列表、长度、样式
结构类型	**record**	列表（见注释）
	record of	列表（见注释）、长度
	set	列表（见注释）
	set of	列表（见注释）、长度
	enumerated	列表（见注释）
	union	列表（见注释）
特殊的数据类型	**anytype**	列表（见注释）
特殊的配置类型	**address**	
	port	
	component	
特殊的缺省类型	**default**	
注 — 当从一个已经存在的父类型定义一个新的约束类型时，可以定义这些类型的列表子类型，但列表子类型不能在第一个父类型的声明中直接定义。		

6.1 基本类型和值

6.1.0 简单基本类型和值

TTCN-3 支持下列基本类型：

a） **integer**：其值为所有正整数、负整数和零的一种类型。

整数型的值应用一个或多个数字来表示；除非0为值，否则第一位不应为0；值0应用单个0来表示。

b） **float：**用于描述浮点数的一种类型。

通常，浮点数可定义为：<尾数> × <基数>^{<指数>}，其中，<尾数>是一个正整数或负整数，<基数>是一个正整数（多数情况为2、10或16），<指数>是一个正整数或负整数。

在TTCN-3中，浮点数的值记法限定为以值10为基数。浮点值可以使用下列两种值记法形式来表示：

— 在一个数字序列中带小数点的十进制记法，如1.23（表示123 × 10^{-2}），2.783（即2783 × 10^{-3}）或 －123.456789（表示－123456789 × 10^{-6}）；或者

— 通过E将两个数字分开来表示，第一个数字描述尾数，第二个数字描述指数，如12.3E4（表示123 × 10^{3}） 或 －12.3E-4（表示－123 × 10^{-5}）。

注 — 与浮点值的常规定义相反，TTCN-3值记法的尾数既可以是整数，也可以是十进制数。

c） **boolean：**由两个不同值组成的一种类型。

布尔类型的值应使用**true**和**false**来表示。

d） 空。

e） 测试判定使用的一种类型，该类型有五个不同的值。**Verdicttype**的值可以通过**pass**、 **fail**、**inconc**、**none** 和 **error**来表示。

6.1.1 基本串类型和值

TTCN-3 支持下列基本串类型：

注 1 — TTCN-3中的通用术语“ 串”或“ 串类型”指的是**bitstring**、**hexstring**、**octetstring**、**charstring**和 **universal charstring**。

a） **bitstring**：一种类型，其不同值为0位、1位或多位的有序序列。

bitstring类型的值将用任意数目（可能为0）的二进制数字：0、1来表示，以字符“'”开始，后接字符对“'B”。

例1： `'01101'B`.

b） **hexstring：**一种类型，其不同值为0位、1位或多位十六进制数字的有序序列，每个值对应一个四比特的有序序列。

Hexstring类型的值将用任意数目（可能为0）的十六进制数字来表示（十六进制数字中大写和小写字母可等同使用）：

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

以单引号“'”开始，后接字符对“'H”；使用十六进制表示法，每个十六进制数字用于表示半八位字节的值。

例2： `'AB01D'H`
`'ab01d'H`
`'Ab01D'H`

c） **octetstring**：一种类型，其不同值为0个或正偶数个十六进制数字的有序序列（每对数字对应一个八比特的有序序列）。

octetstring类型的值将用任意但必须是偶数数目的十六进制数字来表示（十六进制数字中大写和小写字母可等同使用）：

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

以单引号“'”开头，后接字符对“'O”；使用十六进制表示法时，每个十六进制数字用于表示半八位字节的值。

例3： `'FF96'O`
`'ff96'O`
`'Ff96'O`

d） **charstring：**一种类型，其不同值为0个、1个或多个与8.2/T.50 [9]中所述国际参考版本（IRV）相符的ITU-T T.50建议书 [9]版本的字符。

注 2 — ITU-T T.50建议书 [9]中描述的国际参考字母表（即以前的第5号国际字母表－IA5）的IRV版本，与ISO/IEC 646的IRV版本等同。

charstring类型的值将用来自相关字符集的任意数目（可能为0）个字符来表示，并用双引号（"）将它括起来，使用预定义的转换函数int2char来计算，其编码的正整数值作为变量（见第C.1节）。

注 3 — 预定义的转换函数只能返回单字符长度的值。

在串中需要包含字符双引号（"）来定义字符串的情况下，在同一行中使用一对双引号来表示该字符，且其间没有空格字符。

例4： ""abcd""表示文字串"abcd"。

e） 前面跟有关键字**universal**的字符串类型表示其值为0个、1个或多个ISO/IEC 10646 [10]字符的类型。

universal charstring值也可以用相关字符集的任意个（可能为0）字符组成，并用双引号（"）将它括起来，使用以正整数编码值作为参数的预定义转换函数（见第C.3节）或通过一个“四元组”来计算。

注 4 — 预定义的转换函数只能返回单字符长度的值。

在串中需要包含字符双引号（"）来定义字符串的情况下，在同一行中使用一对双引号来表示该字符，且其间没有空格字符。

“四元组”只能表示一个单个字符，依据ISO/IEC 10646 [10]，使用其组、面、行和单元格的十进制值来表示该字符，由关键字**char**来引导，括在一对括号内，并用逗号分隔（例如，**char** (0, 0, 1, 113)表示匈牙利字符 "ű"）。依据第一种方法（括在一对双引号内），在串中需要包含字符双引号（"）来表示的情况下，在同一行中使用一对双引号来表示该双引号字符来表示该字符，且其间没有空格字符。在使用连接运算符的字符串值表示法中，可以混合使用这两种方法。

例5：赋值语句"the Braille character" & **char** (0, 0, 40, 48) & "looks like this" 表示文字串：the Braille character ⣿ looks like this。

注 5 — 可以使用预定义的转换函数或四元组形式来表示控制字符。

缺省情况下，**universal charstring**应该与ISO/IEC 10646 [10]第14.2节中描述的UCS-4编码表示格式相一致。

注 6 — UCS-4是一种编码格式，它使用一个固定的、32位长的字段来表示任意UCS字符。

该缺省编码可以利用定义的变量属性（见第28.2.3节）来替代。在附件E中定义了以下使用这些属性的有用字符串类型：utf8string、bmpstring、utf16string 和 iso8859string。

6.1.2 访问单个串元素

可以使用一个类似数组的语法来访问一个串类型中的各个元素，串中只有一个元素可以被访问。

表 4 指出了不同串类型元素的长度单位。

下标应该从 0 开始编号。

例如：

```
// 给定
MyBitString := '11110111'B;
// 而后进行
MyBitString[4] := '1'B;
// 以比特串表示的结果为 '11111111'B
```

6.2 基本类型的子类型

6.2.0 概述

用户定义的类型将用关键字 **type** 来表示。可以根据表 3，利用用户定义的类型，在基本类型、结构类型和 **anytype** 上创建子类型（如列表、值域和长度限制）。

6.2.1 值列表

TTCN-3 允许描述由基本类型、结构类型和 **anytype** 类型不同值组成的列表，如表 3 中所列。列表中的值应该是源类型的值，并应该是由源类型定义的值的真子集。由该列表定义的子类型限定了子类型允许的值为列表中的那些值。

例如：

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
type float pi (3.1415926);
type charstring MyStringList ("abcd", "rgy", "xyz");
type universal charstring SpecialLetters (char(0, 0, 1, 111), char(0, 0, 1, 112), char(0, 0, 1, 113));
```

6.2.2 值域

6.2.2.0 概述

TTCN-3 允许对类型 **integer**、**charstring**、**universal charstring** 和 **float**（或这些类型的派生）值域做出规定。对 **integer** 和 **float**，值域定义的子类型限定了允许的子类型值为值域中的值，且包括该值域的上边界和下边界。在 **charstring** 和 **universal charstring** 类型的情况下，值域限定了字符串中各单独字符的允许值。边界值应根据该类型的编码字符集合表来计算有效的字符位置（如给定的位置不得为空）。上边界与下边界之间的空位置不认为是所描述值域的有效值。

例 1：

```
type integer MyIntegerRange (0 .. 255);
type float piRange (3.14 .. 3142E-3);
```

例 2：

```
type charstring MyCharString ("a" .. "z");
// 定义了一个任意长度的串类型，且串中的每个字符都在所描述的值域内。
type universal charstring MyUCharString1 ("a" .. "z");
// 定义了一个任意长度的串类型，且串中的每个字符都在从a到z的值域内，（字符编码从97到122），如“abxyz”；
// 不允许包含任何其它字符（包括控制字符）的串，如“abc2”。
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// 定义了一个任意长度的串类型，且串中的每个字符都在使用四元组所描述的值域内。
```

6.2.2.1 无限值域

为了描述一个无限整数或浮点数的值域，可以使用关键字 **infinity** 来代替一个值，以表明没有任何上边界或下边界。上边界应大于或等于下边界。

例如：

```
type integer MyIntegerRange (-infinity .. -1); // 所有负整数
```

注 — 无限的“值”有赖于实现方案。该特性的使用可能会导致可移植性问题。

6.2.2.2 列表和值域的混合

无。

注 — 该子句由第6.2.5节替代。

6.2.3 串长度限定

TTCN-3 允许对串类型的长度限制做出规定。根据使用长度边界的串类型，该长度边界具有不同的单位。在所有情况下，这些边界都应为非负的整数值（或是派生的 **integer** 值）。

例如：

```
type bitstring MyByte length(8);                // 精确的长度值8。
type bitstring MyByte length(8 .. 8);           // 精确的长度值8。
type bitstring MyNibbleToByte length(4 .. 8);   // 最小长度为4，最大长度为8。
```

表 4 描述了不同串类型的长度单位。

表4/Z.140－字段长度描述字段中使用的长度单位

类　型	长度单位
bitstring	比特
hexstring	十六进制数字
octetstring	八位字节组
character strings	字符

对上界，关键字 **infinity** 也可用于表示对长度没有上限。上限应大于等于下限。

6.2.4　字符串类型的样式子类型

TTCN-3 允许使用第 B.1.5 节中描述的字符样式，以约束 **charstring** 和 **universal charstring** 类型的允许值。这种类型约束通过在字符样式前使用 **pattern** 关键字来实现。样式表示的所有值都应是子类型的一个真子集。

注 — 样式子类型可以看作是列表约束的一种特殊形式，其列表成员不是通过列出明确的字符串来定义，而是通过一种生成列表元素的机制来定义。

例如：

```
type charstring MyString (pattern "abc*xyz");
  // MyString的所有允许值都有前缀abc和后缀xyz。

type universal charstring MyUString (pattern "*\r\n")
  // MyUString的所有允许值都以CR/LF结尾。

type charstring MyString2 (pattern "abc?\q{0,0,1,113}");
  // 会发生错误，原因是用四元组{0,0,1,113}表示的字符不是一个合法的TTCN-3字符串类型字符。

type MyString MyString3 (pattern "d*xyz");
  // 会发生错误，原因是MyString类型没有包含以字符d开头的值。
```

6.2.5　混合子类型机制

6.2.5.1 混合样式、列表和值域

integer 和 **float**（或这些类型的衍生类型）子类型定义中，允许混合使用列表和值域。不同约束的交叉不会产生错误。

例 1：

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
```

charstring 和 **universal charstring** 子类型定义中，不允许使用混合样式、列表或值域约束。

例 2：

```
type charstring MyCharStr0 ('gr', 'xyz');
  // 包含字符串gr和xyz；

type charstring MyCharStr1 ('a'..'z');
  // 包含由字母a～z组成的任意长度的字符串。

type charstring MyCharStr2 (pattern '[a-z]#(3,9)');
 //  包含由字母a～z组成的、3～9个字符长度的字符串。
```

6.2.5.2 与其它约束一起使用长度限制

bitstring、**hexstring**、**octetstring** 子类型定义中，列表和长度限制可能会在同一子类型定义中混合使用。

charstring 和 **universal charstring** 子类型定义中，允许在同一子类型定义中，在包含列表、值域或模式子类型的约束中添加长度限制。

当与其它约束混合使用时，长度限制将是子类型定义的最后一个元素。长度限制与其它子类型机制共同发挥作用（例如，类型的值集合包括由列表、值域或模式子类型和长度限制确定的值集合的公共子集）。

例如：

```
type charstring MyCharStr5 ('gr', 'xyz') length (1..9);
  // 包含字符串gr和xyz;

type charstring MyCharStr6 ( 'a'..'z') length (3..9);
  // 包含由字母a～z组成的、3～9个字符长度的字符串。

type charstring MyCharStr7 (pattern '[a-z]#(3,9)') length (1..9);
  // 包含由字母a～z组成的、3～9个字符长度的字符串；

type charstring MyCharStr8 (pattern '[a-z]#(3,9)') length (1..8);
  // 包含由字母a～z组成的、3～8个字符长度的字符串；

type charstring MyCharStr9 (pattern '[a-z]#(1,8)') length (1..9);
  // 包含由字母a～z组成的、1～8个字符长度的任何字符串；

type charstring MyCharStr10 ('gr', 'xyz') length (4);
  // 不包含任何值（空类型）。
```

6.3 结构化的类型和值

6.3.0 概述

关键字 **type** 也用于描述结构化的类型，如 **record** 类型、**record of** 类型、**set** 类型、**set of** 类型、**enumerated** 类型和 **union** 类型。

可以使用一个明确的赋值记法或一个简写的值列表记法来给出这些类型的值。

例 1：

```
const MyRecordType MyRecordValue:=                                    // 赋值记法。
{
  field1 := '11001'B,
  field2 := true,
  field3 := "A string"
}

//或者
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}       //值列表记法。
```

当使用赋值记法来描述部分值时(即仅设置一个结构变量字段子集的值),只有被赋值的字段才必须做出规定。未提到的字段隐含地不予规定。也可以使用未使用的符号“-”来使一些字段显性地不被规定。使用值列表记法时，结构中的所有字段都应使用值、未使用符号“-”或关键字 **omit** 来描述。

例 2：

```
var MyRecordType MyVariable:=                          //赋值记法。
{
  field1 := '11001'B,
  // field2 implicitly unspecified
  field3 := "A string"
}

// 或
var MyRecordType MyVariable:=                          //赋值记法。
{
  field1 := '11001'B,
  field2 := -, // field2 explicitly unspecified
  field3 := "A string"
}

// 或
var MyRecordType MyVariable:= {'11001'B, -, "A string"}     //值列表记法。
```

在相同的（紧接着的）上下文中，不允许混合使用这两种值记法。

例 3：

```
// 这是不允许的。
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "A string"}
```

无论是在赋值记法还是值列表记法中，对相关的字段，都应使用显性的 **omit** 值来省略可选的字段。对强制性字段，不能使用关键字 **omit**。当重新赋值一个之前初始化过的值时，在赋值记法中使用未用过的符号或跳过一个字段都将使相关字段保持不变。

例 4：

```
var MyRecordType MyVariable :=
{
  field1 := '111'B,
  field2 := false,
  field3 := -
}

MyVariable := { '10111'B, -, - };
// 此后，MyVariable包含{ '10111'B, false /* unchanged */, <undefined> }。

MyVariable :=
{
  field2 := true
}
// 此后，MyVariable包含{ '10111'B, true, <undefined> }。

MyVariable :=
{
  field1 := -,
  field2 := false,
  field3 := -
}
// 此后，MyVariable包含{ '10111'B, false, <undefined> }。
```

6.3.1 记录类型和值

6.3.1.0 概述

TTCN-3 支持有序的结构化类型，即 **record**。**record** 类型的元素可以是基本类型，也可以是用户定义的数据类型（如其它记录、集合或数组）。**record** 的值应与 **record** 字段的类型兼容。对 **record**，其元素标识符是该 **record** 的本地标识符，且在该 **record** 中应是唯一的（但不必是全局唯一的）。**record** 类型的常量应既不直接也不间接包含变量或模块参数作为字段值。

例 1：

```
type record MyRecordType
{
  integer              field1,
  MyOtherRecordType    field2 optional,
  charstring           field3
}

type record MyOtherRecordType
{
  bitstring    field1,
  boolean      field2
}
```

可以定义没有字段的记录（即作为一个空记录）。

例 2：

```
type record MyEmptyRecord {}
```

record 赋值基于单个元素进行。值列表记法中字段值的次序应与相关类型定义中的字段次序相一致。

例 3：

```
var integer MyIntegerValue := 1;

const MyOtherRecordType MyOtherRecordValue:=
{
  field1 := '11001'B,
  field2 := true
}

var MyRecordType MyRecordValue :=
{
  field1 := MyIntegerValue,
  field2 := MyOtherRecordValue,
  field3 := "A string"
}
```

用一个值列表来定义相同的值。

例 4：

```
MyRecordValue:= {MyIntegerValue, {'11001'B, true}, "A string"};
```

6.3.1.1 引用记录类型字段

应使用点号记法来对 **record** 类型的元素进行引用：*TypeOrValueId.ElementId*，其中，*TypeOrValueId* 解析为结构化类型或变量的名称，ElementId 解析为结构化类型中字段的名称。

例如：

```
MyVar1 := MyRecord1.myElement1;
// 如果一个记录嵌套在另一个类型中，那么它的引用看起来会像下面这样：
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 记录中的可选元素

应使用关键字 **optional** 来描述 **record** 中的可选元素。

例 1：

```
type record MyMessageType
{
  FieldType1   field1,
  FieldType2   field2  optional,
   :
  FieldTypeN   fieldN
}
```

可使用省略号来删除可选字段。

例 2：

```
MyRecordValue:= {MyIntegerValue, omit , "A string"};

// 注：下面这一赋值语句的含义不同于书写的那样，
// MyRecordValue:= {MyIntegerValue, -, "A string"};
// 而是指field2的值保持不变。
```

6.3.1.3 字段类型的嵌套类型定义

TTCN-3 支持嵌套在 **record** 定义内的记录字段类型定义。可以有新结构类型（**record**、**set**、**enumerated**、**set of** 和 **record of**）的定义，也可以有子类型约束的说明。

例如：

```
// 带有嵌套结构类型定义的记录类型
type record MyNestedRecordType
{
  record
  {
      integer nestedField1,
      float   nestedField2
} outerField1,
enumerated {
      nestedEnum1,
```

```
            nestedEnum2
        } outerField2,
        record of boolean outerField3
}

// 带有嵌套子类型定义的记录类型
type record MyRecordTypeWithSubtypedFields
{
        integer    field1 (1 .. 100),
        charstring field2 length ( 2 .. 255 )
}
```

6.3.2 集合类型和值

6.3.2.0 概述

TTCN-3 支持无序的结构化类型，即 **set**。集合类型和值与记录类似，只是 **set** 字段的排序是没有意义的。

例如：

```
type set MySetType
{
  integer        field1,
  charstring     field2
}
```

对集合而言，字段标识符是局部有效的，且在集合中应是唯一（但不必是全局唯一的）。

设置值的值列表记法不得用于 **set** 类型的值。

6.3.2.1 对集合类型字段的引用

应使用点号记法（见第 6.3.1.1 节）来引用 **set** 元素。

例如：

```
MyVar3 := MySet1.myElement1;
// 如果一个集合嵌套在另一个类型中，那么对它的引用看起来会像下面这样：
MyVar4 := MyRecord1.myElement1.myElement2;
// 注：集合类型嵌入在记录类型中，引用其带标识符‘myElement2’的字段。
```

6.3.2.2 集合中的可选元素

应使用关键字 **optional** 来描述 **set** 中的可选元素。

6.3.2.3 字段类型的嵌套类型定义

TTCN-3 支持嵌套在 **set** 定义内的集合字段类型定义，其机制与第 6.3.1.3 节中所述的记录类型类似。

6.3.3 单一类型的记录和集合

6.3.3.0 概述

TTCN-3 支持对所有元素为同一类型的记录和集合的说明，使用关键字 **of** 来表示。这些记录和集合没有元素标识符，可以认为它们分别与有序和无序的数组类似。

使用关键字 **length** 来限定 **record of** 和 **set of** 的长度。

例 1：

```
type record length(10) of integer MyRecordOfType;        // 是一个记录，正好有10个整数。
type record length(0..10) of integer MyRecordOfType;     //是一个记录，最多有10个整数。
type record length(10..infinity) of integer MyRecordOfType; // 最少10个整数的记录。
type set of boolean MySetOfType;    // 布尔值的一个无限集合。
type record length(0..10) of charstring StringArray length(12);
// 一个记录类型，最多有10个串，每个串正好12个字符。
```

record of 和 **set of** 类型的值记法应是一个值列表记法或是一个对各元素的下标记法（与用于数组的值记法相同，见第 6.5 节）。但这一般规则有一种例外情况：在定义修改的模板情况下，当也允许使用赋值记法时（见第 14.6.0 节）。

当使用值列表记法时，列表中的第一个值被赋给第一个元素，第二个列表值被赋给第二个元素，依此类推。不允许赋空值（例如，两个逗号紧挨着或之间仅有一个空格）。应该在该列表中明确地跳过或省略赋值中要省去的元素。

下标值记法既可以用在赋值符号的左边也可以用在赋值符号的右边。第一个元素的下标值应为 0，且该下标值不应超出子类型设定的长度限制。如果在赋值符号右边下标表示的元素值未被定义，那么将导致语义或运行时间错误。如果在赋值符号左边的一个下标运算符引用了一个不存在的元素，那么赋值符号右边的值将被赋予该元素，并同时创建所有比实际的下标值小的带下标的元素，但并不对这些元素赋值，使这些元素的值是未定义的。仅在中间状态允许未定义的元素（而值仍是不可见的）。发送带有未定义之元素的 **record of** 类型值将导致一个动态的测试用例错误。

例 2：

```
// 给出
type record of integer MyRecordOf;
var integer MyVar;
var MyRecordOf MyRecordVar := { 0, 1, 2, 3, 4 };

MyVar := MyRecordVar[0];
                              // “record of”值中的第一个元素（整数0）赋值给MyVar。
// 也允许下标值在赋值符号的左边：
MyRecordVar[1] := MyVar; // MyVar被赋予了第二个元素，
                         // MyRecordVar的值为{ 0, 0, 2, 3, 4 }。
// 赋值：
MyRecordVar := { 0, 1, -, 2, omit };
// 将使MyRecordVar的值变为{ 0, 1, 2 <unchanged>, 2}；
// 注：如果没有事先赋值，那么第三个元素将是未定义的。

// 赋值：
MyRecordVar[6] := 6;

// 将使MyRecordVar的值变为{ 0, 1, 2 , 2, <undefined>,   <undefined>, 6 }；
// 注：在这最后一个赋值之前第五个元素和第六个元素（下标号为4和5）没有被赋值，因此它是未定义的。
```

注 — 这就使得在一个for循环中一个元素接一个元素地拷贝**record of**值成为可能。例如，下面的函数翻转了一个**record of**值的元素：

```
function reverse(in MyRecord src) return MyRecord
     {
     var MyRecord dest;
     var integer I;
     for(I := 0; I < sizeof(src); I:= I + 1) {
         dest[sizeof(src) - 1 - I] := src[I];
     }
     return dest;
     }
```

嵌入式 **record of** 和 **set of** 类型将导致一个类似多维数组的数据结构（见第 6.5 节）。

例 3：

```
//给出
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType MyRecordOfType;

// 那么变量myRecordOfArray将具有与一个二维数组类似的属性：
var MyRecordOfType myRecordOfArray;
// 且对一个特定元素的引用将看起来像下面这样：
// （第三个‘MyBasicRecordOfType’构造的第二个元素的值）
myRecordOfArray [2][1] := 1;
```

6.3.3.1 嵌套的类型定义

TTCN-3 支持与 **record of** 或 **set of** 定义嵌套的聚合类型定义。新的结构类型定义（**record**、**set**、**enumerated**、**set of** 和 **record of**）和子类型约束规范都是可能的。

例如：

```
type record of enumerated { red, green, blue } ColorList;
type record length (10) of record length (10) of integer Matrix;
type set of record { charstring id, charstring val } GenericParameters;
```

6.3.4 枚举的类型和值

TTCN-3 支持 **enumerated** 类型。枚举类型用于对只采用值的不同命名集的类型进行建模，这些不同的值成为枚举。每个枚举都应有一个标识符。对枚举类型的操作应仅使用这些标识符，且仅限于赋值、等价和排序运算符。在枚举类型内，枚举标识符应是唯一的（但不必是全局唯一的），因而也就仅在给定类型中的上下文中是可见的。枚举标识符仅会在其它结构化类型定义中重复使用，且不会在相同作用范围层次结构分支中的同层或低层中被用作局部或全局可见性的标识符（见第 5.3.0 节中的作用范围层次结构）。

例 1：

```
type enumerated MyFirstEnumType {
  Monday, Tuesday, Wednesday, Thursday, Friday
};

type integer Monday;
// 因为类型名称具有局部或全局可见性，所以该定义是非法的。

type enumerated MySecondEnumType {
  Saturday, Sunday, Monday
};
// 因为该定义在一个不同的枚举类型中重复使用了枚举标识符Monday，所以它是合法的。

type record MyRecordType {
  integer     Monday
};
// 因为该定义在一个不同的结构类型中重复使用了枚举标识符Monday作为该类型给定字段的标识符，所以它是合法的。

type record MyNewRecordType {
  MyFirstEnumType firstField,
  integer     secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// MyFirstEnumType通过MyNewRecordType的firstField元素隐性地引用。

const integer Monday := 7
// 因为该定义在相同作用范围单元中对不同的TTCN-3对象重复使用枚举标识符Monday，所以它是不合法的。
```

每个枚举可以选择性地在括号中枚举名称后定义一个分配给它的整数值。在一个 **enumerated** 类型内，每个分配的整数都应是不同的。对没有分配整数值的每个枚举，系统都将按枚举的文本次序陆续关联一个整数，从左边开始，以 0 开始，步长为 1，并跳过任何一个手工分配值所占用的数值。这些值仅可由系统用来允许使用关系运算符。

注 1 — 整数值也可由系统用于编码/解码枚举值。不过，这超出了本建议书的讨论范围（例外情况是，TTCN-3允许将编码属性关联于TTCN-3条目）。

对 **enumerated** 类型的任何实例化或值引用，将隐性地或显性地引用给定的类型。

注 2 — 如果枚举类型是用户定义的结构类型的一个元素，那么在值赋值、实例化等中，将通过给定的元素来隐性地引用枚举类型（即通过元素的标识符，或者通过值列表记法中的值位置）。

例 2：

```
// MyFirstEnumType和MySecondEnumType的有效实例化将是：
var MyFirstEnumType Today := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// 但是，由于两个枚举类型是不兼容的，因此下面的语句是非法的。
Today := Tomorrow
```

6.3.5 联合类型

6.3.5.0 概述

TTCN-3 支持 **union** 类型，**union** 类型是字段的汇集，这些字段中的每个都由一个标识符来标识。在实际的联合值中，只有一个规定的字段将永远出现。联合类型在对结构进行建模时很有用，结构可以采用已知类型的其中一个有限数值。

例如：

```
type union MyUnionType
{
  integer   number,
  charstring      string
};

// MyUnionType的一个有效实例化将是：
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;       // 通过引用字段的值记法。注：这种记法使得给出的字段就是所选的字段。
oneYearOlder := {number := age.number+1};

ageInMonths := age.number * 12;
```

用于设置值的值列表记法不得用于 **union** 类型的值。

6.3.5.1 对联合类型字段的引用

应通过点号记法来引用 **union** 类型字段（见第 6.3.1.1）节。

例如：

```
MyVar5 := MyUnion1.myChoice1;
// 如果一个联合类型嵌套在另一个类型中，那么对它的引用看起来可以像如下格式：
MyVar6 := MyRecord1.myElement1.myChoice2;
// 注：被引用的、字段带‘myChoice2’标识符的联合类型嵌入一个记录类型中。
```

6.3.5.2 可选性和联合

union 类型不允许使用可选字段，这意味着，关键字 **optional** 不得与 **union** 类型一起使用。

6.3.5.3 字段类型的嵌套类型定义

TTCN-3 支持嵌套在联合定义中的联合字段类型定义，类似于第 6.3.1.3 节中所述的、有关记录类型的机制。

6.4 anytype类型

特殊类型 **anytype** 定义为 TTCN-3 模块中所有已知数据类型和地址类型的联合的简写，术语“已知类型”的定义在第 3.1 节中给出，也就是说，**anytype** 将包括所有已知数据类型，但不包括 **port**、**component** 和 **default** 类型。如果在该模块中明确定义了 **address** 类型，那么模块将包括 **address** 类型。

anytype 的字段名应由相应的类型名称唯一确定。

注 1 — 作为该要求的一个结果，带冲突名称的输入类型（或者在输入模块中带一个定义标识符，或者带一个自第三个模块输入的标识符）无法通过输入模块的**anytype**获得。

例如：

```
// anytype的一个有效用法将是：
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

MyVarOne.integer := 34;
MyVarTwo := {integer := MyVarOne.integer + 1};

MyVarThree := MyVarOne.integer * 12;
```

对各个模块而言，**anytype** 都是在本地定义的，（与其它预定义的类型一样）不能直接从另一个模块引入。不过，**anytype** 类型的一个用户定义类型可以被另一个模块引入。这就使得该模块的所有类型都被引入。

注 2 — **anytype**类型的用户定义类型“包含”所有引入模块中的类型，在该模块中对它进行声明。引入这样一个用户定义类型到一个模块中可能引起副作用，因此，对这种情况应谨慎处理。

6.5 数组

与许多编程语言相同，在 TTCN-3 中不认为数组是类型，而是在变量声明中对它们加以描述。数组可以声明为单维或多维的。数组维数应利用常量表达式来描述，它将按一个正 **integer** 值来计算。

例 1：

```
var integer MyArray1[3];      // 例示了一个带有3个元素的整数数组，其下标为0到2。
var integer MyArray2[2][3];   // 例示了一个具有2×3个元素的二维整数数组，其下标为(0,0)到(1,2)。
```

数组的维数应该用一个结果为整型值的常量表达式来描述，也可以使用作用范围来描述，在后一种情况下，该作用范围上下界的值定义了该数组元素下标的上下界值。

数组元素通过下标记法（[ ]）来访问，它必须在数组范围内规定一个有效的下标。多维数组的单个元素可以通过重复使用下标记法来访问。访问数组范围之外的元素将产生一个编译错误或测试用例错误。

例 2：

```
MyArray1[1] := 5;
MyArray2[1][2] := 12;

MyArray1[4] := 12;     // 错误：下标必须在0～2之间。
MyArray2[3][2] := 15; // 错误：第一个下标必须为0或1。
```

也可以使用值域来规定数组维数。在这些情况下，值域的下边界值和上边界值定义了下边界下标值和上边界下标值。

例 3：

```
var integer MyArray3[1 .. 5];              // 例示了一个有5个元素的整数数组，下标为1到5。
MyArray3[1] := 10; // 最小下标。
MyArray3[5] := 50; // 最大下标。

var integer MyArray4[1 .. 5][2 .. 3 ];    // 例示了一个有5×2个元素的二维整数数组，下标为（1,2）到（5,3）。
```

数组元素的值应兼容于对应的变量声明，这些值可以使用值列表记法来对它们分别赋值，或者使用下标记法或多次使用值列表记法来赋值。当使用值列表记法时，该列表的第一个值被赋予数组的第一个元素（该元素的下标值为 0），第二个值被赋予第二个元素，依此类推。不考虑赋值的元素应在列表中显性地跳过或省略。赋值给多维数组时，要赋值的每一维都应解析到花括号中的一个值集合中去。当规定多维数组的值时，最左边的维数对应值的最外层结构，最右边的维数对应值的最内层结构。允许使用多维数组的数组片段，即当数组值的下标数小于对应数组定义中的维数时。数组片段的下标应从左到右对应数组定义的维数（即数组片段的第一个下标对应定义的第一维）。数组片段下标应与相关数组定义维数相一致。

例 4：

```
MyArray1[0]:= 10;
MyArray1[1]:= 20;
MyArray1[3]:= 30;

// 或者使用一个值列表。
MyArray1:= {10, 20, -, 30};

MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
  // 完全定义了数组值。
```

```
var integer MyArray5[2][3][4] :=
{
  {
    {1, 2, 3, 4}, //  为MyArray5片段[0][0]赋值。
    {5, 6, 7, 8}, //  为MyArray5片段[0][1]赋值。
    {9, 10, 11, 12} //  为MyArray5片段[0][2]赋值。
  }, // 终止对MyArray5片段[0]的赋值。
  {
    {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
  } //  为MyArray5片段[1]赋值。
};

MyArray4[2] := {20, 20};
  // 获得 {{1, 2}, {3, 4}, {20, 20}, {7, 8}, {9, 10}};
MyArray5[1] := { {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};
  // 获得 {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
  //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

MyArray5[0][2] := {3, 3, 3, 3};
  // 获得 {{{1, 2, 3, 4}, {5, 6, 7, 8}, {3, 3, 3, 3}},
  //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

var integer MyArrayInvalid[2][2];
MyArrayInvalid := { 1, 2, 3, 4 }
  // 无效，因为值记法的维数与定义的维数不对应。
MyArrayInvalid[2] := { 1, 2 }
  // 无效，因为片段的下标应为0 或 1。
```

注 — 使用多维数据结构的一个替代方式是通过使用record、record of、set或set of类型。

例 5：

```
// 给出
type record MyRecordType
{
  integer   field1,
  MyOtherStruct   field2,
  charstring      field3
}
// MyRecordType的一个数组可以是：
var MyRecordType myRecordArray[10];
//对一个特定元素的引用可以看起来像如下格式：
myRecordArray[1].field1 := 1;
```

6.6 递归类型

适用的 TTCN-3 类型定义可以是递归的。不过，用户应确保所有的类型递归都是可解析的，且不会发生任何无限的递归。

6.7 类型兼容性

6.7.0 概述

通常，在赋值、实例化和比较时，TTCN-3 要求值的类型兼容。

出于本节的目的，将待赋予的、作为参数传递的实际值等称为“b”。值“b”的类型称为类型“B”。用于获得值“b”实际值的形参类型称为类型“A”。

6.7.1 非结构化类型的类型兼容性

对简单基本类型以及比特串、十六进制串和八位字节串类型的变量、常量、模板等，如果类型“B”解析为与类型“A”相同的源类型（如 **integer**）且不违反类型“A”的子类型机制（如值域、长度限制）时，值“b”兼容于与型“A”。

例如：

```
// 给出
type integer MyInteger(1 .. 10);
:
var integer x;
var MyInteger y;
```

```
// 那么
y := 5;  // 是一个有效的赋值

x := y;
// 是一个有效的赋值，因为y与x有相同的源类型，且不违反任何有关子类型的要求。

x := 20; // 是一个有效赋值
y := x;
// 不是一个有效的赋值，因为x的值超出了MyInteger的值域。

x := 5;  // 是一个有效赋值
y := x;
// 是一个有效的赋值，因为x的值现在MyInteger的值域内。

// 给出
type charstring MyChar length (1);
type charstring MySingleChar length (1);
var MyChar myCharacter;
var charstring myCharString;
var MySingleChar mySingleCharString := "B";

// 那么
myCharString := mySingleCharString;
// 是一个有效的赋值，因为长度限制为1的字符串兼容于字段串。
myCharacter := mySingleCharString;
// 是一个有效的赋值，因为两个单字符长度的字符串是兼容的。

// 给出
myCharString := "abcd";

// 那么
myCharacter := myCharString[1];
// 是有效的，因为r.h.s.记法从字符串存取一个单个元素。

// 给出
var charstring myCharacterArray [5] := {"A", "B", "C", "D", "E"}

// 那么
myCharString := myCharacterArray[1];
// 是有效的，并将值“B”赋予myCharString;
```

对 **charstring** 类型的变量、常量、模板等，值 'b' 兼容于 **universal charstring** 类型 'A'，除非它违反类型“A”的任何类型约束规范（值域、列表或长度）。

对 **universal charstring** 类型的变量、常量、模板等，如果在类型 **charstring 中，**值 'b' 中所用的所有字符有其对应的字符（即使用相同字符代码的、相同的控制或图形字符），且它不违反类型“A”的任何类型约束规范（值域、列表或长度），那么值 'b' 兼容于 **charstring** 类型 'A'。

6.7.2 结构化类型的类型兼容性

6.7.2.0 概述

在结构化类型情况下（**enumerated** 类型除外），如果类型“B”的有效值结构与类型“A”是兼容的，那么类型“B”的值“b”与类型“A”兼容，在这种情况下，允许进行赋值、实例化和比较。

6.7.2.1 枚举类型的类型兼容性

枚举类型与其它基本类型或结构化类型从不兼容（也就是说，对枚举类型，要求强类型机制）。

6.7.2.2 record和record of类型的类型兼容性

对 **record** 类型，如果按定义的文本次序，字段的数量和可选性是一致的；那么有效值结构是兼容的；各个字段的类型是兼容的；并且值“b”的各个现有字段值与类型“A”中其对应字段的类型是兼容的。值“b”中各个字段的值赋给类型“A”值中的对应字段。

例 1：

```
// 给出
type record AType {
      integer     a(0..10)    optional,
      integer     b(0..10)    optional,
```

```
      boolean        c
}

type record BType {
      integer        a       optional,
      integer     b(0..10)   optional,
      boolean        c
}

type record CType {     // 带有不同字段名的类型
      integer        d  optional,
      integer        e  optional,
      boolean        f

}

type record DType {     // 带有可选字段c的类型
      integer        a  optional,
      integer        b  optional,
      boolean        c  optional
}

type record EType {     // 带有额外字段d的类型
      integer        a  optional,
      integer        b  optional,
      boolean        c,
        float        d  optional
}

      var AType MyVarA := { -, 1, true};
      var BType MyVarB := { omit, 2, true};
      var CType MyVarC := { 3, omit,  true};
      var DType MyVarD := { 4, 4, true};
      var EType MyVarE := { 5, 5, true, omit};

      // 那么

      MyVarA := MyVarB;       // 是一个有效的赋值，MyVarA的值为( a := <undefined>, b:= 2, c:= true)。
      MyVarC := MyVarB;       // 是一个有效的赋值，MyVarC的值为( d := <undefined>, e:= 2, f:= true)。
      MyVarA := MyVarD;       // 因为字段的可选性不匹配，所以不是一个有效的赋值。
      MyVarA := MyVarE;       // 因为字段的数目不匹配，所以不是一个有效的赋值。

      MyVarC := { d:= 20 };  // MyVarC的实际值为{ d:=20, e:=2,f:= true }。
      MyVarA := MyVarC        // 不是一个有效的赋值，因为MyVarC的字段'd'违反AType类型字段'a'的子类型规定。
```

对 **record of** 类型和数组，如果其部件类型是兼容的，且类型“B”的值“b”不违反 **record of** 类型的任何长度子类型机制或类型“A”的数组维数，那么有效值结构是兼容的。值“b”的元素值应依次赋值给类型“A”的实例，包括未定义的元素。

如果 **record of** 类型和单维数组的有效值结构是兼容的，且 **record of** 类型“B”的值“b”的元素数目或数组“b”的维数与 **record** 类型“A”的元素数目完全相同，那么 **record of** 类型和单维数组与 **record** 类型是兼容的。在确定兼容性时，**record** 类型字段的可选性并不重要，也就是说，它不影响字段的计数（这就意味着，可选字段在计数时总是包括在内）。**record of** 类型元素值或数组赋值给 **record** 类型的实例应按照对应 **record** 类型定义的文本次序进行，包括未定义的元素。如果一个带未定义值的元素被赋给了 **record** 类型的一个可选字段，那么将引起省略该可选元素。将一个带未定义值的元素赋给一个 **record** 类型强制性元素的尝试将产生一个错误。

注 — 如果**record of**类型没有长度限制或者长度限制超出了相比较**record**类型的元素数目，且**record of**类型任何已定义元素的下标小于或等于**record**类型元素数目减1，那么总能满足兼容性的要求。

如果未违反 **record of** 类型的任何长度限制，或者数组的维数大于或等于 **record** 类型的元素数目，那么 **record** 类型的值也可以赋给 **record of** 类型的一个实例或一个单维数组。在 **record** 值中丢失的可选元素应作为带未定义值的元素来赋值。

例 2：

```
// 给出
type record HType {
  integer a,
  integer b optional,
  integer c
}

type record of integer IType

var HType MyVarH := { 1, omit, 2};
var IType MyVarI;
var integer MyArrayVar[2];

// 那么

MyArrayVar := MyVarH;
// 是一个有效的赋值，因为类型MyArrayVar和HType是兼容的。

MyVarI := MyVarH;
// 是一个有效的赋值，因为类型兼容，且未违反任何子类型规定。

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// 不是一个有效的赋值，因为Htype的强制性字段“c”没有收到任何值。
```

6.7.2.3 set和set of类型的类型兼容性

仅 **set** 类型与其它 **set** 类型和 **set of** 类型兼容。对 **set** 类型和 **set of** 类型，应使用与 **record** 和 **record of** 类型相同的兼容性规则。

注 1 — 这就意味着，尽管不知道元素在发送和接收时的次序，但在为**set**类型决定类型兼容性时，类型定义中字段的文本次序为决定性因素。

注 2 — 在**set**值中，字段的次序可以是任意的；不过，这并不影响类型的兼容性，因此字段名称明确确定了相关**set**类型的哪些字段对应哪些**set**值字段。

例如：

```
// 给出
type set FType {
  integer aoptional,
  integer boptional,
  boolean c
}

type set GType {
  integer d   optional,
  integer e   optional,
  boolean f
}

var FType MyVarF := { a:=1, c:=true };
var GType MyVarG := { f:=true, d:=7};

// 那么

MyVarF := MyVarG; // 是一个有效的赋值，因为类型Ftype和Gtype是兼容的。

MyVarF := MyVarA; // 不是一个有效赋值，因为MyVarA为记录类型。
```

6.7.2.4 子结构之间的兼容性

本节中为结构化类型兼容性定义的规则对这些类型的子结构来说也是有效的。

例如：

```
// 给出
type record JType {
      HType H,
      integer  b optional,
      integer  c
}
```

```
var JType MyVarJ

// 如果考虑上面的声明，那么

MyVarJ.H := MyVarH;
// 是一个有效的赋值，因为Jtype的字段H的类型与Htype是兼容的。

MyVarI := MyVarJ.H;
// 是一个有效的赋值，因为Itype和Jtype的字段H的类型是兼容的。
```

6.7.3 部件类型的类型兼容性

在两种不同条件下需要考虑部件类型的类型兼容性。

1） 部件引用值与部件类型的兼容性（例如，当作为实参将部件引用传给一个函数或可选步骤时，或者当为一个不同部件类型变量赋予部件引用值时）：如果“A”的所有定义在“B”中都有同样的定义，那么部件类型“B”的部件引用“b”兼容于部件类型“A”。

2） 运行兼容性：如果“A”的所有定义在“B”中都有同样的定义，那么在其运行中引用部件类型“A”的函数或可选步骤可以在类型“B”的部件实例上予以调用或启动。

基于以下规则来确定“A”中的定义与“B”的定义是否一致：

- 对端口实例，类型和标识符都应一致。
- 对定时器，标识符应是一致的，或者两个都应具有等同的起始周期，或者两个都没有起始周期。
- 对变量实例和常量定义，标识符、类型和初始值应是一致的（在变量情况下，这意味着，或者在两个定义中都不存在，或者相同）。
- 对局部样板定义，标识符、类型、形参列表和赋值样板或样板字段值应是一致的。

6.7.4 通信操作的类型兼容性

通信操作（见第 23 节）**send**、**receive**、**trigger**、**call**、**getcall**、**reply**、**getreply** 和 **raise** 是较弱类型兼容性规则的例外，它们要求强类型机制。直接用作这些操作参数的值类型或模板也必须在相关的端口类型定义中明确进行定义。在 **receive** 或 **trigger** 操作中，也使用强类型机制来存储接收到的值、地址或部件引用。

6.7.5 类型转换

如果需要把一个类型的值变换为另一个不是自相同源类型派生而来的类型的值，那么将使用附件 C 中定义的预定义转换函数，或者使用用户定义的函数。

例如：

```
// 使用预定义函数int2hex把一个整数值转换为一个hexstring值
MyHstring := int2hex(123, 4);
```

7 模块

7.0 概述

模块是 TTCN-3 的基本构造块。例如，一个模块可以定义一个完整的、可执行的测试套件或仅仅是一个库。一个模块由定义部分（可选的）和模块控制部分（可选的）组成。

注 — 术语“测试套件”与一个完整的、包含测试用例和控制部分的TTCN-3模块是同义的。

7.1 模块的命名

模块名具有 TTCN-3 标识符的格式。此外，模块说明可以带一个可选的属性，它由 **language** 关键字确定，用于确定 TTCN-3 语言的版本，在当中来对模块进行说明。目前，对符合 TTCN-3 2001 版本要求的模块说明，支持以下语言串：“TTCN-3:2001”；对符合 TTCN-3 2003 版本要求的模块说明，支持以下语言串：“TTCN-3:2003”；对符合 TTCN-3 2006 版本要求的模块说明，支持以下语言串：“TTCN-3:2005”；

注 — 模块标识符是模块的非正式文本名。

例如：

```
module SIPTestSuite language "TTCN-3:2003"
{ … }
```

7.2 模块参数

7.2.0 概述

模块参数列表定义了由测试环境在运行时间提供的一个值集合。在测试执行时，将把这些值当作常量。通过在关键字 **modulepar** 后说明类型和列出其标识符来声明模块参数。如果地址类型在相关模块中明确定义，那么模块参数只能是类型地址。模块参数应仅在模块定义部分中予以声明。允许模块参数声明多次出现，但每个参数只能声明一次（即不允许模块参数的重复定义）。

例如：

```
module MyModulewithParameters
{
  modulepar integer TS_Par0, TS_Par1;
  modulepar boolean TS_Par2;
  modulepar hexstring TS_Par3;
}
```

7.2.1 模块参数的缺省值

描述模块参数的缺省值是允许的，这将通过在模块参数列表中的一个赋值来完成。一个缺省值可以只是一个字面值，并且只能在参数声明处进行赋值。如果测试系统不为给定的参数提供一个实际的运行时间值，那么在测试执行期间应使用缺省值；否则的话，将使用由测试系统提供的实际值。

例如：

```
module MyModuleDefaultParameter
{
  modulepar integer TS_Par0 := 0, TS_Par1;
  modulepar boolean TS_Par2 := true;
  :
}
```

7.3 模块定义部分

7.3.0 概述

模块定义部分描述模块的顶层定义，可以从其它模块引入标识符。用于模块定义部分中所做声明和引入声明的作用范围规则在第 5.3 节中给出。可以在 TTCN-3 模块中定义的那些语言要素列在表 1 中，模块定义可以被其它模块引入。

例如：

```
module MyModule
{ // 这个模块仅包含定义。
  :
  const integer MyConstant := 1;
  type record MyMessageType { … }
  :
  function TestStep(){ … }
  :
}
```

动态语言要素（如 **var** 或 **timer**）的声明应仅在控制部分、测试用例、函数、可选步骤或部件类型中进行。

注 — TTCN-3不支持在模块定义部分进行变量声明。这就意味着，在TTCN-3中不能定义全局变量。不过，在一个测试部件中，定义的变量可以被在该部件上运行的所有测试用例、函数等使用，在控制部分中定义的变量有能力使其值独立于测试用例的执行。

7.3.1 定义组

在模块定义部分中，定义可以被汇集在所谓的组中。可以在允许单个声明的任何地方描述一个声明组。组可以嵌套，即组可能包含其它组，这允许测试套件描述符在其它概念中来构建测试数据集或描述测试行为的函数。

需要的话，使用分组来提高可读性，并为模块添加逻辑结构。除了通过关联的 **with** 语句给出的组标识符和属性的上下文中外，组和嵌套的组没有任何作用范围。这意味着：

- 整个模块的组标识符不必是唯一的。不过，一个组的子组的所有组标识符都应是唯一的。必要的话，应使用点号记法来唯一标识组层次结构内的子组，例如，对一个特定子组的引入。
- 属性的覆盖规则在第28.4节中给出。

例如：

```
module MyModule {
   :
   // 一个定义集
   group MyGroup {
       const integer MyConst:= 1;
        :
       type record MyMessageType { … };
       group MyGroup1 { // 带定义的子组
            type record AnotherMessageType { … };
            const boolean MyBoolean := false
       }
   }

   // 一个可选步骤组
   group MyStepLibrary {
       group MyGroup1 {
                          // 子组与带定义的子组具有相同的名称
            altstep MyStep11() { … }
            altstep MyStep12() { … }
             :
            altstep MyStep1n() { … }
       }
       group MyGroup2 {
            altstep MyStep21() { … }
            altstep MyStep22() { … }
            :
            altstep MyStep2n() { … }
       }
   }
   :
}

// 在MyStepLibrary中引入MyGroup1的一个引入语句。
import from MyModule {
   group MyStepLibrary.MyGroup1
}
```

7.4 模块控制部分

模块控制部分可以包含局部定义，描述实际测试用例的执行次序（可能是重复的）。应在模块定义部分定义测试用例，并在控制部分对测试用例进行调用。

例如：

```
module MyTestSuite
{     // 这个模块包含定义…
      :
      const integer MyConstant := 1;
      type record MyMessageType { … }
      template MyMessageType MyMessage := { … }
      :
      function MyFunction1() { … }
      function MyFunction2() { … }
      :
      testcase MyTestcase1() runs on MyMTCType { … }
      testcase MyTestcase2() runs on MyMTCType { … }
      :
      // …和控制部分，因此它是可执行的
      control
      {
           var boolean MyVariable; // 局部控制变量
           :
           execute( MyTestCase1()); // 测试用例的顺序执行
           execute( MyTestCase2());
           :
      }
}
```

7.5 从模块引入

7.5.0 概述

可以使用 **import** 语句来重复使用在不同模块中说明的定义。TTCN-3 没有明确的输出构架；因此，缺省地，模块定义部分中的所有模块定义都可以被引入。可以在模块定义部分的任何地方使用 **import** 语句，但不得在控制部分使用该语句。

如果引入的定义具有属性（通过 **with** 语句方式来定义），那么也应引入各属性。用于改变引入定义属性的机制在第 28.6 节中进行描述。

注 — 如果模块具有全局属性，那么这些属性与不带这些属性的定义相关联。

例如：

```
module MyModuleA
{       // 这个模块包含定义和引入的定义
        :
        const integer MyConstant := 1;
        import from MyModuleB all;
                                    // 引入定义的作用范围对MyModuleA来说是全局的。
        import from MyModuleC {
            type MyType1, MyType2;
            template all
        }
        type record MyMessageType { … }
        :
        function MyBehaviourC()
        {
            const integer MyConstant := 2;
            // 这里不能使用引入操作。
             :
        }
        :
        control
        {   // 这里不能使用引入操作。
            :
        }
}
```

7.5.1 可引入定义的结构

TTCN-3 支持对下列定义的引入：模块参数、用户定义类型、特征、常量、外部常量、数据模板、特征模板、函数、外部函数、可选步骤和测试用例。每个定义有一个*名称*（用于定义定义的标识符，如一个函数名）、一个*规范*（如一个类型规范或一个函数的特征），在函数、可选步骤和测试用例情况下，还包括一个相关的*行为描述*。

例如：

```
                名称                    规范                                  行为描述
function    MyFunction    (inout MyType1 MyPar) return MyType2    {
                          runs on MyCompType                        const MyType3 MyConst := …;
                                                                    : // 更多的行为
                                                                  }

            规范          名称                规范
Type        record        MyRecordType        {
                                                  field1 MyType4,
                                                  field2 integer
                                                }

            规范          名称                规范
template    MyType5       MyTemplate          := {
                                                  field1 := 1,
                                                  field2 := MyConst,  // MyConst是一个模块常量
                                                  field3 := ModulePar // ModulePar是一个模块参数
                                                }
```

因为在相应的函数、可选步骤或测试用例被引入时，认为行为描述的内部对引入者来说是不可见的，所以行为描述对引入机制没有任何影响。因此，不在后续的说明中对它们进行考虑。

可引入定义的说明部分包含*本地定义*（如结构化类型定义的字段名或枚举类型的值）和*引用的定义*（如对类型定义、模板、常量或模块参数的引用）。对上述例子，这意味着：

	名　　称	本 地 定 义	引用的定义
function	MyFunction	MyPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType4, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

注 1 — 本地定义一栏指的只是在可引入定义中新定义的标识符。赋给可引入定义单个字段的值也可认为是本地定义，如在模板定义中，但它们对说明引入机制并不重要。

注 2 — 模板MyTemplate被引用的字段field1、field2和field3是MyType5的字段名，即通过MyType5来引用它们。

引用的定义也是可引入的定义，也就是说，一个被引用定义的源可能再次被构造成一个名称和一个描述部分，且描述部分也包含本地定义和引用定义。换句话说，一个可引入的定义可以从其它可引入的定义递归地构造。

TTCN-3 引入机制与可引入定义说明部分中使用的本地定义和引用定义相关。因此，表 5 说明了可引入定义可能的本地定义和引用定义。

表 5/Z.140－可引入定义可能的本地定义和引用定义

可引入的定义	可能的本地定义	可能的引用定义
模块参数		模块参数类型
用户定义的类型 （对所有）	参数名称	参数类型
枚举类型	具体值	
结构化类型	字段名称	字段类型
端口类型		消息类型、特征
部件类型	常量名称、变量名称、 定时器名称和端口名称	常量类型、变量类型、端口类型
特征	参数名称	参数类型、返回类型、异常类型
常量		常量类型
外部常量		常量类型
数据模板	参数名称	模板类型、参数类型、常量、模块参数、函数
特征模板		特征定义、常量、模块参数、函数
函数	参数名称	参数类型、返回类型、部件类型（**runs on** 子句）
外部函数	参数名称	参数类型、返回类型
可选步骤	参数名称	参数类型、部件类型（**runs on** 子句）
测试用例	参数名称	参数类型、部件类型（**runs on** 和 **system** 子句）

TTCN-3 引入机制对引入定义中引用定义的标识符与引用定义用法的必要信息予以区分。对引用定义的用法，不需要引用定义的标识符，因此它不是自动引入的。

7.5.2 使用引入操作的规则

使用引入操作时，应运用以下规则：

a） 只有模块最上层的定义可以被引入。出现在较低层次作用范围的定义（如定义在一个函数中的局部常量）不应被引入。

b） 只允许从定义源模块（即在**import**语句中引用的标识符的实际定义所在的模块）的直接引入。

c） 定义的名称和所有局部定义一起被引入。

注 1— 一个本地定义，如一个用户定义记录类型的字段名，仅在定义它的上下文中有意义，例如，一个记录类型的字段名只能用来访问该记录类型的字段，而不能超出这个范围。

d） 一个定义与引用定义使用所需的该引用定义的所有信息一起被引入。

注 2— 引入语句是传递的，例如，如果模块A从使用模块C中定义之类型引用的模块B中引入一个定义，那么该类型使用所需的相应信息将自动被引入模块A中。

e） 不自动引入引用定义的标识符。

注 3— 如果希望在引入模块中使用被引用的定义，那么应从其源模块处显性地引入。

f） 当引入一个函数、可选步骤或测试用例时，相应的行为说明和行为说明内使用的所有定义对引入模块而言仍是不可见的。

g） 禁止循环引入。

例如：

```
module ModuleONE {

  modulepar integer ModPar1, ModPar2 := 7

  type record RecordType_T1 {
     integer  Field1_T1,
     boolean  Field2_T1
  }
```

```
   type record RecordType_T2 {
      RecordType_T1    Field1_T2,   // RecordType_T1的使用
      RecordType_T1    Field2_T2,
      integer      Field3_T2
   }

   const integer MyConst := 13;

   template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { // 参数化的模板
      Field1_T2 := TempPar_T2,      // 对模板参数的引用
      Field2_T2 := {MyConst, true}, // 对模块常量的引用
      Field3_T2 := ModPar1          // 对模块参数的引用
   }

} // 结束模块ModuleONE


module ModuleTWO {

   import from ModuleONE {
      template Template_T2
   }

   // 在ModuleTWO中只有名称Template_T2和TempPar_T2是可见的。
   // 注意，标识符TempPar_T2只能用在Template_T2的上下文中，例如在提供一个实参值时。
   // 为引用定义RecordType_T2、RecordType_T1、Field1_T2、Field2_T2、Field3_T3、MyConst 和 ModPar1，
   // 引入Template_T2用法所需的所有信息（如用于类型检查目的信息），但其标识符在ModuleTWO中是不可见的。
   // 这意味着，例如，不可能在不显性引入ModuleTWO中类型RecordType_T1或RecordType_T2类型的情况下，
   // 使用这些类型的常量MyConst或声明这些类型的变量。

   import from ModuleONE {
      modulepar ModPar2
   }

   // 从ModuleONE引入的、ModuleONE的模块参数ModPar2可以当作一个整数常量来使用。

} // 结束模块ModuleTWO。



module ModuleTHREE {

   import from ModuleONE all; //从ModuleONE引入所有定义。

   type port MyPortType {
      inout RecordType_T2
   }

   type component MyCompType {
      var integer MyComponentVar := ModPar2; // 对ModuleONE一个模块参数的引用。
      port MyPortType MyPort
   }

   function MyFunction () return integer {
      return MyConst    // 返回在ModuleONE中定义的一个模块常量。
   }

   testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

      var integer MyTCVar := ModPar2; // 对ModuleONE一个模块参数的引用。

      MyPort.send(Template_T2); // 发送在ModuleONE中定义的一个模板。
      MyPort.receive(RecordType_T2 : ?) -> value MyPar;
                                                        // 接收到的值赋给输出参数MyPar。

   } // 结束测试用例MyTestCase。

} // 结束 ModuleTHREE。


module ModuleFOUR {

   import from ModuleTHREE {
      testcase MyTestCase
   }
```

```
  // 只有名称MyTestCase和MyPar在ModuleFOUR中是可见的和可用的。
  // 通过ModuleTHREE从ModuleONE引入RecordType_T2的类型信息，
  // 从ModuleTHREE引入MyCompType的类型信息。
  // MyTestCase行为部分中使用的所有定义对ModuleFOUR使用者来说仍是隐藏的。

} // 结束ModuleFOUR
```

7.5.3 空缺

7.5.4 引入单个定义

可以引入单个定义。

例如：

```
import from MyModuleA {
  type MyType1          //从MyModuleA引入一个类型定义。
}

import from MyModuleB {
  type MyType2, Mytype3, MyType4;  // 引入三个类型。
  template MyTemplate1;            // 引入一个模板。
  const MyConst1, MyConst2         // 引入两个常量。
}
```

7.5.5 引入一个模块的所有定义

可以使用模块名称旁的关键字 **all** 来引入一个模块定义部分的所有定义。如果使用关键字 **all** 来引入一个模块的所有定义，那么不应对同一 **import** 语句使用任何其它的引入形式（单个定义的引入、同一种类的引入等）。

例 1：

```
import from MyModule all;
```

如果一些声明不希望被引入，那么它们的种类和标识符应列在关键字 **except** 后一对花括号内的例外列表中。

例 2：

```
import from MyModule all except {
  type MyType3, MyType5
     // 从引入语句中排除类型声明MyType3和MyType5，
     // 但引入MyModule的所有其它声明。
}
```

也允许在例外列表中使用关键字 **all**；这将从引入语句中排除相同种类的所有声明。

例 3：

```
import from MyModule all except {
  type MyType3, MyType5;   // 从引入语句中排除两个类型。
  template all  // 从引入语句中排除在MyModule中声明的所有模板。
}
```

7.5.6 引入组

可以引入定义组。

例 1：

```
import from MyModule {
  group MyGroup
}
```

引入一个组的效果与一个列出该组所有可引入定义（包括子组）的 **import** 语句相同。

TTCN-3 的组只是用于结构化的目的，而不是作用范围单元，因此不允许直接引入子组（即定义在另一个组内的组），即不通过子组所嵌套的组来直接引入子组。如果应引入的子组名与相同模块中另一个子组名相同（见第 7.3.1 节），那么应使用点号记法来唯一标识待引入的子组。

如果不希望引入一个组的某些定义，那么应在关键字 **except** 后花括号对内的例外列表中列出这些定义的种类和标识符。

例 2：

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5
           // 从引入语句中排除类型定义MyType3和MyType5，但引入MyGroup的所有其它定义。
  }
}
```

也允许在例外列表中使用关键字 **all**，这将从引入语句排除相同种类的所有定义。

例 3：

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5;  // 从引入语句排除两个类型，以及
      template all // 从引入语句排除所有在MyGroup中定义的模板。
  }
}
```

7.5.7 引入相同种类的定义

关键字 **all** 可以用来引入一个模块中相同种类的所有定义。与关键字 **constant** 一起使用的关键字 **all** 用于确定所有常量以及在引入语句所指的模块定义部分中声明的所有外部常量。同样，与关键字 **function** 一起使用的关键字 **all** 用于确定所有函数以及在引入语句所表示的模块中定义的所有外部函数。

例 1：

```
import from MyModule {
  type all;             // 引入MyModule的所有类型。
  template all          // 引入MyModule的所有模板。
}
```

如果希望从一个给定的引入语句中排除一个种类的某些声明，那么应在关键字 **except** 后列出其标识符。

例 2：

```
import from MyModule {
  type all except MyType3, MyType5;      // 引入除MyType3和MyType5外的所有类型。
  template all                           // 引入在Mymodule中定义的所有模板。
}
```

7.5.8 处理引入中的名称冲突

所有的 TTCN-3 模块都应有其自身的名称空间，在各自的名称空间中所有的定义都应被唯一标识。由于引入，可能引起名称冲突，例如，来自不同模块的引入。应使用引入定义的模块标识符作为引入定义（它引起名称冲突）的前缀来解决名称冲突，前缀和标识符使用点号（.）隔开。

在不存在多义性的情况下，当使用引入定义时，前缀不必（但可以）出现。当定义在定义它的模块中被引用时，模块（当前模块）的标识符也可用作定义标识符的前缀。

例如：

```
module MyModuleA  {
   :
  type bitstring MyTypeA;
  import from SomeModuleC {
      type MyTypeA,     // MyTypeA是字符串类型。
           MyTypeB      // MyTypeB是字符串类型。
  }
   :
  control {
       :
      var  SomeModuleC.MyTypeA MyVar1 := "Test String"; // 必须使用前缀。
      var MyTypeA MyVar2 := '10110011'B; // 这是最初的MyTypeA。
       :
```

```
        var MyTypeB MyVar3 := "Test String"; // 不必使用前缀 …
        var SomeModuleC.MyTypeB MyVar3 := "Test String"; // … 但如果希望，它可以用前缀。
         :
    }
}
```

注 — 即便在不同模块中带相同名称的定义其实际定义是相同的，也总是假设在不同模块中带相同名称的定义是不同的。例如，引入一个已经在本地定义的类型，甚至使用相同的名称，将导致在该模块中由两个不同的类型可用。

7.5.9 处理相同定义的多个引用

在单个定义、定义组、相同种类定义等中使用 **import** 可能导致对相同定义的多次引用，这样的情况应通过系统来解决，定义应只能被引入一次。

注 — 解决此类多义性的机制（如重写和向用户发出警告）超出了本建议书的讨论范围，应由TTCN-3工具提供。

所有 **import** 语句和引入语句中的定义都应按照其出现的次序一个接一个地分别加以考虑。有必要指出的是，**except** 语句通常不从待引入列表中排除所列的定义；引入相同种类定义的所有语句可以看作是单个定义标识符等价列表的简写记法。**except** 语句仅从该单个列表中排除定义。

例如：

```
import from MyModule {
     type all except MyType3;      // 引入MyModule中除MyType3外的所有类型。
     type MyType3                  // 显性引入MyType3。
}
```

7.5.10 从非TTCN-3模块中引入定义

在从其它来源而非 TTCN-3 模块中引入定义的情况下，应使用语言说明来表示引入定义源的语言（可能与版本号一起）（如模块、包、库或者甚至是文件）。它由关键字 **language** 和一个后续的、表示语言的文本声明组成。当从一个与引入模块具有相同版本的 TTCN-3 模块中引入时，该语言说明的使用是可选的。在语言标识（包括通过省略语言说明的隐性标识）与引入定义的模块语法之间存在不兼容性时，工具应提供合理的帮助以解决该冲突。

定义了以下 TTCN-3 语言标识符：

- 'TTCN-3:2001' －与符合本建议书2001年版要求的模块一起使用（见参考资料）。
- 'TTCN-3:2003' －与符合本建议书2003年版要求的模块一起使用（见参考资料）。
- 'TTCN-3:2005' －与符合本建议书要求的模块一起使用。

例如：

```
import from MyModule language "TTCN-3:2003" {
  type MyType
}
```

注 — 设计引入机制的目的是为了允许重复使用来自其它TTCN-3或其它语言模块的定义。从其它语言（如SDL包）书写的说明中引入定义的规则可以遵循TTCN-3规则，或者可能需要单独进行定义。

8 测试配置

8.0 概述

TTCN-3 允许对并发测试配置（或简称配置）进行（动态）描述。一个配置由一个带良好定义的通信端口的互连测试部件集合和用于定义测试系统边界的明确的测试系统接口组成。

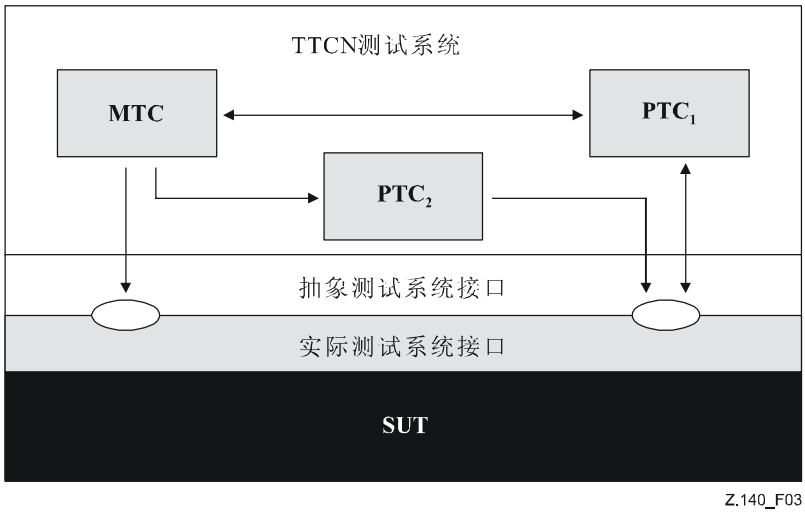


图 3/Z.140－一个典型TTCN-3测试配置的概念视图

在每个配置中都应有一个（且仅有一个）主测试部件（MTC），非主测试部件（MTC）的测试部件称为并行测试部件或 PTC。MTC 应在每个测试用例执行之初由系统自动创建，测试用例主体中定义的行为应在该部件上执行。在一个测试用例执行期间，通过显性使用 **create** 操作来动态创建其它部件。

应在 MTC 终止时结束测试用例执行，平等对待所有其它的 PTC，即在它们之间没有明确的层次结构，一个 PTC 的终止既不能终止其它 PTCs 也不能终止 MTC。当 MTC 终止时，测试系统必须停止在测试用例执行结束之时未终止的所有 PTC。

测试部件之间以及部件与测试系统接口之间的通信通过通信端口来实现（见第 8.1 节）。

用关键字 **component** 和 **port** 指示的测试部件类型和端口类型应在模块定义部分中定义，部件的实际配置以及它们之间的连接通过在测试用例行为中执行 **create** 和 **connect** 操作来实现，利用 **map** 操作来将部件端口连接至测试系统接口的端口上（见第 22.2 节）。

8.1 端口通信模型

测试部件通过它们的端口连接，也就是说，部件之间以及部件与测试系统接口之间的连接是面向端口的。每个端口都被建模为一个无限的先入先出（FIFO）队列，该队列用于存储进来的消息或是过程调用，直至拥有该端口的部件处理它们。

注 — 原则上TTCN-3端口是无限的，然而在实际的测试系统中它们可能会溢出，此时应以一个测试用例错误来看待它（见第25.2.1节）。

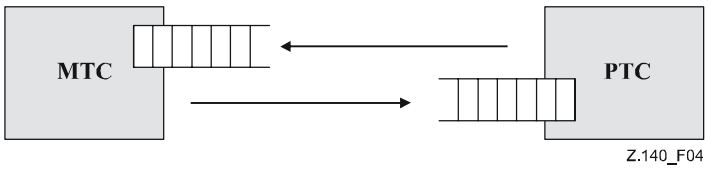


图 4/Z.140－TTCN-3通信端口模型

8.2 连接方面的限制

TTCN-3 连接是端口到端口和端口到测试系统接口的连接（见图 5）。对一个部件可以维护的连接数目没有任何限制，也允许一对多的连接（例如，图 5（g）或图 5（h））。

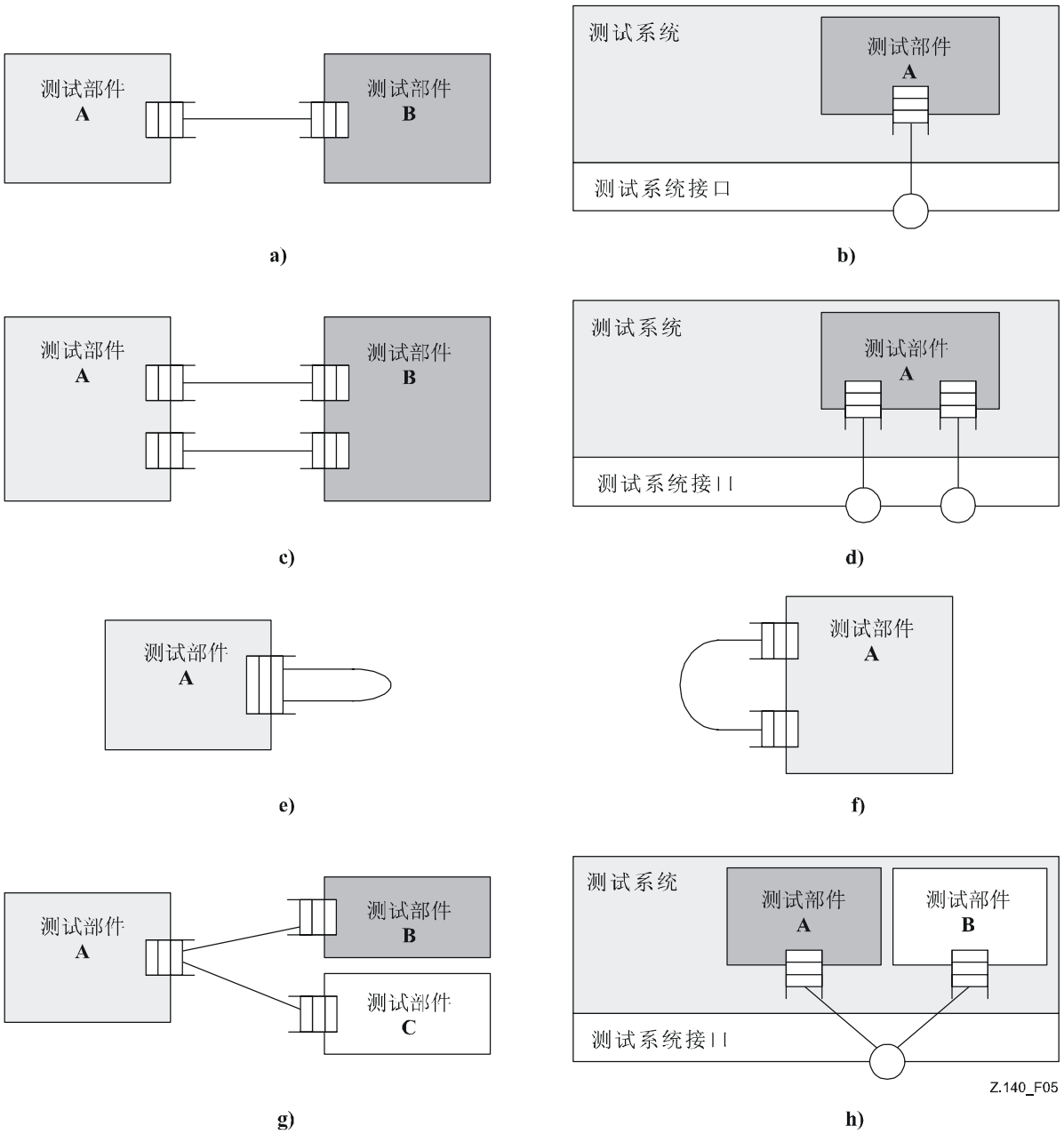


图 5/Z.140－允许的连接

不允许以下连接：

- 部件A所拥有的一个端口不应与该部件所拥有的两个或两个以上端口相连（见图6（a）和图6）e））。
- 部件A所拥有的一个端口不应与部件B所拥有的两个或两个以上端口相连（见图6（c））。
- 部件A所拥有的一个端口与测试系统接口之间只能有一个一对一的连接，这就意味着，图6（b）和图6（d）所示的连接是不允许的。
- 不允许测试系统接口内部的连接（见图6（f））。
- 被连接的端口不应被映射，而被映射的端口不应被连接（见图6（g））。

因为 TTCN-3 允许动态配置和地址，所以不可能在编译时总是检查连接方面的限制，应在运行时间上进行检查，在失败时将引起一个测试用例错误。

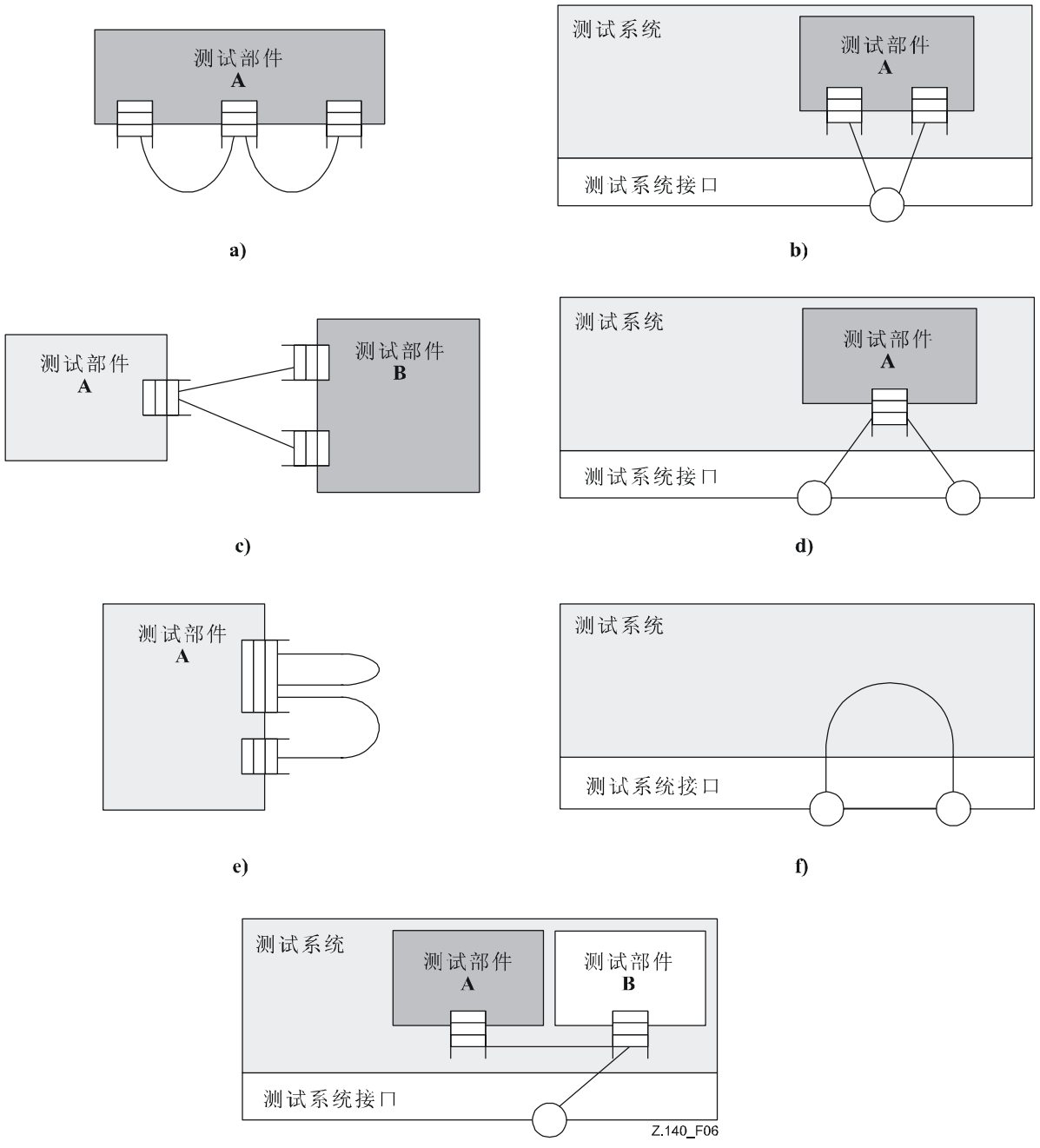


图 6/Z.140－不允许的连接

8.3 抽象测试系统接口

TTCN-3 用于测试实现方案，被测试的对象被认为是接受测试的实现方案或 IUT。IUT 可以为测试提供直接的接口，或者在被测对象是接受测试的系统或 SUT 时，它可以是系统的一部分。在最小情况下，IUT 和 SUT 是等同的。在本建议书中，术语 SUT 被作为一个通用的概念，指的是 SUT 或 IUT。

在一个实际的测试环境中，测试用例需要与 SUT 进行通信。不过，对实际物理连接的说明超出了 TTCN-3 的讨论范围。作为替代，一个良好定义的（但是抽象的）测试系统接口应与每个测试用例相关联。一个测试系统接口定义与一个部件定义是相同的，即它是一个所有可能通信端口的列表，通过它们，测试用例连接至 SUT。

测试运行期间，测试系统接口静态地定义与 SUT 连接的端口连接数目和类型。不过，测试系统接口与 TTCN-3 测试部件之间的连接实际上是动态的，通过使用 **map** 和 **unmap** 操作，可以在测试运行期间对其进行修改（见第 22.2 节和第 22.3 节）。

8.4 定义通信端口类型

8.4.0 概述

端口便于测试部件之间和测试部件与测试系统接口之间的通信。

TTCN-3 支持基于消息的端口和基于过程的端口。每个端口都应被定义为基于消息的或基于过程的（或者如第 8.4.1 节所述可以同时是二者）。通过关键字 **message** 来标识基于消息的端口，相关端口类型定义中的关键字 **procedure** 则用来标识基于过程的端口。

端口是双向的，它的方向通过关键字 **in**（输入方向）、**out**（输出方向）和 **inout**（双向）来标识的。每个端口类型定义都应有一个或多个列表来指明允许的（消息）类型集合与/或带有允许之通信方向的过程。

无论何时当在基于过程的端口的“输出”方向上定义了一个特征时，其所有 **inout** 和 **out** 参数的类型、返回类型和例外类型都将自动为该端口“输入”方向的一部分。无论何时当在基于过程的端口的“输入”方向上定义了一个特征时，其所有 **inout** 和 **out** 参数的类型、返回类型和例外类型都将自动为该端口“输出”方向的一部分。

例如：

```
// 基于消息的端口，其上允许接收MsgType1和MsgType2类型，发送MsgType3类型，并可经端口发送和接收任何整数值。
type port MyMessagePortType message
{
  in  MsgType1, MsgType2;
  out MsgType3;
  inout   integer
}

// 基于过程的端口，它允许远程调用过程Proc1、Proc2和Proc3。
//注：Proc1、Proc2和Proc3定义为特征。
type port MyProcedurePortType procedure
{
  out Proc1, Proc2, Proc3
}
```

注 — 术语“消息”指的是由模板定义的消息和源自表达式的表达式，因此，限制在一个基于消息的端口上可以使用何种消息的列表只是一个简单的类型名称列表。

8.4.1 混合型端口

将一个端口定义为基于消息的通信和基于过程的通信都是可能的，这通过关键字 **mixed** 来表示。这意味着混合型端口的列表也将是混合的，并包括特征和类型。此时，在定义中不做任何单独说明。

```
// 混合型端口，定义了具有相同名称的一个基于消息的端口和一个基于过程的端口。
// in、、out和inout列表也是混合型的：MsgType1、MsgType2、MsgType3和整数指的是混合型端口中基于消息的部分，
// Proc1、Proc2、Proc3、Proc4和Proc5指的是端口中基于过程的部分。
type port MyMixedPortType mixed
{
  in  MsgType1, MsgType2, Proc1, Proc2;
  out MsgType3, Proc3, Proc4;
  inout   integer, Proc5;
}
```

TTCN-3 中的混合型端口定义为两类端口的缩写记法，也就是说，一个基于消息的端口和一个基于过程的端口拥有相同的名称。在运行时间，通过通信操作来区分两类端口。

如果通过一个混合型端口的标识符来调用，那么用于控制端口的操作（见第 23.5 节），即 **start**、**stop** 和 **clear** 应在两个队列上（按任意次序）都执行操作。

8.5 定义部件类型

8.5.0 概述

component 类型定义了与一个部件相关联的端口。这些定义应在模块定义部分进行定义，一个部件定义中的端口名称对该部件来说是本地的，也就是说，另一个部件可以拥有相同名称的端口。相同部件的所有端口名称都应是唯一的。一个部件的定义本身并不意味着该部件在这些端口上有任何连接。

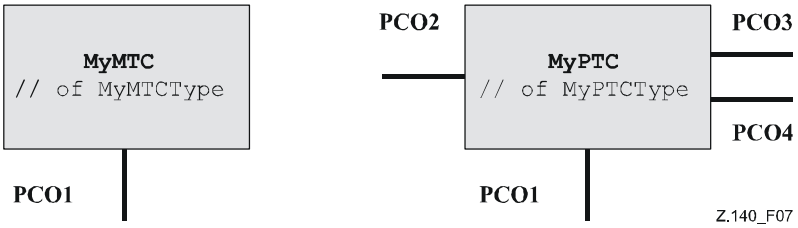


图 7/Z.140－典型的部件

例如：

```
type component MyMTCType
{
  port MyMessagePortType   PCO1
}

type component MyPTCType
{
  port MyMessagePortType     PCO1, PCO4;
  port MyProcedurePortType PCO2;
  port MyAllMesssagesPortTypePCO3
}
```

8.5.1 在一个部件中声明本地变量、常量和定时器

可以声明一个特殊部件的本地常量、变量和定时器。

例如：

```
type component MyMTCType
{
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType   PCO1
}
```

这些声明对该部件上运行的所有测试用例、函数和可选步骤都是可见的。这将使用关键字 **runs on** 来显性地说明（见第 16 节）。

部件变量和定时器与部件实例相关联，并遵循第 5.3 节中所定义的作用范围规则。一个部件的各个新实例因此将拥有它自己的变量和定时器集合，就像部件定义中所描述的那样（如果已经声明，那么包括任何初始化的值）。

注 — 当用作测试系统接口时（见第8.8节），部件不能使用在部件中声明的任何常量、变量和定时器。

8.5.2 定义带有端口数组的部件

在部件类型定义中可以定义端口数组（另见第 22.12 节）。

例如：

```
type component My3pcoCompType
{
  port MyMessageInterfaceType PCO[3]
  port MyProcedureInterfaceType PCOm[3][3]
  //  定义一个部件类型，它带有一个由3个端口组成的数组，以及一个由9个过程端口组成的二维数组。
}
```

8.5.3 部件类型的扩展

利用 **extends** 关键字，可以将部件类型定义为其它部件类型的扩展。

例 1：

```
type component MyExtendedMTCType extends MyMTCType
{
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

在这样一个定义中，新的类型定义指的是*扩展的类型*，**extends** 关键字之后的类型定义指的是*父类型*。

该定义的作用是，扩展的类型也将隐性地包含来自父类型的所有定义。因此，上述定义等同于写（因此称为*有效的类型定义*）：

例 2：

```
// 来自例子1的定义效能上等同于以下定义：
type component MyExtendedMTCType
{
  /* 来自MyMTCType的定义 */
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType PCO1

  /* 额外的定义 */
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

只要创建了定义的循环链，那么就允许扩展利用扩展方式定义的部件类型，

例 3：

```
type component MTCTypeA extends MTCTypeB { /* … */ };
type component MTCTypeB extends MTCTypeC { /* … */ };
type component MTCTypeC extends MTCTypeA { /* … */ };  // 错误 — 循环扩展。
type component MTCTypeD extends MTCTypeD { /* … */ };  // 错误 — 循环扩展。
```

当通过扩展来定义部件类型时，在源自父类型的定义与加至扩展类型的定义之间，不应存在任何名称冲突；也就是说，不应存在这样一个端口、变量、常量、定时器或模板标识符，它既在父类型中进行了声明，也在扩展类型中进行了声明。

例 4：

```
type component MyExtendedMTCType extends MyMTCType
{
  var integer MyLocalInteger; // 错误 — 已经在MyMTCType做了定义（见例子2）。
  var float MyLocalTimer;     // 错误 — 带该名称的定时器在MyMTCType中已存在。
  port MyOtherMessagePortType PCO1; // 错误 — 带该名称的端口在MyMTCType已存在。
}

type component MyBaseComponent { timer MyLocalTimer };
type component MyInterimComponent extends MyBaseComponent { timer MyOtherTimer };
type component MyExtendedComponent extends MyInterimComponent
{
  timer MyLocalTimer; // 错误 — 已经通过扩展在MyInterimComponent中做了定义。
}
```

在一个定义中，允许一个部件类型扩展若干个父类型，在定义中，这需描述为一个用逗号分隔开的类型列表：

例 5：

```
type component MyCompA extends MyCompB, MyCompC, MyCompD {
  /* MyCompA的额外定义 */
}
```

还可以通过扩展方式来定义任何父类型。

扩展类型的有效部件类型定义作为通过父类型发挥作用的所有常量、变量、定时器、端口和模板定义（如果一个父类型也通过扩展方式来定义，那么递归地进行确定）以及在扩展类型中直接声明之定义的集合来获得。有效部件类型定义应不会产生名称冲突。为了实现该条件，在扩展类型定义中所用的父类型集合内，所有的定义都应具有唯一的名称，这些名称将不同于在扩展类型中直接声明之定义的任何名称。

注 1 — 这不认为是一个不同的声明，因此，如果通过不同的父类型来使相同的定义对扩展类型发挥作用（通过不同的扩展路径），那么这不会产生任何错误。

例 6：

```
type component MyCompB { timer T };
type component MyCompC { var integer T };
type component MyCompD extends MyCompB, MyCompC {}
  // 错误 — MyCompB与MyCompC之间出现名称冲突。

// MyCompB is defined above
type component MyCompE extends MyCompB {
  var integer MyVar1 := 10;
}

type component MyCompF extends MyCompB {
  var float MyVar2 := 1.0;
}

type component MyCompG extends MyCompB, MyCompE, MyCompF {
  // 没有任何名称冲突。
  // All three parent types of MyCompG的所有三个父类型都拥有一个定义器T，或者直接拥有，
  // 或者通过MyCompB的扩展而拥有；原因是所有这些源自（直接地或通过扩展）MyCompB中所声明的定义器T，
  // 这使得这种形式的冲突是合法的。
  /* 在此做额外定义 */
}
```

带扩展的部件类型的语义通过简单地用其有效部件类型定义替代各个部件类型定义来定义，作为使用它之前的预处理步骤。

注 2 — 对部件类型兼容性，这意味着类型CT1的部件引用c，它是CT2的扩展，兼容于CT2，在其**runs on**子句中用于描述CT2的测试用例、函数和可选步骤可以在c上执行（见第6.7.3节）。

8.6 SUT内部的寻址实体

一个 SUT 可以由若干个必须分别编址的实体组成。地址数据类型是用来与端口操作结合起来寻址 SUT 实体的一种类型。当地址数据类型与 **to**、**from** 和 **sender** 一起使用时，它只能用在映射至测试系统接口的端口的接收和发送操作中。**address** 的实际数据表示通过测试套件中一个明确的类型定义来解析，或者由测试系统来进行外部解析（也就是说，**address** 类型将作为 TTCN-3 说明中的一种开放式类型）。这就允许独立于 SUT 特定的任何实际地址机制来描述抽象测试用例。

如果地址类型定义在一个模块内，那么应该仅在该 TTCN-3 模块内生成明确的 SUT 地址。如果该类型不是在一个模块内定义，那么明确的 SUT 地址应只能在消息字段中作为参数来传递或接收，或者作为远程过程调用的参数来传递或接收。

另外，可以使用特殊值 **null** 来指明一个未定义的地址，例如，用于地址类型变量的初始化。

例如：

```
// 把类型整数与开放类型地址相关联。
type integer address;
 :
// 用null初始化新的地址变量。
var address MySUTentity := null;
 :
// 接收一个地址值，并把它赋值给变量MySUTentity。
PCO.receive(address:*) -> value MySUTentity;
 :
// 使用接收到的地址来发送模板MyResult。
PCO.send(MyResult) to MySUTentity;
 :
// 使用接收到的地址来接收确认模板。
PCO.receive(MyConfirmation) from MySUTentity;
```

8.7 部件引用

部件引用对在一个测试用例执行期间创建的测试部件是唯一的引用。该唯一的部件引用在部件创建之时由测试系统产生，也就是说，部件引用是 **create** 操作的结果（见第 22.1 节）。另外，部件引用由预定义操作 **system**（返回标识测试系统接口端口的部件引用）、**mtc**（返回 MTC 的部件引用）和 **self**（返回调用 **self** 之部件的部件引用）返回。

在配置操作 **connect**、**map** 和 **start** 操作（见第 22 节）中使用部件引用来建立测试配置，并出于寻址目的，在连接至测试部件而非测试系统 **interface** 端口的通信操作的 **from**、**to** 和 **sender** 部分中使用部件引用（见第 23 节和图 5）。

此外，特殊值 **null** 可用来指示一个未定义的部件引用，例如用于处理部件引用的变量初始化。

部件引用的实际数据表示方法应由测试系统来进行外部解析，这允许独立于任何实际的 TTCN-3 运行时间环境来描述抽象测试用例；换句话说，也就是 TTCN-3 并不对有关测试部件处理和标识的测试系统实现方案加以限制。

注 — 一个部件引用包括部件类型信息。例如，这意味着，用于处理部件引用的变量在其声明中必须使用对应的部件类型名称。

例如：

```
// 一个部件类型定义
type component MyCompType {
  port PortTypeOne PCO1;
  port PortTypeTwo PCO2
}

// 为处理MyCompType类型部件的引用而声明一个变量，并创建该类型的一个部件。
var MyCompType MyCompInst := MyCompType.create;

// 在配置操作中部件引用的使用。
// 总是指的是上面已创建的部件。
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self)); // self作为参数传递给MyBehavior

// 在from-和to-子句中的部件引用用法。
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:?) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MPCO2.send(integer:5) to MyCompInst;

// 下面的例子解释了端口PCO1上一对多连接的情况，此时可以从不同类型CompType1、CompType2和CompType3
// 的若干部件处接收类型M1的值，且需要重新得到发送者。在这种情况下，可以使用以下方案：
 :
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 := null;
var MyCompType3 MyInst3 := null;
 :
alt {
   [] PCO1.receive(M1:?) from MyInst1 -> value MyMessage sender MyInst1 {}
   [] PCO1.receive(M1:?) from MyInst2 -> value MyMessage sender MyInst2 {}
   [] PCO1.receive(M1:?) from MyInst3 -> value MyMessage sender MyInst3 {}
}
 :
MyResult := MyMessageHandling(MyMessage);// 得自函数的一些结果。
 :
if (MyInst1 != null) {PCO1.send(MyResult) to MyInst1};
if (MyInst2 != null) {PCO1.send(MyResult) to MyInst2};
if (MyInst3 != null) {PCO1.send(MyResult) to MyInst3};
 :
```

8.8 定义测试系统接口

从概念上讲，由于部件类型定义和测试系统接口定义具有相同的形式（都是定义可能的连接点的端口集合），因此部件类型定义用于定义测试系统接口。

注 — 用作测试系统接口的、在部件类型中声明的变量、定时器和常量，将没有任何效用。

```
type component MyISDNTestSystemInterface
{
  port MyBchannelInterfaceType     B1;
  port MyBchannelInterfaceType     B2;
  port MyDchannelInterfaceType     D1
}
```

通常，定义测试系统接口的部件类型引用应与使用多个测试部件的每个测试用例都相关联。当测试用例开始执行时，测试系统接口的端口应与 MTC 一起由系统自动实例化。

返回测试系统接口部件引用的操作为 **system**，该操作将用于寻址测试系统的端口。

例如：

```
map(MyMTCComponent:Port2, system:PCO1);
```

在测试执行期间，MTC 是唯一实例化的部件时，测试系统接口不需要与测试用例相关联。在这种情况下，与 MTC 相关联的部件类型定义隐性地定义了相应的测试系统接口。

9 常量声明

可以在模块定义部分、部件类型定义、模块控制部分、测试用例、函数和可选步骤中声明和使用常量。常量定义由关键字 **const** 标识。常量不得为端口类型。应在声明位置处为常量赋值。

注 — 唯一能赋给缺省常量和部件类型的值为特殊值**null**。

例 1：

```
const integer MyConst1 := 1;
const boolean MyConst2 := true, MyConst3 := false;
```

可以在模块内给常量赋值，也可以在模块外进行赋值。后一种情况用关键字 **external** 来表示外部常量声明。

例 2：

```
external const integer MyExternalConst; // 外部常量声明。
```

一个外部常量可以具有任意类型，除了端口类型、缺省类型或部件类型，以及模块中已知的类型；即应是模块中定义的源类型或用户定义的类型，或者从另一个模块中引入的类型。从该类型到一个外部常量的外部表示的映射，以及如何将一个外部常量值传入模块的机制超出了本建议书的讨论范围。

10 变量声明

10.0 概述

变量可以是简单的基本类型、基本串类型、结构类型、特殊的数据类型（包括源自这些类型的子类型），以及地址、部件或缺省类型。

注 — 结构和部件类型变量只能依据用户定义的类型进行声明。

变量可以在模块控制部分、测试用例、函数和可选步骤中进行声明和使用。此外，变量还可以在部件类型定义中进行声明，并在测试用例、可选步骤以及在给定部件类型上运行的函数中使用这些变量。变量不得在模块定义部分进行声明或使用（也就是说，TTCN-3 中不支持全局变量）。

除赋值符号左侧外，在其它地方使用未初始化或未完全初始化的变量，或者作为实参传递给输出形参，将产生一个错误。

10.1 值变量

值变量通过 **var** 关键字来声明，后跟一个类型标识符和一个变量标识符。可以在声明中赋给一个初始值。值变量只能保存值，可以在赋值符号右侧、赋值符号左侧和表达式中使用，在函数主体中后跟 **return** 关键字，在其头部带一个返回子句，并可以传递给值和模板类型形参。

例如：

```
var integer MyVar0;
var integer MyVar1 := 1;
var boolean MyVar2 := true, MyVar3 := false;
```

10.2 模板变量

模板变量通过 **var template** 关键字来声明，后跟一个类型标识符和一个变量标识符。可以在声明中赋给一个初始内容。值之外，模板变量还能保存匹配机制（见第 14.3 节）。它们可以在赋值符号右侧和赋值符号左侧使用，在函数主体中后跟 **return** 关键字，在其头部定义一个模板类型返回子句，并可以作为实参传递给模板类型形参。当在赋值符号右侧使用时，它们不得是 TTCN-3 运算符的操作数（见第 15 节），在赋值符号左侧的变量也应是一个模板变量。还允许它将一个模板实例赋值给一个模板变量或模板变量字段。

注 — 类似全局和局部模板，应对模板变量做全面描述，以便在发送和接收操作中使用。

例如：

```
template MyRecord MyTempl ( template boolean par_bool ) :=
  { field1 := par_bool, field2 := * }
:
function Myfunc () return template MyRecord {
  var template integer MyVarTemp1 := ?;
  var template MyRecord MyVarTemp2 := { field1 := true, field2 := * },
               MyVarTemp3 := { field1 := ?, field2 := MyVarTemp1 };
  MyVarTemp2 := MyTempl (?);
:
  return MyVarTemp2
}
```

不允许它直接对模板变量应用 TTCN-3 操作，允许它使用点号记法和索引记法来检查和修改模板变量字段。当这些记法试图延伸至匹配机制之外的字段时，所用规则在第 14.3.1 节中给出。

11 定时器声明

11.0 概述

定时器可以在模块控制部分、测试用例、函数和可选步骤中进行声明和使用。此外，定时器还可以在部件类型定义中进行声明，并在测试用例、函数以及在给定部件类型上运行的可选步骤中使用这些变量。声明一个定时器时，可以选择给它赋一个缺省的持续时间值。如果没有指定任何其它值，那么该定时器将使用该缺省值来开始计时。定时器的值是一个非负的 **float** 值（即大于或等于 0.0），以秒为基本单位。

例 1：

```
timer MyTimer1 := 5E-3;  // 声明定时器MyTimer1，缺省值为5ms。

timer MyTimer2;   // 声明定时器MyTimer2，没有赋缺省值，
                  // 将在定时器启动之时赋予一个值。
```

除了声明单个的定时器外，还可以声明定时器数组。定时器数组中元素的缺省持续时间使用一个值数组来赋值。缺省的持续时间赋值应使用数组值记法，如第 6.5 节中所述。如果对定时器缺省持续时间赋值时希望跳过定时器数组的某些元素，那么应显性地使用未用的符号（“-”）来声明。

例 2：

```
timer t_Mytimer1[5] := { 1.0, 2.0, 3.0, 4.0, 5.0 }
  //  定时器数组的所有元素都有一个缺省的周期。

timer t_Mytimer2[5] := { 1.0, -, 3.0, 4.0, 5.0 }
  //  第二个定时器(t_Mytimer2[1]) 没有缺省的周期。
```

11.1 作为参数的定时器

定时器只能通过引用传递给函数和可选步骤。传入函数或可选步骤的定时器在函数或可选步骤的行为定义中是已知的。

作为参数由引用传入的定时器可以像任何其它定时器一样来使用；即它们不必被声明。一个已经启动了的定时器也可以传递给函数或可选步骤。定时器继续运行，也就是说，它不会在暗中被停止。因此，在传入定时器的函数或可选步骤内，可以对可能发生的超时事件进行处理。

例如：

```
// 在形参列表中带有一个定时器的函数定义。
function MyBehaviour (timer MyTimer)
{  :
   MyTimer.start;
   :
}
```

12 消息声明

TTCN-3 的其中一个关键元素是在测试配置所定义的通信端口上收发复杂消息的能力。这些消息可以是与测试 SUT 或测试配置特定的内部协调和控制消息一起明确进行考虑的消息。

注 — 在TTCN-2中，这些消息为抽象服务元语（ASP）、协议数据单元（PDU）和协调消息。在TTCN-3核心语言中，消息没有任何这种语法或语义上的区别，从这个意义上讲它是通用的。

13 声明过程特征

13.0 概述

对基于过程的通信，过程特征（或简称特征）是必需的。基于过程的通信可以用于测试系统内的通信，即测试部件之间或测试系统与 SUT 之间的通信。在后一种情况中，一个过程可以在 SUT 中被调用（即测试系统执行该调用）或在测试系统中被调用（即 SUT 执行该调用）。对所有用到的过程，即用于测试部件之间通信的过程、SUT 调用的过程和测试系统调用的过程，都要在 TTCN-3 的模块中定义完整的过程 **signature**。

13.1 阻塞和非阻塞通信的特征

TTCN-3 支持阻塞的和非阻塞的、基于过程的通信。对非阻塞的通信，特征定义将使用关键字 **noblock**，只能有 **in** 参数（见第 13.2 节），不得有任何返回值（见第 13.3 节），但可能出现异常（见第 13.4 节）。缺省地，假定不带 **noblock** 关键字的特征定义用于阻塞的、基于过程的通信。

例如：

```
signature MyRemoteProcOne ();
                                        // MyRemoteProcOne将用于阻塞的、基于过程的通信。
                                        // 它既没有参数也没有返回值。

signature MyRemoteProcTwo () noblock;
                                        // MyRemoteProcTwo将用于非阻塞的、基于过程的通信。
                                        // 它既没有参数也没有返回值。
```

13.2 过程特征的参数

特征定义可以有参数。在一个 **signature** 定义中，参数列表可以包括参数标识符、参数类型及其方向，即 **in**、**out** 或 **inout**。方向 **inout** 和 **out** 指明这些参数用于从远程过程获取信息。注意，参数的方向指的是从被调用方看的方向，而不是从调用方看的方向。

例如：

```
signature MyRemoteProcThree (in integer Par1, out float Par2, inout integer Par3);
// MyRemoteProcThree将用于阻塞的、基于过程的通信。该过程有三个参数：Par1是整数型输入参数，
// Par2是一个浮点数型输出参数，Par3是一个整数型的输入/输出参数。
```

13.3 返回远程过程的值

一个远程过程可以在其终止后返回一个值。返回值的类型应通过对应 **signature** 定义中的 **return** 子句来指明。

例如：

```
signature MyRemoteProcFour (in integer Par1) return integer;
// MyRemoteProcFour将用于阻塞的、基于过程的通信，该过程有一个整数型的输入参数Par1，
// 在其终止后返回一个整数型的值。
```

13.4 异常描述

在 TTCN-3 中，用一种特定类型的值来表示可能在远程过程中出现的异常。因此，可以使用模板机制和匹配机制来描述或检查远程过程的返回值。

注 — 由于从SUT产生的异常或发送到SUT的异常向对应TTCN-3类型或SUT表示方法的转换是工具特定的或系统特定的，因此它超出了本建议书的讨论范围。

异常以包括在 **signature** 定义中的异常列表形式进行定义。该列表定义了与可能的异常集合相关的所有可能的类型（在异常的含义中，它们通常只是通过这些类型的特定值来区分）。

例如：

```
signature MyRemoteProcFive (inout float Par1) return integer
      exception (ExceptionType1, ExceptionType2);
// MyRemoteProcFive将用于阻塞的、基于过程的通信。它可以在输入输出参数Par1中返回一个浮点数值和一个整数值，
// 或是产生ExceptionType1类型或ExceptionType2类型的异常。

signature MyRemoteProcSix (in integer Par1) noblock
      exception (integer, float);
// MyRemoteProcSix将用于非阻塞的、基于过程的通信。在过程非成功终止情况下，MyRemoteProcSix可能
// 会产生整数型或浮点数型异常。
```

14 模板声明

14.0 概述

模板用于传送特定值的集合，或者用于测试接收值的集合是否与模板说明匹配。模板可以全局地在模板定义部分进行定义，可以局部地在测试用例、函数、可选步骤或声明块中进行定义，或者作为通信操作的子变量嵌入，或者作为测试用例、函数或可选步骤调用的实参嵌入。

模板提供了以下可能性：

a） 模板提供了一种组织和重用测试数据的方法，包括继承的简单形式；

b） 模板可以被参数化；

c） 模板允许匹配机制；

d） 模板可以用于基于消息的通信，也可以用于基于过程的通信。

可以在一个模板值中说明值域和匹配属性，然后在基于消息的通信和基于过程的通信中中使用。模板可以用来说明任何 TTCN-3 类型或过程特征。基于类型的模板用于基于消息的通信，特征模板在基于过程的通信中使用。

一个模板声明必须详细说明一个基本值的集合，或是一个与相应类型或特征定义中每一个字段相匹配的符号的集合；即必须完整地对它进行描述。一个修改后的模板声明（见第 14.6 节）只说明基础模板中需修改的字段；也就是说，它是一个部分说明。“未用的符号”只能在特征模板的不相关参数以及经修改的模板声明和经修改的内嵌模板中使用，用于指明特定字段或元素未做任何变动。

当说明模板或模板字段时，对表达式中所用的函数，存在许多限制；对这些，在第 16.1.4 节中进行说明。

14.1 消息模板声明

14.1.0 概述

可以用模板来说明带有实际值的消息实例，一个模板可以被认为是创建一个发送消息或匹配一个接收消息的指令集合。

可以为表 3 中定义的除 **port** 和 **default** 类型外的任何 TTCN-3 类型说明模板。

例如：

```
// 用于接收操作时，该模板将匹配任意整数值。

template integer Mytemplate := ?;
//该模板将仅匹配整数值1、2或3。
template integer Mytemplate := (1, 2, 3);
```

14.1.1 用于发送消息的模板

在一个 **send** 操作中使用的模板定义一个完整的字段值集合，其中包含在测试端口上传输的消息。在执行 **send** 操作时，应对模板进行完全定义，也就是说，所有字段都将解析为实际值，且在模板字段中不得直接或间接地使用任何匹配机制。

注 — 对发送模板，省略一个可选择字段被看作是一个值记法，而不是一个匹配机制。

例如：

```
// 假定消息定义
type record MyMessageType
{
  integer field1 optional,
  charstring  field2,
  boolean field3
}

// 一个消息模板可以是：
template MyMessageType MyTemplate:=
{
  field1 := omit,
  field2 := "My string",
  field3 := true
}

// 一个相应的发送操作可以是：
MyPCO.send(MyTemplate);
```

14.1.2 用于接收消息的模板

在一个 **receive**、**trigger** 或 **check** 操作中使用的模板定义一个与进入消息相匹配的数据模板。在附件 B 中定义的匹配机制可以用在接收模板中。任何进入值都不得绑定于模板。

例如：

```
// 假定消息定义
type record MyMessageType
{
  integer field1 optional,
  charstring  field2,
  boolean  field3
}

// 一个消息模板可以是：
template MyMessageType MyTemplate:=
```

```
{
  field1 := ?,
  field2 := pattern "abc*xyz",
  field3 := true
}

// 一个相应的接收操作可以是：
MyPCO.receive(MyTemplate);
```

14.2 特征模板的声明

14.2.0 概述

可以使用模板来说明带有实际值的过程参数列表实例，可以通过引用相关的特征定义来为任意过程定义模板。特征模板只定义过程参数的值和匹配机制，而不返回值。返回的值或匹配机制必须在 **reply** 或 **getreply** 操作内进行定义（分别见第 23.3.3 节和第 23.3.4 节）。

例如：

```
// 一个远程过程的特征定义
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3) return integer;

// 与定义的过程特征相关的模板例子
template RemoteProc Template1:=
{
  Par1 := 1,
  Par2 := 2,
  Par3 := 3
}

template RemoteProc Template2:=
{
  Par1 := 1,
  Par2 := ?,
  Par3 := 3
}

template RemoteProc Template3:=
{
  Par1 := 1,
  Par2 := ?,
  Par3 := ?
}
```

14.2.1 用于调用过程的模板

用于 **call** 或 **reply** 操作的模板为所有的 **in** 和 **inout** 参数定义一个完整的字段值集合。在 **call** 操作时，模板中的所有 **in** 和 **inout** 参数都应解析为实际的值；在这些字段中不得直接或间接地使用任何匹配机制。任何 **out** 参数的模板说明都将被简单地忽略掉；因此，允许为这些字段规定匹配机制或省略匹配机制（见附件 B）。

例如：

```
// 在第14.2.0节中假定的例子

// 有效的调用，因为所有的in和inout参数都具有一个不同的值。
MyPCO.call(RemoteProc:Template1);

// 有效的调用，因为所有的in和inout参数都具有一个不同的值。
MyPCO.call(RemoteProc:Template2);

// 无效的调用，因为inout参数Par3具有的是匹配属性，而不是一个值。
MyPCO.call(RemoteProc:Template3);

// 模板从不返回值。对于Par2和Par3的情况，必须在调用语句末尾处使用赋值子句来重新获得通过调用操作返回的值。
```

14.2.2 用于接收过程调用的模板

用于 **getcall** 操作的模板定义一个与进入参数字段相匹配的数据模板。附件 B 中定义的匹配机制可以用于由该操作使用的任何模板。不得发生任何进入值与模板的绑定。在匹配过程中，将忽略任何 **out** 参数。

例如：

```
// 在第14.2.0节中假定的例子

// 有效的getcall，如果Par1 == 1且Par3 == 3，那么它匹配。
MyPCO.getcall(RemoteProc:Template1);

// 有效的getcall，如果Par1 == 1且Par3 == 3，那么它匹配。
MyPCO.getcall(RemoteProc:Template2);

//有效的getcall，如果Par1 == 1，Par3可以为任意值，那么它匹配。
MyPCO.getcall(RemoteProc:Template3);
```

14.3 模板匹配机制

14.3.0 概述

一般来讲，匹配机制用来替换单个模板字段的值或甚至是模板的全部内容，其中一些机制可以联合使用。

匹配机制和通配符也可以只内嵌在接收事件使用（即 **receive**、**trigger**、**getcall**、**getreply** 和 **catch** 操作）。它们可以以明确的值的形式出现。

例 1：

```
MyPCO.receive(charstring:"abcxyz");
MyPCO.receive (integer:complement(1, 2, 3));
```

当值可明确确定类型时，可省略类型标识符。

例 2：

```
MyPCO.receive("AAAA"O);
```

注 — 下列类型可以被省略：integer、float、Boolean、bitstring、hexstring、octetstring。

不过，内嵌式模板类型应在用于接收模板的端口列表中。在列出的类型和提供的值类型之间存在不确定性的情况下（如由子类型造成的），类型名称将包括在接收语句中。

匹配机制分为四类：

a） 特定的值：
- — 求特定值的表达式；
- — **omit**：值被省略；

b） 可以用来代替值的特殊符号：
- — **(…)**：值列表；
- — **complement (…)**：值列表的补集；
- — **?**：表示任意值的通配符；
- — ***：表示任意值或根本不表示任何**value**（即一个省略值）的通配符；
- — **(**下限**..**上限**)**：上限与下限之间（包括上限和下限）的**整数**或浮点数值域；
- — **superset**：至少是列出的所有元素，即可能更多；
- — **subset**：至多是列出的元素，即可能更少；

c） 可以在值内部使用的特殊符号：
- — **?**：表示串、数组、**record of**或**set of**中任意单个元素的通配符；
- — ***：表示串、数组、**record of**或**set of**中任意数目连续元素或根本没有任何元素（即一个省略元素）的通配符；
- — **permutation**：以任意次序列出的所有元素（注：？和*也允许作为数列列表的元素）；

d） 描述值属性的特殊符号：

— **length**：用于串类型的串长度限制以及**record of**、**set of**和数组的元素数目限制；

— **ifpresent**：用于匹配（如果没有省略的话）可选择字段值。

支持的匹配机制及其相关的符号（如果有的话），及其应用范围，见表 6。表中左起第一列列出了适用于这些匹配机制的所有 TTCN-3 类型。每个匹配机制的完整描述见附件 B。

表 6/Z.140－TTCN-3匹配机制

与…值一起使用	值		代 替 值							内 部 值			属 性	
	特定值	省略值	补集	值列表	任意值 (?)	任意值或无 (*)	值域	超集	子集	任意元素 (?)	任意元素或无 (*)	依据变化	长度限制	如果出现
boolean	是	是	是	是	是	是 [a]								是 [b]
integer	是	是	是	是	是	是 [a]	是							是 [b]
float	是	是	是	是	是	是 [a]	是							是 [b]
bitstring	是	是	是	是	是	是 [a]				是	是		是	是 [b]
octetstring	是	是	是	是	是	是 [a]				是	是		是	是 [b]
hexstring	是	是	是	是	是	是 [a]				是	是		是	是 [b]
character strings	是	是	是	是	是	是 [a]	是			是	是		是	是 [b]
record	是	是	是	是	是	是 [a]								是 [b]
record of	是	是	是	是	是	是 [a]				是	是	是	是	是 [b]
array	是	是	是	是	是	是 [a]				是	是		是	是 [b]
set	是	是	是	是	是	是 [a]								是 [b]
set of	是	是	是	是	是	是 [a]		是	是	是	是		是	是 [b]
enumerated	是	是	是	是	是	是 [a]								是 [b]
union	是	是	是	是	是	是 [a]								是 [b]
anytype	是	是	是	是	是	是 [a]								是 [b]

[a] — 使用时，只能用于记录和集合类型的可选字段（对该字段的类型没有限制）。

[b] — 使用时，只能用于记录和集合字段（对该字段的类型没有限制）。

14.3.1 引用模板元素或模板字段

14.3.1.1 引用单个的串元素

不允许在模板或模板字段内引用单个的串元素。

例如：

```
var template charstring t_Char1 := 'MYCHAR';
var template charstring t_Char2;

t_Char2 := t_Char1[1];
  // 由于不允许引用单个的串元素，因此将引起一个错误；
```

14.3.1.2 引用record 和 set字段

模板和模板变量都允许使用点记法来引用模板定义内的子字段。不过，所引用的字段可以是一个结构字段的子字段，对它赋予匹配机制。本节规定了有关如何处理这些情况的规则。

- omit、AnyValueOrNone、值列表和列表补集：引用结构字段内的子字段（对它赋予**omit**、AnyValueOrNone (*)、值列表或列表补集），将引起一个错误。

例 1：

```
type record R1 {
  integer f1 optional,
  R2      f2 optional
}
type record R2 {
  integer g1,
  R2      g2 optional
}

:
var template R1 t_R1 := {
  f1 := 5,
  f2 := omit
}
var template R2 t_R2 := t_R1.f2.g2;
  // 由于将omit赋予了t_R1.f2，因此将引起一个错误。
t_R1. f2 := *
t_R2 := t_R1.f2.g2;
  // 由于将*赋予了t_R1.f2，因此将引起一个错误。

t_R1 := ({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}});

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // 由于将值列表赋予了t_R1，因此所有这些赋值都将引起错误。

t_R1 :=
  complement({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}})

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // 由于将作为补集的列表赋予了t_R1，因此所有这些赋值都将引起错误。
```

- AnyValue：当在赋值符号的右侧引用结构字段（对它赋予AnyValue (?)）内的子字段时，对强制性子字段将返回AnyValue (?)，对可选子字段将AnyValueOrNone。

  当在赋值符号的左侧引用结构字段（对它赋予AnyValue (?)）内的子字段时，结构字段必须递归地扩充至所引用子字段的深度。在该扩充期间，应将AnyValue (?)赋给强制性子字段，并将AnyValueOrNone赋给可选子字段。在该扩充后，赋值符号右侧的值或匹配机制将赋给所引用的子字段。

例 2：

```
t_R1 := {f1:=0, f2:=?}
t_R2 := t_R1.f2.g2;
  // t_R2赋值之后将是{g1:=?, g2:=*}
t_R1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
  // 首先假设字段t_R1.f2扩充为{g1:=?,g2:={g1:=?,g2:=*}},
  // 因此t_R1赋值之后将是：
  //   {f1:=0, f2:={g1:=?,g2:={g1:=?,g2:=({g1:=1, g2:=omit},{g1:=2, g2:=omit})}}}
```

- Ifpresent属性：引用结构字段内的一个子字段（**ifpresent**属性附于其上），将引起一个错误（而不管添加**ifpresent**的值或匹配机制是什么）。

14.3.1.3 引用record of 和 set of元素

模板和模板变量都允许使用下标记法来引用 **record of** 或 **set of** 模板或字段的元素。不过，可以将匹配机制赋给在其内引用元素的模板或字段。本节规定了有关如何处理这些情况的规则。

- omit、AnyValueOrNone、值列表、列表补集、子集和超集：引用record of或set of字段内的元素（对它赋予**omit**、带或不带长度属性的AnyValueOrNone (*)、值列表、列表补集、子集或超集），将引起一个错误。

例 1：

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoRoI := ({},{0},{0,0},{0,0,0});
t_RoI := t_RoRoI[0];
  // 由于将值列表赋予了t_RoRoI，因此将引起一个错误；
```

- AnyValue：当在赋值符号的右侧引用**record of** 或 **set of**模板或字段的元素时（对它赋予AnyValue (?)，不带长度属性），将返回AnyValue (?)。如果长度属性附于AnyValue (?)上，那么引用的下标不得与长度属性相冲突。
  当在赋值符号的左侧引用**record of** 或 **set of**模板或字段内的元素时（对它赋予AnyValue (?)，不带长度属性），将对赋值符号右侧的值或匹配机制赋予所引用的元素，对所引用元素（如果有的话）之前的所有元素赋予AnyElement(?)，在末尾添加一个单个的AnyElementsOrNone(*)。当长度属性附于AnyValue (?)上时，属性将被透明地传送给新的模板或字段。在上述的任何情况下，下标都不得与类型限制相冲突。

例 2：

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoI := ?;
t_Int := t_RoI[5];
  // t_Int赋值之后将是AnyValue(?);

t_RoRoI := ?;
t_RoI := t_RoRoI[5];
  // t_RoI赋值之后将是AnyValue(?);
t_Int := t_RoRoI[5].[3];
  // t_Int赋值之后将是AnyValue(?);

t_RoI := ? length (2..5);
t_Int := t_RoI[3];
  // t_Int赋值之后将是AnyValue(?);
t_Int := t_RoI[5];
  // 由于所引用的下标处于长度属性之外，因此将引起一个错误（注：下标5指的是第6个元素）；

t_RoRoI[2] := {0,0};
  // t_RoRoI赋值之后将是{?,?,{0,0},*};
t_RoRoI[4] := {1,1};
  // t_RoRoI赋值之后将是{?,?,{0,0},?,{1,1},*};
t_RoI[0] := -5;
  // t_RoI赋值之后将是{-5,*}length(2..5);
t_RoI := ? length (2..5);
t_RoI[1] := 1;
  // t_RoI赋值之后将是{?,1,*}length(2..5);
t_RoI[3] := ?
  // t_RoI赋值之后将是{?,1,?,?,*}length(2..5);
t_RoI[5] := 5
  // t_RoI赋值之后将是{?,1,?,?,?,5,*}length(2..5)；注：t_RoI变成一个空集，但不会引起任何错误；
```

- 数列：当引用一个位于数列内的**record of**模板或字段的元素时（基于下标），将引起一个错误。受数列保护的元素下标应基于数列元素的数量来确定。AnyValueOrNone作为数列元素将使数列保护所有的**record of**元素下标。

例 3：

```
t_RoI := {permutation(0,1,3,?),2,?}
t_Int := t_RoI[5];
  // t_Int赋值之后将是AnyValue(?)

t_RoI := {permutation(0,1,3,?),2,*}
t_Int := t_RoI[5];
  // t_Int赋值之后将是* (AnyValueOrNone)
t_Int := t_RoI[2];
  // 由于第3个元素（下标为2）处于数列之内，因此会引起错误。

t_RoI := {permutation(0,1,3,*),2,?}
t_Int := t_RoI[5];
  // 由于数列包含AnyValueOrNone(*)，它能覆盖任何下标记录，因此会引起错误。
```

- Ifpresent属性：引用**record of** 或 **set of**字段中的一个元素（**ifpresent**属性附于其上），将引起一个错误（而不管添加**ifpresent**的值或匹配机制是什么）。

14.4 模板参数化

14.4.0 概述

用于发送和接收操作的模板都可以被参数化。一个模板的实参可以包括值和模板、函数和特殊的匹配符号。形参和实参列表的规则应遵循第 5.2 节中所述的要求。

例如：

```
// 模板
template MyMessageType MyTemplate (integer MyFormalParam):=
{
  field1 := MyFormalParam,
  field2 := pattern "abc*xyz",
  field3 := true
}

// 可以按以下方式来使用：
pco1.send(MyTemplate(123));
```

14.5 空缺

14.6 修改后的模板

14.6.0 概述

通常，模板用于说明基本值或缺省值的集合，或者在适当类型或特征定义中定义的每个字段的匹配符号的集合。在只需要少量变化来说明一个新模板的情况下，可以说明一个修改后的模板。一个修改后的模板用于说明对原始模板中特定字段的直接或间接更改。

关键字 **modifies** 表示获得新模板或修改后模板的父模板，该父模板可以是一个原始模板，也可以是一个经修改的模板。

更改以链接的方式发生，最终要追溯到原始模板。如果一个模板字段及其对应的值或匹配符号在修改后的模板中进行说明，那么将使用所描述的值或匹配符号来替换父模板中所描述的值或匹配符号。如果模板字段及其对应的值或匹配符号未在修改后的模板中进行说明，那么将使用父模板中的值或匹配符号。当要修改的字段嵌套在一个自身为结构字段的模板字段中时，除了予以显性说明外，不修改结构字段的任何其它字段。

无论是直接地还是间接地，修改后的模板都不能自己调用自己，也就是说，不允许递归调用。

例 1：

```
// 假定
type record MyRecordType
{
  integer field,
```

```
  charstring field2,
  boolean field3
}
template MyRecordType MyTemplate1 :=
{
  field1 := 123,
  field2 := "A string",
  field3 := true
}
// 则写为
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
  field1 := omit,          // field1是可选的，但出现在MyTemplate1中。
  field2 := "A modified string"
                           // field3未发生变化。
}
// 同写。
template MyRecordType MyTemplate2 :=
{
  field1 := omit,
  field2 := "A modified string",
  field3 := true
}
```

当修改模板的各个值或 **record of** 类型的修改模板字段希望改变时，且只能在这些情况下，还可以使用赋值记法，当中，赋值符号的左侧为要修改元素的下标。

例 2：

```
template MyRecordOfType MyBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
template MyRecordOfType MyModifTemplate modifies MyBaseTemplate := { [2] := 3, [3] := 2 };
// MyModifTemplate应匹配值序列{ 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }。
```

14.6.1 修改后模板的参数化

如果一个基础模板有一个形参列表，那么以下规则将应用于所有源自该基础模板的修改模板，可以经过也可以不经过一个或多个修改步骤产生：

a） 派生模板不能省略在基础模板与实际修改模板之间的任何修改步骤上定义的参数；

b） 如果需要，派生模板可以拥有额外的（附加的）参数；

c） 对每一个修改模板，形参列表都要跟在模板名称之后。

例如：

```
// 假定
template MyRecordType MyTemplate1(integer MyPar):=
{
  field1 := MyPar,
  field2 := "A string",
  field3 := true
}

// 则可以有如下改变：
  template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{ // field1在Template1中参数化，也仍在Template2参数化。
  field2 := "A modified string",
}
```

14.6.2 内嵌式修改模板

与显性创建命名的修改模板一样，TTCN-3 允许定义内嵌式修改模板。

例如：

```
// 假定
template MyMessageType Setup :=
{ field1 := 75,
  field2 := "abc",
  field3 := true
}

// 可用于定义Setup的一个内嵌式修改模板。
pco1.send (modifies Setup := {field1:= 76});
```

14.7 改变模板字段

在通信操作中（如 **send**、**receive**、**call**、**getcall** 等），只允许通过参数化或内嵌式派生模板来改变模板字段。在相应通信事件后的模板中，这些改变对模板字段值不会造成影响。

在通信事件中，不得使用点记法 *MyTemplateId.FieldId* 来设置或重新获得模板的值，对此，应使用“->”符号（见第 23 节）。

14.8 匹配操作

match 操作允许变量或参数的值与模板进行比较，该操作将返回一个布尔值。如果模板类型与变量类型不兼容（见第 6.7 节），那么该操作将返回假（false）。如果模板类型与变量类型兼容，那么该操作的返回值将指明变量值是否与给定模板相一致。

例如：

```
template integer LessThan10 := (-infinity..9);

testcase TC001()
runs on MyMTCType
{
  var integer RxValue;
   :
  PCO1.receive(integer:?) -> value RxValue;

  if( match( RxValue, LessThan10)) { … }
// 如果Rxvalue的实际值小于10，则返回真（true），否则返回假（false）。
   :
}
```

14.9 valueof操作

valueof 操作允许将模板中指定的值赋给一个变量。变量和模板的类型应是兼容的（见第 6.7 节），模板的各个字段都应解析为一个单个的值。

例如：

```
type record ExampleType
{
  integer field1,
  boolean field2
}

template ExampleType SetupTemplate :=
{
  field1 := 1,
  field2 := true
}

 :
var ExampleType RxValue := valueof(SetupTemplate);
```

15 运算符

15.0 概述

TTCN-3 支持许多可用于 TTCN-3 表达式的术语中的预定义运算符。这些预定义运算符分为以下七类：

a) 算术运算符；
b） 串运算符；
c） 关系运算符；
d） 逻辑运算符；
e） 位运算符；
f） 移位运算符；
g） 循环移位运算符。

这些运算符列于表 7 中。

表 7/Z.140－TTCN-3运算符列表

类　　别	运　算　符	符号或关键字
算术运算符	加	**+**
	减	**-**
	乘	*****
	除	**/**
	取模	**mod**
	取余	**rem**
串运算符	串联	**&**
关系运算符	等于	**==**
	小于	**<**
	大于	**>**
	不等于	**!=**
	大于或等于	**>=**
	小于或等于	**<=**
逻辑运算符	逻辑非	**not**
	逻辑与	**and**
	逻辑或	**or**
	逻辑异或	**xor**
位运算符	位非	**not4b**
	位与	**and4b**
	位或	**or4b**
	位异或	**xor4b**
移位运算符	左移	**<<**
	右移	**>>**
循环移位运算符	左循环移位	**<@**
	右循环移位	**@>**

表 8 显示了这些运算符的优先级。表中每一行中所列的运算符都具有相同的优先级。如果一个表达式中出现多个具有相同优先级的运算符，那么从左到右进行运算。括号可以用于将操作数归拢在一起，在这种情况下，括起来的表达式具有最高计算优先级。

表 8/Z.140－运算优先级

优先级	运算符类型	运　算　符
最高		**(　…　)**
	一元	**+, -**
	二元	***, /, mod, rem**
	二元	**+, -, &**
	一元	**not4b**
	二元	**and4b**
	二元	**xor4b**
	二元	**or4b**
	二元	**<<, >>, <@, @>**
	二元	**<, >, <=, >=**
	二元	**==, !=**
	一元	**not**
	二元	**and**
	二元	**xor**
最低	二元	**or**

15.1 算术运算符

算术运算符指的是加、减、乘、除、取模和取余操作。除了 **mod** 和 **rem** 运算符只能与 **integer**（包括 **integer** 的衍生类型）类型一起使用，这些运算符的操作数应为 **integer** 类型（包括 **integer** 的派生类型）或 **float** 类型（包括 **float** 的派生类型）。

对 **integer** 类型的操作数，其算术运算的结果类型为 **integer**。对 float 类型的操作数，其算术运算的结果类型为 **float**。

在加（＋）或减（－）用作一元运算符的情况下，操作数的应用规则同前。如果对正的操作数使用减运算符，那么其结果为操作数的负值，反之亦然，即对负的操作数使用加运算符，那么其结果为操作数的正值。

执行除（/）操作的结果有两种：

a） 当被除数和除数均为**integer**时，结果为计算值的**integer**部分（也就是说，小数部分将被抛弃）；

b） 当被除数和除数均为**float**时，结果为计算值的**float**值（也就是说，不抛弃小数部分）。

对 **integer** 类型的操作数进行 **rem** 和 **mod** 操作，其结果为 **integer** 类型。x **rem** y 和 x **mod** y 操作都计算 x 被 y 整除后的余数。因此，只能对非零的操作数 y 定义这两个运算符。对正数 x 和 y，x **rem** y 和 x **mod** y 操作具有相同的结果，但对负数，计算结果就不一样了。

mod 和 **rem** 操作的形式化定义如下：

```
x rem y = x - y * (x/y)
x mod y    = x rem |y|        when  x >= 0
      = 0                     when  x < 0 and   x rem |y| = 0
      = |y| + x rem |y|       when  x < 0 and   x rem |y| < 0
```

表 9 描述了 **mod** 和 **rem** 运算符的区别：

表9/Z.140－mod和rem运算符的结果

x	–3	–2	–1	0	1	2	3
x mod 3	0	1	2	0	1	2	0
x rem 3	0	–2	–1	0	1	2	0

15.2 串运算符

预定义的串运算符执行兼容串类型值的连接操作，操作为从左至右的简单连接，没有暗含任何算术加法的形式，结果类型为操作数的原始类型。

例如：

```
'1111'B & '0000'B & '1111'B gives '111100001111'B
```

15.3 关系运算符

预定义的关系运算符指的是等于（==）、小于（<）、大于（>）、不等于（!=）、大于等于（>=）和小于等于（<=）关系。除了 **enumerated** 类型，等于和不等于运算符的操作数可以是任意的但兼容的类型，在这种情况下，操作数应是相同类型的实例。所有其它关系运算符的操作数类型都只能是 **integer**（包括 **integer** 的派生类型）、**float**（包括 **float** 的派生类型）或相同 **enumerated** 类型的实例。这些运算符的结果类型为 **boolean**。

当且仅当两个 **charstring** 或 **universal charstring** 值的长度以及所有对应位置的字符都相同时，两个值才相等。对 **bitstring**、**hexstring** 或 **octetstring** 类型的值，使用相同的相等规则，但有例外 —— 所有对应位置上都应相同的元素分别为位、十六进制数字或十六进制数字对。

当且仅当其有效值结构兼容（见第 6.7 节）且所有对应字段的值都相等时，两个 **record** 值、**set** 值、**record of** 值或 **set of** 值才是相等。**record** 值还可以与 **record of** 值进行比较，**set** 值还可以与 **set of** 值进行比较。在这些情况下的比较规则同两个 **record** 值或 **set** 值的比较规则。

注 — “所有字段”意味着在**record**类型实际值中未出现的可选字段将被认为是未定义的值。这样一个字段当与另一个**record**类型的值进行比较时，仅等同于一个遗漏的可选字段（同样被认为是一个未定义的值），或者当它与一个**record of**类型的值进行比较时，仅等同于一个带未定义值的元素。该规则也适用于在两个**set**类型值或者一个**set**类型值与一个**set of**类型值进行比较时。

当且仅当在两个 **union** 类型的值中，所选字段的类型兼容且所选字段的实际值相等时，才认为这两个 **union** 类型值是相等的。

例如：

```
// 假定
type  set  SetA{
       integer     a1    optional,
       integer     a2    optional,
       integer     a3    optional
       };

type  set  SetB{
       integer     b1    optional,
       integer     b2    optional,
       integer     b3    optional
       };

type  set  SetC{
       integer     c1    optional,
       integer     c2    optional,
       };

type  set of integer    SetOf;

type  union UniD  {
       integer     d1,
       integer     d2,
       };

type  union UniE  {
       integer     e1,
       integer     e2,
       };

type  union UniF  {
       integer     f1,
       integer     f2,
       boolean     f3,
       };

// 且
const      Set A  conSetA1  := { a1 := 0, a2 := omit, a3 := 2 };
            // 注意：字段定义值的次序无关紧要。
const      SetB   conSetB1  := { b1 := 0, b3 := 2, b2 := omit };
const      SetB   conSetB2  := { b2 := 0, b3 := 2, b1 := omit };
const      SetC   conSetC1  := { c1 := 0, c2 :=2 };
const      SetOf  conSetOf1 := { 0, omit, 2 };
const      SetOf  conSetOf2 := { 0, 2 };
const      UniD   conUniD1  := { d1:= 0 };
const      UniE   conUniE1  := { e1:= 0 };
const      UniE   conUniE2; := { e2:= 0 };
const      UniF   conUniF1; := { f1:= 0 };

// 则
conSetA1 == conSetB1;
  // 返回true
conSetA1 == conSetB2;
  //  返回false，因为a1和a2与其对应值不等（对应的元素没有被省略）。
conSetA1 == conSetC1;
  //  返回false，因为SetA和SetC的有效值结构不兼容。
conSetA1 == conSetOf1;
  // 返回true
conSetA1 == conSetOf2;
  //  返回false，因为省略之a2的对应值为2，但a3的对应值未定义。
conSetC1 == conSetOf2;
  // 返回true
conUniD1 == conUniE1;
  // 返回true
```

```
conUniD1 == conUniE2;
  // 返回false，因为选中的字段e2不是UniD1的d1字段的对应字段。
conUniD1 == conUniF1;
  // 返回false，因为UniD1和UniF的有效值结构不兼容。
```

15.4 逻辑运算符

预定义的 **boolean** 运算符执行逻辑 **not**、逻辑 **and**、逻辑 **or** 和逻辑 **xor** 操作，其操作数应为 **boolean** 类型，逻辑运算的结果为 **boolean** 类型。

逻辑 **not** 是一个一元运算符，如果其操作数的值是 **false**，则返回 **true**，如果操作数的值为 **true**，则返回 **false**。

如果逻辑 **and** 运算的两个操作数的值均为 **true**，则返回 **true**；否则返回值 **false**。

如果逻辑 **or** 运算的两个操作数中至少有一个为 **true**，则返回值 **true**；当且仅当两个操作数都为 **false** 时，返回值 **false**。

如果逻辑 **xor** 运算的两个操作数中有且只有一个操作数为 **true**，则返回值 **true**；如果其两个操作数的值都为 **false** 或多为 **true**，则返回 **false**。

对布尔表达式使用短路计算方法；即一旦知道整个结果则停止对逻辑运算符操作数的计算：在运算符 **and** 情况下，如果左侧变元的计算结果为 **false**，那么对右侧变元不再进行计算，整个表达式的计算结果为 **false**。在运算符 **or** 情况下，如果左侧变元的结算结果为 **true**，那么对右侧变元不再进行计算，整个表达式的结算结果为 **true**。

15.5 位运算符

预定义的位运算符执行按位 **not**、按位 **and**、按位 **or** 和按位 **xor** 操作，这些运算符分别为 **not4b**、**and4b**、**or4b** 和 **xor4b**。

注 — 读作“按位取反”、“按位与”等。

位运算符的操作数应为 **bitstring**、**hexstring** 或 **octetstring**。对 **and4b**、**or4b** 和 **xor4b** 的情况，两个操作数的类型应是兼容的。位运算符的结果类型应是操作数的原始类型。

位运算 **not4b** 是一个一元运算符，它对操作数的每一位执行取反操作，将操作数中为 1 的位变为 0，为 0 的位变为 1，即：

```
not4b  '1'B  得  '0'B
not4b  '0'B  得  '1'B
```

例 1：

```
not4b  '1010'B  得  '0101'B
not4b  '1A5'H   得  'E5A'H
not4b  '01A5'O  得  'FE5A'O
```

位运算符 **and4b** 接受两个等长的操作数。对每个对应位位置，如果两个操作数中各位都设为 1，那么结果值为 1；否则结果位的值为 0。即：

```
'1'B and4b '1'B  得  '1'B
'1'B and4b '0'B  得  '0'B
'0'B and4b '1'B  得  '0'B
'0'B and4b '0'B  得  '0'B
```

例 2：

```
'1001'B and4b '0101'B  得  '0001'B
'B'H and4b '5'H        得  '1'H
'FB'O and4b '15'O      得  '11'O
```

位运算符 **or4b** 接受两个等长的操作数。对每个对应位位置，如果两个操作数中各位都设为 0，那么结果值为 0；否则结果位的值为 1。即：

```
'1'B or4b '1'B  得  '1'B
'1'B or4b '0'B  得  '1'B
'0'B or4b '1'B  得  '1'B
'0'B or4b '0'B  得  '0'B
```

例 3：

```
'1001'B or4b '0101'B  得  '1101'B
'9'H or4b '5'H        得  'D'H
'A9'O or4b 'F5'O      得  'FD'O
```

位运算符 **xor4b** 接受两个等长的操作数。对每个对应位位置，如果两个操作数中各位都设为 0 或 1，那么结果为 0；否则结果位的值为 1。即：

```
'1'B xor4b '1'B  得  '0'B
'0'B xor4b '0'B  得  '0'B
'0'B xor4b '1'B  得  '1'B
'1'B xor4b '0'B  得  '1'B
```

例 4：

```
'1001'B xor4b '0101'B  得  '1100'B
'9'H xor4b '5'H        得  'C'H
'39'O xor4b '15'O      得  '2C'O
```

15.6 移位运算符

预定义的移位运算符执行左移（**<<**）和右移（**>>**）操作。其左侧的操作数应是 **bitstring**、**hexstring** 或 **octetstring** 类型。其右侧的操作数应是 **integer** 类型。这些运算符的结果类型应与其左侧操作数的类型相同。

移位运算符根据其左侧操作数类型的不同而不同，如果其左侧操作数的类型为：

a） **bitstring**，那么应用的移位单位是1个位；

b） **hexstring**，那么应用的移位单位是1个十六进制数字；

c） **octetstring**，那么应用的移位单位是1个八位字节。

左移（**<<**）运算符接受两个操作数，它将左侧操作数左移右侧操作数所述的位数，将抛弃移出的过多的单位（位、十六进制数字或八位字节）。对左移的每个移动单位，自移动后的操作数右侧开始插入 0（根据左侧操作数的类型决定是'0'B、'0'H 或是 '00'O）。

例 1：

```
'111001'B  << 2  得  '100100'B
'12345'H  << 2   得  '34500'H
'1122334455'O  << (1+1)  得  '3344550000'O
```

右移（**>>**）运算符接受两个操作数，它将左侧操作数右移右侧操作数所述的位数，将抛弃移出的过多的单位（位、十六进制数字或八位字节）。对右移的每个移动单位，自移动后的操作数左侧开始插入 0（根据左侧操作数的类型决定是'0'B、'0'H 或是 '00'O）。

例 2：

```
'111001'B  >> 2  得  '001110'B
'12345'H  >> 2   得  '00123'H
'1122334455'O  >> (1+1)  得  '0000112233'O
```

15.7 循环移位运算符

预定义的循环移位运算符执行循环左移（**<@**）和循环右移（**@>**）操作。其左侧的操作数应是 **bitstring**、**hexstring**、**octetstring**、**charstring** 或 **universal charstring** 类型。其右侧的操作数应是 **integer** 类型。这些运算符的结果类型应与其左侧操作数的类型相同。

循环移位运算根据其左侧操作数类型的不同而不同，如果其左侧操作数的类型为：

a） **bitstring**，那么应用的循环移位单位是1个位；

b） **hexstring**，那么应用的循环移位单位是1个十六进制数字；

c） **octetstring**，那么应用的循环移位单位是1个八位字节；

d） **charstring** 或 **universal charstring**，那么应用的循环移位单位是1个字符。

循环左移（**<@**）运算符接受两个操作数，它将左侧操作数循环左移右侧操作数所述的位数，移出的过多的单位（位、十六进制数字或八位字节）从其右侧重新插入左侧操作数中。

例 1：

```
'101001'B  <@ 2  得  '100110'B
'12345'H   <@ 2  得  '34512'H
'1122334455'O  <@ (1+2)  得  '4455112233'O
"abcdefg" <@ 3 得  "defgabc"
```

循环右移（**@>**）运算符接受两个操作数，它将左侧操作数循环右移右侧操作数所述的位数，移出的过多的单位（位、十六进制数字或八位字节）从其左侧重新插入左侧操作数中。

例 2：

```
'100001'B  @> 2  得  '011000'B
'12345'H   @> 2  得  '45123'H
'1122334455'O  @> (1+2)  得  '3344551122'O
"abcdefg" @> 3  得  "efgabcd"
```

16 函数和可选步骤

在 TTCN-3 中，函数和可选步骤用于规范和构造测试行为、定义一个模块中的缺省行为并组织计算，下面的章节对此做了描述。

16.1 函数

16.1.0 概述

在 TTCN-3 中，用函数来表达测试行为，来组织测试执行，或者来在一个模块中构建计算，例如计算一个单个值，以初始化一个变量集合或检查某些条件。函数可以返回一个值或一个模板。通过后跟一个类型标识符的关键字 **return** 来表示函数要返回一个值。通过后跟一个类型标识符的关键字 **return template** 来表示函数要返回一个模板。

当关键字 **return** 用在函数体中且返回一个在其头部定义的值时，关键字 **return** 后总要接一个表示返回值的表达式。返回值的类型应与返回类型兼容。当关键字 **return** 用在函数体中且返回一个在其头部定义的模板时，关键字 **return** 后总要接一个表示返回模板的表达式或模板实例。返回模板的类型应与返回模板类型兼容。

函数体中的返回语句引起函数终止，并向函数调用处返回返回值。

例 1：

```
// 不带任何参数的MyFunction定义
function MyFunction() return integer
{

  return 7;    //函数终止时返回整数值7
}

// 可以返回符号或模板的函数定义
function MyFunction2() return template integer
{
:
  return ?;    // 返回匹配机制AnyValue
}
function MyFunction3() return template octetstring
{
:
  return "FF??FF"O;    // 在其内部返回一个带AnyElement的八位字节串。
}
```

函数可以在一个模块内定义，或者声明为在外部进行定义（即 **external**）。对一个外部函数，只需在 TTCN-3 模块中提供函数接口。外部函数的实现超出了本建议书的讨论范围。外部函数不允许包含端口操作。外部函数不允许返回模板。

```
external function MyFunction4() return integer;
                                                    // 返回一个整数值的、不带参数的外部函数。

external function InitTestDevices();
                                          // 在TTCN-3模块外仅有一个作用的外部函数
```

在一个模块中，可以使用程序语句和第 18 节中所述的操作来定义一个函数的行为。如果函数使用了在一个部件类型定义中声明的变量、常量、定时器和端口，那么应在该函数头部中使用关键字 **runs on** 来引用该部件类型。该规则的一个例外是当该部件作用范围内的所有必要信息是作为参数传入函数时。

例 2：

```
function MyFunction3() runs on MyPTCType {
  var integer MyVar := 5;     // MyFunction3不返回一个值，但使用端口操作
  PCO1.send(MyVar);
}                             //通过引用一个部件类型来发送，且因此要求一个runs on子句来解析端口标识符。
```

不带 **runs on** 子句的函数从不会调用一个函数或可选步骤，或是激活一个作为缺省的带局部 **runs on** 子句的可选步骤。

使用 **start** 测试部件操作启动的函数总有一个 **runs on** 子句（见第 22.5 节），并被认为是在待启动的部件中被调用，也就是说，它不是局部的。不过，**start** 测试部件操作可以在不带 **runs on** 子句的函数中被调用。

注 — 关于**runs on**子句的限制仅与函数和可选步骤有关，而与测试用例无关。

在 TTCN-3 模块控制部分使用的函数下不会有 **runs on** 子句，不过允许它们执行测试用例。

16.1.1 函数的参数化

函数可以被参数化，应遵循有关形参列表的规则，如第 5.2 节中所定义。

例如：

```
function MyFunction2(inout integer MyPar1) {
                            // MyFunction2不返回值，但改变作为形参传入的MyPar1的值。
  MyPar1 := 10 * MyPar1;
}
```

16.1.2 调用函数

可以通过引用函数名并提供实参列表来调用一个函数。不返回值的函数应直接被调用，返回值的函数既可以被直接调用，也可以在表达式内被调用。应遵循第 5.2 节中定义的有关实参列表的规则。

例如：

```
MyVar := MyFunction4();
                          // MyFunction4返回的值被赋给MyVar，返回值的类型和MyVar的类型必须兼容。

MyFunction2(MyVar2);
                          // MyFunction2不返回值，带实参MyVar2被调用，可以通过引用来传递。

MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3); // 用在表达式中的函数。
```

特殊限制适用于利用 **start** 测试部件操作绑定于测试部件的函数。在第 22.5 节中对这些限制进行了描述。

16.1.3 预定义的函数

TTCN-3 包含了许多无需在使用前声明的预定义（内置）函数。

表 10/Z.140－TTCN-3预定义函数列表

类　别	函　数	关 键 字
转换函数	转换 **integer** 值为 **charstring** 值	**int2char**
	转换 **integer** 值为 **universal charstring** 值	**int2unichar**
	转换 **integer** 值为 **bitstring** 值	**int2bit**
	转换 **integer** 值为 **hexstring** 值	**int2hex**
	转换 **integer** 值为 **octetstring** 值	**int2oct**
	转换 **integer** 值为 **charstring** 值	**int2str**
	转换 **integer** 值为 **float** 值	**int2float**
	转换 **float** 值为 **integer** 值	**float2int**
	转换 **charstring** 值为 **integer** 值	**char2int**
	转换 **charstring** 值为 **octetstring** 值	**char2oct**
	转换 **universal charstring** 值为 **integer** 值	**unichar2int**
	转换 **bitstring** 值为 **integer** 值	**bit2int**
	转换 **bitstring** 值为 **hexstring** 值	**bit2hex**
	转换 **bitstring** 值为 **octetstring** 值	**bit2oct**
	转换 **bitstring** 值为 **charstring** 值	**bit2str**
	转换 **hexstring** 值为 **integer** 值	**hex2int**
	转换 **hexstring** 值为 **bitstring** 值	**hex2bit**
	转换 **hexstring** 值为 **octetstring** 值	**hex2oct**
	转换 **hexstring** 值为 **charstring** 值	**hex2str**
	转换 **octetstring** 值为 **integer** 值	**oct2int**
	转换 **octetstring** 值为 **bitstring** 值	**oct2bit**
	转换 **octetstring** 值为 **hexstring** 值	**oct2hex**
	转换 **octetstring** 值为 **charstring** 值	**oct2str**
	转换 **octetstring** 值为 **charstring** 值	**oct2char**
	转换 **charstring** 值为 **integer** 值	**str2int**
	转换 **charstring** 值为 **octetstring** 值	**str2oct**
	转换 **charstring** 值为 **float** 值	**str2float**
长度/大小函数	返回一个任意串类型值的长度	**lengthof**
	返回 **record**、**record of**、**template**、**set**、**set of** 或 **array** 类型中的元素数目	**sizeof**
	返回结构化类型中的元素数目	**sizeoftype**
出现/选择函数	确定 **record**、**record of**、**template**、**set** 或 **set of** 类型中的可选字段是否出现	**ispresent**
	确定选择了 **union** 类型中的哪一个	**ischosen**
串处理函数	返回输入串中与指定的模式描述相匹配的部分	**regexp**
	返回输入串中的指定部分	**substr**
	用一个串替代一个子串，或者将输入串插入一个串中。	**replace**
其它函数	产生一个随机浮点数	**rnd**

当调用一个预定义函数时：

1） 实参的数目应与形参的数目相同；且

2） 每个实参都应确定其对应形参类型元素的值；且

3） 所有出现在实参列表中的变量都应被绑定。

附件 C 给出了预定义函数的完整描述。

16.1.4 对从特定位置调用的函数的限制

可以在通信操作期间（在模板、模板字段或内置模板中）或者快照计算期间（在 **alt** 语句或可选步骤的布尔防卫中（见第 20.1.1 节）以及在可选步骤局部定义的初始化中（见第 16.2.2 节））调用返回函数的值。为避免会引起部件或实际快照改变的副作用，并防止对未改变快照的后续计算出现不同的结果，对在上述情况中调用的函数不得使用以下操作：

- 所有的部件操作，即**create**、**start**（部件）、**stop**（部件）、**kill**、**running**（部件）、**alive**、**done** 和 **killed**（见注1、注3、注4和注6）。
- 所有的端口操作，即**start**（端口）、**stop**（端口）、**halt**、**clear**、**send**、**receive**、**trigger**、**call**、**getcall**、**reply**、**getreply**、**raise**、**catch**、**check**、**connect**、**map**（见注1、注2、注3和注6）。
- **action**操作（见注2和注6）。
- 所有的定时器操作，即**start**（定时器）、**stop**（定时器）、**running**（定时器）、**read**、**timeout**（见注4和注6）。
- 调用外部函数（见注4和注6）。
- 调用**rnd**预定义函数（见注4和注6）。
- 改变部件变量，即使用赋值语句右侧的部件变量，以及**out**和**inout**参数实例中的部件变量（见注4和注6）。
- 调用**setverdict**操作（见注4和注6）。
- 缺省的激活和去激活，即**activate** 和 **deactivate**语句（见注5和注6）。
- 调用带**out**或**inout**参数的函数（见注7和注8）。

注 1 — 执行**start**、**stop**、**done**、**killed**、**halt**、**clear**、**receive**、**trigger**、**getcall**、**getreply**、**catch**和**check**操作会引起对当前快照的改变。

注 2 — 出于可读性目的，应避免**send**、**call**、**reply**、**raise** 和 **action**操作，即应显性地进行所有的通信，而不是作为另一个通信操作或快照计算的副作用。

注 3 — 出于可读性目的，应避免**map**、**unmap**、**connect**、**disconnect**、**create**操作，即应显性地进行所有的配置操作，而不是作为另一个通信操作或快照计算的副作用。

注 4 — 应避免调用外部函数、**rnd**、**running**、**alive**、**read**、**setverdict** 以及写入部件变量，原因是，这可能导致相同快照的后续计算出现不同的结果，因此应采取一些措施，例如使死锁检测变得不可能。

注 5 — 应避免**activate** 和**deactivate**操作，原因是，它们更改缺省集，这在计算当前快照期间考虑。

注 6 — 除**out** 或 **inout**参数化使用方面的限制外，应递归地应用限制，即不允许直接使用它们，或者通过一个任意长的函数调用链来使用它们。

注 7 — 不能递归应用有关带**out** 或 **inout**参数的调用函数的限制，即调用函数自身调用带**out** 或 **inout**参数的函数是合法的。

注 8 — 应避免使用**out**或**inout**参数，原因是，这也可能导致相同快照的后续计算出现不同的结果。

16.2 可选步骤

16.2.0 概述

TTCN-3 使用可选步骤来描述缺省行为，或构造一个 **alt** 语句的选择对象。可选步骤是与函数相似的作用范围单位。可选步骤主体定义一个局部定义的可选集合和选择对象的集合，所谓的顶层选择对象构成可选步骤的主体。顶层选择对象的语法规则同 **alt** 语句选择对象的语法规则。

可以使用程序语句和第 18 节中综述的操作来定义可选步骤的行为。如果一个可选步骤包括端口操作或使用部件变量、常量或定时器，那么应在可选步骤头部使用关键字 **runs on** 来引用相关的部件类型。该规则的一个例外是当可选步骤中使用的所有端口、变量、常量和定时器都作为传入时。

例如：

```
// 假定
type component MyComponentType {
  var integer MyIntVar := 0;
  timer MyTimer;
  port MyPortTypeOne PCO1, PCO2;
  port MyPortTypeTwo PCO3;
}

// 使用MyComponentType的PCO1、PCO2、MyIntVar和MyTimer的可选步骤定义

altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
  [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) {
     setverdict(inconc);
     }
  [] PCO2.receive {
          repeat
     }
  [] MyTimer.timeout {
     setverdict(fail);
          stop
     }
}
```

可选步骤可以调用函数和可选步骤，或作为缺省来激活可选步骤。一个不带 **runs on** 子句的可选步骤从不会调用函数或可选步骤，或者作为缺省局部激活一个带 **runs on** 子句的可选步骤。

16.2.1 可选步骤的参数化

可选步骤可以被参数化。作为缺省激活的可选步骤应仅带有 **in** 参数、端口参数和定时器参数。在 **alt** 语句中仅作为选择对象被调用的可选步骤，或者在 TTCN-3 行为描述中仅作为独立语句被调用的可选步骤，可以有 **in**、**out** 和 **inout** 参数。应遵循第 5.2 节中定义的、有关形参列表的规则。

16.2.2 可选步骤中的局部定义

16.2.2.0 概述

可选步骤可以定义常量、变量和定时器的局部定义。应该在设置选择对象前定义局部定义。

例如：

```
altstep AnotherAltStep(in integer MyPar1) runs on MyComponentType {
  var integer MyLocalVar := MyFunction();      //   局部变量
  const float MyFloat := 3.41;              //  局部常量
  [] PCO1.receive(MyTemplate(MyPar1, MyLocalVar) {
     setverdict(inconc);
     }
  [] PCO2.receive {
      repeat
     }
}
```

16.2.2.1 可选步骤中局部定义初始化的限制

通过调用值返回函数来初始化局部定义可能会有副作用。为避免会引起部件实际快照与其状态之间不一致的副作用，并防止对未改变快照的后续计算出现不同的结果，应对局部定义的初始化应用第 16.1.4 节中给出的限制。

16.2.3 可选步骤的调用

可选步骤的调用总与 **alt** 语句相关。调用可以通过缺省机制隐性地完成（见第 21 节），或者通过在 **alt** 语句内的一个直接调用显性地完成（见第 20.1.6 节）。可选步骤的调用不会导致任何新的快照，通过使用调用可选步骤的 **alt** 语句的实际快照，来完成对可选步骤顶层选择对象的计算。

注 — 如果在一个选定的顶层选择对象中，指定并进入了一个新的**alt**语句，那么当然会采纳可选步骤中的一个新快照。

对利用缺省机制的可选步骤的隐性调用，在到达调用位置之前，应利用 **activate** 语句将可选步骤激活为一个缺省。

例 1：

```
  :
 var default MyDefVarTwo := activate(MySecondAltStep()); //  一个可选步骤激活为缺省
  :
```

alt 语句中可选步骤的显性调用看起来像是一个作为选择对象的函数调用。

例 2：

```
  :
 alt {
   [] PCO3.receive {
        …
       }
   [] AnotherAltStep();
                        // 作为一个alt语句可选对象显性地调用可选步骤AnotherAltStep
   [] MyTimer.timeout {}
 }
```

在 **alt** 语句中显性调用可选步骤时，下一个要检查的选择对象是 **altstep** 的第一个选择对象。使用与带异常（当进入 **altstep** 时没有采用新的快照）的 **alt** 语句的选择对象（见第 20.1 节）一样的方法，来检查和执行 **altstep** 的选择对象。可选步骤的非成功终止（即 **altstep** 的所有顶层选择对象都做了检查，但未发现任何匹配分支）将引起对下一个选择对象的计算，或者调用缺省机制（如果显性调用是 **alt** 语句的最后一个选择对象）。一个成功的终止将引起测试部件的终止，即 **altstep** 以一个 **stop** 语句结束，或者引起 **alt** 语句的一个新快照和重新计算，即以 **repeat** 语句结束（见第 20.2 节），或者在 **alt** 语句后立即继续，即可选步骤的选定顶层选择对象不以显性的 **repeat** 或 **stop** 语句结束。

一个 **altstep** 也可以作为 TTCN-3 行为描述中的一个独立语句来调用。在这种情况下，对 **altstep** 的调用可解释为 **alt** 语句的简写形式，只有一个选择对象描述 **altstep** 的显性调用。

例 3：

```
// 语句
AnotherAltStep(); //   假设AnotherAltStep是一个正确定义的可选步骤

// 是下面alt语句的一个简写：

alt {
   [] AnotherAltStep();
}
```

16.3 用于不同部件类型的函数和可选步骤

请参见第 6.7.3 节。

17 测试用例

17.0 概述

测试用例是函数的一个特殊种类。在模块控制部分，使用 **execute** 语句来启动测试用例（见第 27.1 节）。一个测试用例的执行结果总是一个 **verdicttype** 类型的值。每个测试用例都应包含一个且仅一个 MTC，其类型在测试用例定义的头部进行引用。在测试用例主体中定义的行为为 MTC 行为。

当调用一个测试用例时，创建 MTC，并实例化 MTC 端口和测试系统接口，测试用例定义中指定的行为在 MTC 上进行启动。所有这些行为都将被隐性执行，也就是说，没有显性的 **create** 和 **start** 操作。

为提供信息以便允许发生这些隐性操作，一个测试用例头部应具有两部分内容：

a） 接口部分（强制的）：用关键字**runs on**表示，它为MTC引用所需的部件类型，并使相关的端口名称在MTC行为中是可见的；且

b） 测试系统部分（可选的）：用关键字**system**表示，它引用部件类型，用于为测试系统接口定义所需的端口。如果在测试执行期间只有MTC被实例化了，那么只能省略测试系统部分。在这种情况下，MTC类型隐性地定义测试系统接口端口。

例如：

```
testcase MyTestCaseOne()
runs on MyMtcType1          // 定义MTC类型
system MyTestSystemType     // 使TSI的端口名称对MTC是可见的。
{
   :   //   当测试用例被调用时，此处定义的行为在MTC上执行。
}

// 或者，一个仅有MTC被实例化的测试用例。
testcase MyTestCaseTwo() runs on MyMtcType2
{
   :   //当测试用例被调用时，此处定义的行为在MTC上执行。
}
```

17.1 测试用例的参数化

测试用例可以被参数化。应遵循如第 5.2 节中所定义的有关形参列表的规则。

18 程序语句和操作综述

测试用例、函数、可选步骤和 TTCN-3 模块控制部分的基本程序元素为表达式、基本的程序语句（如赋值、循环构造等）、行为语句（如顺序行为、选择对象行为、交叉、缺省等）以及操作（如 **send**、**receive**、**create** 等）。

语句既可以是单个语句（不包括其它程序语句），也可以是复合语句（可以包括其它语句以及语句和声明块）。

语句将按照其出现的次序进行执行，也就是说，按图 8 所示的顺序执行。

图 8/Z.140－顺序行为图示

在序列中，用分隔符“;”来分隔各个单独的语句。

例如：

```
MyPort.send(Mymessage); MyTimer.start; log("Done!");
```

对一个语句和声明空块的说明，即{}，可以出现在复合语句中，例如，**alt** 语句中的一个分支，并意味着不采取任何动作。

表 11/Z.140－TTCN-3表达式、语句和操作综述

语　　句	相关的关键字或符号	能用在模块控制中	能用在函数、测试用例和可选步骤中	能用在从模板、布尔防卫或可选步骤局部定义初始化调用的函数中
表达式	（…）	是	是	是
基本程序语句				
赋值	`:=`	是	是	是（见注 3）
日志	`log`	是	是	是
标记和跳转	`label / goto`	是	是	是
If-else	`if （…） {…} else {…}`	是	是	是
For 循环	`for （…） {…}`	是	是	是
While 循环	`while （…） {…}`	是	是	是
Do while 循环	`do {…} while （…）`	是	是	是
停止执行	`stop`	是	是	
Select case	`select case （…） { case （…） {…} case else {…}}`	是	是	是
行为程序语句				
选择对象行为	`alt {…}`	是（见注 1）	是	
重新确定选择对象行为	`repeat`	是（见注 1）	是	
交叉行为	`interleave {…}`	是（见注 1）	是	
返回控制	`return`		是（见注 4）	是
用于缺省处理的语句				
激活一个缺省	`activate`	是（见注 1）	是	
去激活一个缺省	`deactivate`	是（见注 1）	是	
配置操作				
创建并行测试部件	`create`		是	
连接两个部件	`connect`		是	
断开两个部件	`disconnect`		是	
映射端口到测试接口	`map`		是	
从测试系统接口取消端口映射	`unmap`		是	
获得 MTC 部件引用值	`mtc`		是	是
获得测试系统接口部件引用值	`system`		是	是
获得自身部件引用值	`self`		是	是
开始执行测试部件行为	`start`		是	
停止执行测试部件行为	`stop`		是	
从系统中移去一个测试部件	`kill`		是	
检查一个 PTC 行为是否已终止	`running`		是	
检查一个 PTC 是否存在于测试系统中	`alive`		是	
等待一个 PTC 行为的终止	`done`		是	

语　　句	相关的关键字或符号	能用在模块控制中	能用在函数、测试用例和可选步骤中	能用在从模板、布尔防卫或可选步骤局部定义初始化调用的函数中
等待一个 PTC 停止存在	**killed**		是	
通信操作				
发送消息	**send**		是	
调用过程调用	**call**		是	
从远程实体回答过程调用	**reply**		是	
引发异常（对一个已被接受的调用）	**raise**		是	
接收消息	**receive**		是	
触发消息	**trigger**		是	
从远程实体接受过程调用	**getcall**		是	
处理来自前一个调用的响应	**getreply**		是	
获取异常（从被调用的实体）	**catch**		是	
检查（当前）消息/接收到的调用	**check**		是	
清楚端口队列	**clear**		是	
清除队列，并使在端口上能够进行发送和接收	**start**		是	
使在端口上不能进行发送，并且不允许接收操作来匹配	**stop**		是	
使发送不能进行，并且不允许接收操作来匹配新的消息/调用	**halt**		是	
定时器操作				
启动定时器	**start**	是	是	
停止定时器	**stop**	是	是	
读取消逝的时间	**read**	是	是	
检查定时器是否在运行	**running**	是	是	
超时事件	**timeout**	是	是	
判定操作				
设置本地判定	**setverdict**		是	
获得本地判定	**getverdict**		是	是
外部活动				
外部模拟一个（SUT）活动	**action**	是	是	
测试用例执行				
测试用例执行	**execute**	是	是（见注 2）	

注 1 — 仅能用于控制定时器操作。

注 2 — 仅能用于在模块控制中使用的函数和可选步骤。

注 3 — 不允许改变部件变量。

注 4 — 能用于函数和可选步骤，但不能用于测试用例。

19 表达式和基本的程序语句

19.0 概述

表达式和基本的程序语句可用在模块的控制部分和 TTCN-3 函数、可选步骤和测试用例中。

表 12a/Z.140－TTCN-3基本的程序语句综述

基本的程序语句	
语句	**相关的关键字或符号**
赋值	**:=**
日志	**log**
Label 和 Goto	**label / goto**
If-else	**if (…) { … } else { … }**
For 循环	**for (…) { … }**
While 循环	**while (…) { … }**
Do while 循环	**do { … } while (…)**
Stop 执行	**stop**
Select case	**select case (…) { case (…) {…} case else {…} }**

19.1 表达式

19.1.0 概述

TTCN-3 允许表达式说明使用第 15 节中定义的运算符。表达式自其它（简单）表达式构建而来。表达式可以只使用值返回函数。一个表达式的结果应是一个特定类型的值，并且所用的运算符应与操作数类型兼容。

例如：

```
(x + y - increment(z))*3;
```

19.1.1 布尔表达式

一个 **boolean** 表达式应仅包含 **boolean** 值与/或 **boolean** 运算符与/或关系运算符，并应计算得到一个 **boolean** 值，为 **true** 或 **false**。

例如：

```
((A and B) or (not C) or (j<10));
```

19.2 赋值

值可以赋给变量，用符号“:=”表示。在执行赋值期间，赋值符号右侧将计算得到一个兼容于赋值符号左侧类型的值。赋值的结果是将一个变量与一个表达式的值绑定。表达式应该不包含任何未绑定的变量。所有赋值的发生次序均按照其出现的次序，也就是说，按照从左至右的次序进行处理。

例如：

```
MyVariable := (x + y - increment(z))*3;
```

19.3 log语句

log 语句提供了向测试控制部件或测试部件（在其中使用该语句）相关之某日志设备写入一个或多个日志条目的方法。在日志语句变元中，待写入的日志条目通过一个以逗号隔开的列表来确定。日志条目可以是在表 12b 中或由这些日志条目组成的表达式中的单个语言要素。

强烈建议日志语句的执行不应对测试行为产生任何影响。尤其是，日志语句中使用的函数不应显性或隐性地改变部件变量值、端口或定时器状态，并且不应改变其任何 **inout** 或 **out** 参数的值。

例如：

```
var integer myVar:= 1;
log("Line 248 in PTC_A: ", myVar, " (actual value of myVar)");
// 把串“Line 248 in PTC_A”写入测试系统的某个日志设备。
```

注 1 — 日志语句中使用的函数不应直接或间接地使用**if…else**、**for**、**while**、**do…while**、**label**、**goto**、**return**、**mtc**、**system**、**self**、**running**（PTC或定时器）、**read** 和 **getverdict**之外的语句。

注 2 — 定义复杂的日志和可能依赖工具的回溯能力，超出了本建议书的讨论范围。

表 12b/Z.140－可被日志记录的TTCN-3语言要素

在日志语句中使用	日志记录内容	注　　视
模块参数标识符	实际值	
字面值	值	这也包括自由文本。
数据常量标识符	实际值	
外部常量标识符	实际值	
模板实例	实际模板或字段值和匹配符号	
数据类型变量标识符	实际值或“UNITIALIZED”	见注 3 和注 4。
Self、**mtc**、**system** 或部件类型变量标识符	实际值以及若指定则为部件实例名称或“UNITIALIZED”	关于日志记录实际值，见注 2～注 4。将依据注 5 来日志记录实际部件状态。
运行操作（部件或定时器）	返回值	**true** 或 **false**。在部件或定时器数组情况下，将包括数组元素说明。
活动的操作（部件）	返回值	**true** 或 **false**。在数组情况下，将包括数组元素说明。
端口实例	实际状态	将依据注 6 来日志记录端口状态。
缺省类型变量标识符	实际状态或“UNITIALIZED”	将依据注 7 来日志记录缺省状态。也见注 2～注 4。
定时器名称	实际状态	将依据注 8 来日志记录定时器状态。
读操作	返回值	见第 24.3 节。
预定义函数	返回值	见附件 C。
函数实例	返回值	只允许带返回子句的函数。
外部函数实例	返回值	只允许带返回子句的外部函数。
形参标识符	见注释栏。	实参的日志记录将遵循为替代之语言要素所规定的规则。在值参数情况下，将日志记录实参值；在模板类型参数情况下，将日志记录实际模板或字段值以及匹配符号；在部件类型参数情况下，将日志记录实际部件引用等等。对定时器参数，也允许使用读操作；对部件类型和定时器参数，也允许使用运行操作。

表 12b/Z.140－可被日志记录的TTCN-3语言要素

注 1 — 实际值/模板为日志语句执行之时的值/模板。

注 2 — 日志记录值的类型依赖于工具。

注 3 — 在不带数组元素说明的数组标识符情况下，应日志记录实际值以及所有数组元素的部件引用名称。

注 4 — 当且仅当日志条目非绑定（非初始化）时才日志记录串“UNITIALIZED”。

注 5 — 可以日志记录的部件状态有：非活动的、运行的、停止的和断开的（更多的细节，见附件 F）。

注 6 — 可以日志记录的端口状态有：启动的、停止的（更多的细节，见附件 F）。

注 7 — 可以日志记录的缺省状态有：激活的和去激活的。

注 8 — 可以日志记录的定时器状态有：非活动的、运行的和到期的（更多的细节，见附件 F）。

19.4 label语句

label 语句允许在测试用例、函数、可选步骤和模块控制部分中对标签进行说明。可以根据附件 A 中定义的语法规则，像其它 TTCN-3 行为程序语句那样来自由地使用 **label** 语句。**label** 语句可以在一个 TTCN-3 语句之前或之后使用，但不能作为选择对象的第一个语句，或者 **alt** 语句、**interleave** 语句或 **altstep** 中的顶层选择对象。用在 **label** 关键字之后的标签在同一个测试用例、函数、可选步骤或控制部分中所定义的所有标签中都应是唯一的。

例如：

```
label MyLabel;                  // 定义标签MyLabel

// 在下面的TTCN-3代码段中定义标签L1、L2和L3
 :
label L1;                       // 标签L1的定义
alt{
[] PCO1.receive(MySig1)
   {   label L2;                // 标签L2的定义
       PCO1.send(MySig2);
       PCO1.receive(MySig3)
   }
[] PCO2.receive(MySig4)
   {   PCO2.send(MySig5);
       PCO2.send(MySig6);
       label L3;                // 标签L3的定义
       PCO2.receive(MySig7);
   }
}
 :
```

19.5 goto语句

goto 语句可以用在函数、测试用例、可选步骤和 TTCN-3 模块控制部分中。**goto** 语句执行一个跳转，跳至一个 **label** 处。

goto 语句提供自由跳转（即在一个语句序列内向前和向后跳转）、跳出一个单个复合语句（如一个 **while** 循环）和跳过若干层嵌套的复合语句的可能性（如嵌套的选择队形）。不过，**goto** 语句的使用将受到以下规则的限制：

a） 不允许使用**goto**语句跳出或跳入函数、测试用例、可选步骤和TTCN-3模块的控制部分。

b） 不允许使用**goto**语句跳入在一个复合语句（即**alt**语句、**while**循环、**for**循环、**if-else**语句、**do- while**循环和**interleave**语句）中定义的语句序列。

c） 不允许在一个**interleave**语句中使用**goto**语句。

例如：

```
// 下面的TTCN-3代码段包括：
 :
label L1;                              // … 标签L1的定义，
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }         // … 跳回到L1，
```

```
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }    // … 向前跳到L2，
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                           // … 标签L2的定义，
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
      label L3;                     // … 标签L3的定义，
      PCO2.send(MyVar);
      alt {
        [] PCO1.receive { }
        [] PCO2.receive(integer: 90) {
           PCO2.send(integer: 33);
           PCO2.receive(integer: 13);
           goto L4;                 // … 向前跳出两个嵌套的alt语句，
           }
        [] PCO2.receive(MyError) {
           goto L3;                 //…回跳，跳出当前的alt语句，
           }
        [] any port.receive {
           goto L2;                 // … 回跳，跳出两个嵌套的alt语句，
           }
      }
     }
  [] any port.receive {
      goto L2;                      // … 一个大的回跳，跳出alt语句。
     }
}
label L4;
 :
```

19.6 if-else 语句

if-else 语句，也就是通常所说的条件语句，用于表示 **boolean** 表达式导致的控制流中的分支，其用法如下所示：

```
if (expression1)
  statementblock1
else
  statementblock2
```

其中，$\mathtt{statementblock}_x$ 指的是一个语句块。

例如：

```
if (date == "1.1.2005") { return ( fail ); }

if (MyVar < 10) {
      MyVar := MyVar * 10;
      log ("MyVar < 10");
}
else {
  MyVar := MyVar/5;
}
```

一个更复杂的方案可以是：

```
if (expression1)
  statementblock1
else if (expression2)
  statementblock2
:
else if (expressionn)
  statementblockn
else
  statementblockn+1
```

在这种情况下，可读性主要依赖于格式，但格式没有任何语法或语义意义。

19.7 for语句

for 语句定义了计数器循环。增加、减少下标变量的值，或操纵下标变量的值，使其在执行一定次数的循环后到达终止条件。

for 语句包含两个赋值语句和一个 **boolean** 表达式。第一个赋值语句对初始化循环的下标（或计数器）变量是必需的，**boolean** 表达式用于终止循环，第二个赋值语句用于控制标变量。

例 1：

```
for (j:=1; j<=10; j:= j+1) { … }
```

循环的终止条件将由 **boolean** 表达式来表示，在每次新的循环反复开始之时，对布尔表达式进行检查。如果布尔表达式的值为 **true**，那么继续执行 **for** 语句中的语句块；如果布尔表达式的值为 **false**，那么继续执行紧跟在 **for** 循环后面的语句。

可以在 **for** 语句使用下标变量前声明该 **for** 循环语句的下标变量，或者在 **for** 语句头部中声明并初始化该下标变量。如果下标变量在 **for** 语句头部中进行声明并初始化，那么下标变量的作用范围仅限于该循环体内，即它仅在该循环体内是可见的。

例 2：

```
var integer j;                          // 声明整数变量j
for (j:=1; j<=10; j:= j+1) { … }    //   作为for循环下标变量的j的用法


for (var float i:=1.0; i<7.9; i:= i*1.35) { … }
                                                    // 在for循环头部声明和初始化下标变量i。
                                                    // 变量i仅在循环体内是可见的。
```

19.8 while语句

只要满足循环条件，就一直执行 **while** 循环。在每次新的循环反复开始之时，对循环条件进行检查。如果不满足循环条件了，那么退出循环，并继续执行紧跟在 **while** 循环后面的语句。

例如：

```
while (j<10){ … }
```

19.9 do-while语句

do-while 循环与 **while** 循环类似，只是 **do-while** 循环在每次循环反复结束之时对循环条件进行检查，这就意味着，当使用 **do-while** 循环时，在第一次对循环条件进行判断之前，循环体的行为将至少执行一次。

例如：

```
do { … } while (j<10);
```

19.10 stop执行语句

根据 **stop** 语句使用情况的不同，**stop** 语句以不同的方式来终止执行操作。当在一个模块的控制部分或模块控制部分使用的函数中使用 **stop** 语句时，它终止模块控制部分的执行。当在一个测试部件上执行的测试用例、可选步骤或函数中使用 **stop** 语句时，它终止相关的测试部件。

例如：

```
module MyModule {
    : //   模块定义
   testcase MyTestCase() runs on MyMTCType system MySystemType{
      var MyPTCType ptc:= MyPTCType.create; // PTC创建
      ptc.start(MyFunction());              // 开始执行PTC
       :        // 继续测试用例行为
      stop      // 停止MTC、所有PTC和整个测试用例
   }
   function MyFunction() runs on MyPTCType {
        :
      stop  // 只停止PTC，测试用例继续。
   }
   control {
        :   //  测试执行
      stop //  停止测试活动
   } //结束控制
} //  结束模块
```

注 — 停止一个测试部件的**stop**语句的语义与停止部件操作的**self.stop**的语义是相同的（见第22.6节）。

19.11 select case语句

当一个值与一个或几个其它值进行比较时，**select case** 语句是 **if-else** 语句的一个可选方案。语句包含一个头部和零个或多个分支。从不会执行多个分支。**select case** 语句用法如下所示：

```
select (expression)
   {
   case (templateInstance1a, templateInstance1b,…)
      statementblock1
   case (templateInstance2a, templateInstance2b,…)
      statementblock2
   case else
      statementblock3
   }
```

其中，templateInstance 指的是一个已定义的或内置的模板，statementblock_X 指的是一个语句块。

注 — 上述示意性的语句块等同于以下使用**if-else**语句的示意性的语句块。

```
if (match(expression, templateInstance1a or match(expression, templateInstance1b
or …)
   statementblock1
else if (match(expression, templateInstance2a or match(expression, templateInstance2b or …)
   statementblock2
else
   statementblock3
```

在 select case 语句头部，会提供一个表达式。每个分支以关键字 **case** 开始，后跟一个 templateInstance 列表（一个列表分支，它也可以包含一个单个元素），或者以关键字 **else** 开始（一个 **else** 分支）以及一个语句块。

所有列表分支中的所有 templateInstance 其类型都应兼容于头部中表达式的类型。当且仅当某个 templateInstance 匹配语句头部中表达式的值时，才选中一个列表分支以及选中分支的语句块。执行选中分支的语句块后（即不以 go to 语句跳出），继续执行 **select case** 语句后的语句。

如果未选中 **else** 分支之前的任何其它分支，那么总执行 **else** 分支的语句块。

按其文本次序对各分支进行判断。如果没有任何 templateInstance 匹配头部中表达式的值，且语句不包含 **else** 分支，那么继续执行下面的语句，而不执行任何 **select case** 分支。

例如：

```
select (MyModulePar)  // MyModulePar为charstring类型
{
  case ("firstValue")
      {
       log ("The first branch is selected");
      }
  case (MyCharVar, MyCharConst)
      {
       log ("The second branch is selected");
      }
  case else
      {
       log ("The value of the module parameter MyModulePar is selected");
      }
}
```

20 行为的程序语句

20.0 概述

行为的程序语句可用在测试用例、函数和可选步骤中，除了：

a） 仅用在函数中的**return**语句；以及

b） 也可用在模块控制中的**alt**语句、**interleave**语句和**repeat**语句。

行为的程序语句用于描述经由通信端口的测试部件的动态行为。可以作为选择对象的一个集合或二者的组合来顺序描述测试行为。一个交叉运算符允许对交叉序列或选择对象作出说明。

表 13/Z.140－TTCN-3行为的程序语句综述

行为的程序语句	
语句	**相关的关键字或符号**
选择性行为	**alt { … }**
alt 语句的重新计算	**repeat**
交叉行为	**interleave { … }**
返回控制	**return**

20.1 选择性行为

20.1.0 概述

行为的一个更为复杂的形式是，语句序列被表示为形成执行路径树的、可能的选择对象的集合，如图 9 所示：

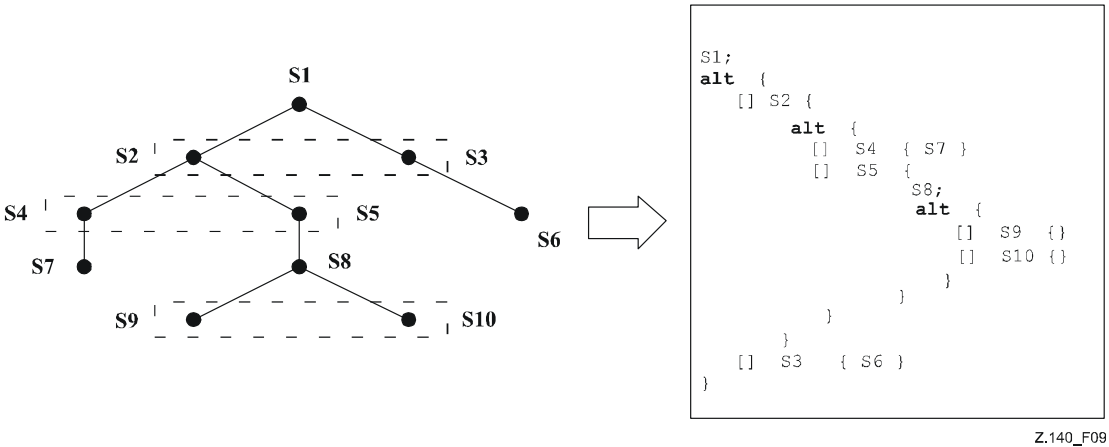


图 9/Z.140－选择性行为描述

alt 语句表示因通信接收和处理与/或定时器事件与/或并行测试部件终止而引起的测试行为分岔，也就是说，它与 TTCN-3 的 **receive**、**trigger**、**getcall**、**getreply**、**catch**、**check**、**timeout**、**done** 和 **killed** 操作有关。**alt** 语句表示将与一个特定快照相匹配的可能事件集（见第 20.1.1 节）。

例如：

```
//嵌套的选择语句的用法：
 :
alt {
  [] L1.receive(DL_REL_CO:*) {
      setverdict(pass);
      TAC.stop;
      TNOAC.start;
      alt {
        [] L1.receive(t_DL_EST_IN) {
             TNOAC.stop;
             setverdict(pass);
           }
        [] TNOAC.timeout {
           L1.send(t_DEL_EST_RQ);
           TAC.start;
           alt {
             [] L1.receive(DL_EST_CO:*) {
                  TAC.stop;
                  setverdict(pass)
              }
              [] TAC.timeout {
                  setverdict(inconc);
                }
              [] L1.receive {
                  setverdict(inconc)
                }
           }
       }
         [] L1.receive {
              setverdict(inconc)
           }
      }
     }
  [] TAC.timeout {
      setverdict(inconc)
     }
  [] L1.receive {
      setverdict(inconc)
     }
}
 :
```

20.1.1 选择行为的执行

当进入一个 **alt** 语句时，照一张快照。快照被认为是测试部件的一部分状态，测试部件包括：计算防卫选择对象分支的布尔条件所需的所有信息，所有相关的被终止的测试部件，所有相关的超时事件和顶层消息、调用、答复，以及相关输入端口队列中的异常。在 **alt** 语句的至少一个选择对象中被引用的任何测试部件、定时器和端口，或者在 **alt** 语句中作为一个选择对象被调用的可选步骤的顶层步骤中的任何测试部件、定时器和端口，或者作为缺省被激活的可选步骤的顶层步骤中的任何测试部件、定时器和端口，被认为是相关的。快照语义的详细描述在 TTCN-3 的操作语义中给出（ITU-T Z.143 建议书 [3]）。

注 1 — 快照只是用于描述**alt**语句行为的一个概念性方法，用于快照处理的具体算法可在ITU-T Z.134建议书 [3] 中找到。

注 2 — TTCN-3语义假设照快照是瞬间完成的，即没有延迟。在实际的实现中，照快照可能会占用一些时间，并可能出现竞争条件。此类竞争条件的处理超出了本建议书的讨论范围。

alt 语句中的选择对象分支和被调用可选步骤中的顶层选择对象以及被激活为缺省的可选步骤按其出现次序进行处理。如果激活了若干缺省，那么其激活次序的逆序确定缺省中顶层选择对象的计算次序。通过第 21 节中所述的缺省机制来实现活动缺省中的选择对象分支。

各选择对象分支是由布尔表达式或 else 分支防卫的分支，即选择对象分支以[**else**]开始。

在到达 else 分支时，总是选中和执行它们（见第 20.1.3 节）。

可以通过一个布尔表达式调用一个可选步骤（可选步骤分支），或者以 **done** 操作（*done 分支*）、**killed** 操作（*killed 分支*）、**timeout** 操作（*timeout 分支*）或接收操作（*接收分支*）开始，即 **receive**、**trigger**、**getcall**、**getrepy**、**catch** 或 **check** 操作，来防卫各分支。布尔防卫的计算将基于快照。如果没有定义任何布尔防卫，或者布尔防卫计算值为 **true**，那么认为满足布尔防卫条件。按以下方式处理和执行各分支。

如果实现了布尔防卫，那么选中一个可选步骤分支。一个可选步骤分支的选择会引起对所引用可选步骤的调用，即调用该可选步骤，且在该可选步骤内继续进行对快照的计算。

如果实现了布尔防卫，并且如果指定的测试部件在快照的被停止部件列表中，那么选中一个 **done** 分支。该选择会引起执行 **done** 操作后的语句块。**done** 操作本身没有更进一步的影响。

如果实现了布尔防卫，并且如果指定的测试部件在快照的 **killed** 部件列表中，那么选中一个 **killed** 分支。该选择会引起执行 **killed** 操作后的语句块。**killed** 操作本身没有更进一步的影响。

如果实现了布尔防卫，并且如果指定的超时事件在快照的超时事件列表中，那么选中一个超时分支。该选择会引起执行指定的 **timeout** 操作，也就是说，从超时队列中移去超时事件，并执行 **timeout** 操作后的语句块。

如果实现了布尔防卫，并且如果快照中的一个消息、调用、应答或异常与接收操作的匹配准则相匹配，那么选中一个接收分支。该选择会导致执行接收操作，即从端口队列中移去相匹配的消息、调用、应答或异常，或许是将接收到的消息赋值给一个变量，并执行接收操作后的语句块。在 **trigger** 操作的情况下，如果实现了布尔防卫，但不符合匹配准则，那么也从端口队列中移去度列的顶层消息。在这种情况下，不执行给定选择对象的语句块。

注 3 — TTCN-3语义描述了对作为测试部件一系列不可分割活动的快照的计算。该语义并不假设对快照计算没有任何延时。在对快照进行计算过程中，测试部件可以停止，定时器可以超时，新的消息、调用、应答或异常可以进入该部件的端口队列中。不过，这些事件并不改变实际的快照，因此对快照计算并不考虑这些事件。

如果在 **alt** 语句中没有任何选择对象分支、在被调用的可选步骤中没有任何顶层选择对象以及没有任何活动缺省被选中和执行，那么应再次执行 **alt** 语句，也就是说，照一个新的快照，并以新的快照重复对选择对象分支的计算。该重复计算过程应继续进行，直至选中和执行一个选择对象分支，或者该测试用例被另一个部件或测试系统（例如，由于 MTC 被停止）或动态错误停止。

如果测试部件被完全阻塞，那么测试用例应停止并指示一个动态错误。这意味着没有任何选择对象可被选中，没有任何相关的测试部件和定时器在执行，所有的相关端口都包含至少一个不匹配的消息、调用、应答或异常。

注 4 — 照完整快照和对所有选择对象重新计算的反复过程对描述**alt**语句的语义而言只是一个概念性的方法。实现该语义的具体算法超出了本建议书的讨论范围。

20.1.2 选择对象的选择和取消选择

如果需要的话，有可能利用置于选择对象方括号“[ ]”之间的一个布尔表达式来启动/禁止一个选择对象。

对防卫一个选择对象的布尔表达式进行计算可能会有副作用。为了避免可能引起实际快照与部件状态之间不一致性的这种副作用，应将相同的限制作为可选步骤内局部定义初始化的限制（见第 16.2.2.1 节）。

方括号的开“[”和关“]”应在每个选择对象的开始之处出现，即使它们是空的。这不仅有助于可读性，对从语法上区分一个选择对象和另一个选择对象也是必要的。

例如：

```
// 带布尔表达式（防卫）的选择对象的用法
 :
alt {
  [x>1] L2.receive {            // 布尔防卫/表达式
        setverdict(pass);
     }
  [x<=1] L2.receive {           // 布尔防卫/表达式
         setverdict(inconc);
     }
}
 :
```

20.1.3 选择对象中的else分支

可以通过在选择对象开始之处的开和关括号之间包括关键字 **else**，来将 **alt** 语句中的任何分支定义为一个 else 分支。该 else 分支不得包含布尔表达式防卫的分支中允许的任何动作（即 **altsetp** 调用、**done** 操作、**killed** 操作、**timeout** 操作或接收操作）。如果在该 else 分支之前文本上未处理过任何其它选择对象，那么总执行该 else 分支的语句块。

例如：

```
// 带布尔表达式（防卫）的选择对象的用法
 :
alt {
  [x>1] L2.receive {
      setverdict(pass);
     }
  [x<=1] L2.receive {
      setverdict(inconc);
     }
  [else] {              // else分支
      MyErrorHandling();
      setverdict(fail);
      stop;
     }
}
 :
```

应该注意，缺省机制（见第 21 节）总是在所有选择对象结束之时才被调用。如果定义了一个 **else** 分支，那么将不调用缺省机制，也就是说，绝不会进入活动的缺省。

注 1 — 也有可能在可选步骤内使用**else**。

注 2 — 允许在一个**else**分支中使用一个**repeat**语句。

注 3 — 允许在一个alt语句或一个可选步骤中定义多个else分支，不过，总是只执行第一个else分支。

20.1.4 空缺

20.1.5 alt语句的重新求值

可以使用一个 **repeat** 语句来描述对 **alt** 语句的重新求值（见第 20.2 节）。

例如：

```
alt {
  [] PCO3.receive {
      count := count + 1;
      repeat                    // repeat的用法
     }
  [] T1.timeout { }
  [] any port.receive {
      setverdict(fail);
      stop;
     }
}
```

20.1.6 当作选择对象来调用可选步骤

TTCN-3 允许在 **alt** 语句中作为选择对象来调用可选步骤（见第 16.2.3 节）。

例如：

```
  :
alt {
  [] PCO3.receive { }
  [] AnotherAltStep();  // 对作为alt语句选择对象的可选步骤AnotherAltStep的显性调用
  [] MyTimer.timeout { }
}
  :
```

20.2 repeat语句

当用在语句块和 **alt** 语句选择对象的声明中时，**repeat** 语句将引起对 **alt** 语句的重新计算，即照一个新的快照，并按其说明次序对 **alt** 语句的选择对象进行计算。当用在语句块、响应声明以及块程序调用的异常处理部分中时，repeat 以及将引起对响应和调用异常处理部分的重新计算（见第 23.3.1.5 节）。

例 1：

```
// alt语句中repeat的用法
alt {
  [] PCO3.receive {
      count := count + 1;
      repeat            //repeat的用法
     }
  [] T1.timeout { }
  [] any port.receive {
      setverdict(fail);
      stop;
     }
}
```

如果 **repeat** 语句用在可选步骤定义中的顶层选择对象中，那么它将引起一个新的快照，并对调用可选步骤的 **alt** 语句重新进行计算。可以通过缺省机制隐性地调用可选步骤（见第 21 节），也可以在 **alt** 语句中显性地调用可选步骤（见第 20.1.6 节）。

例 2：

```
//  可选步骤中repeat的用法
altstep AnotherAltStep() runs on MyComponentType {
  [] PCO1.receive{
      setverdict(inconc);
      repeat                  // repeat的用法
     }
  [] PCO2.receive {}
}
```

20.3 交叉行为

interleave 语句允许规定 **done**、**killed**、**timeout**、**receive**、**trigger**、**getcall**、**catch** 和 **check** 语句的交叉发生和处理。

包括通信操作的控制转移语句（**for**、**while**、**do-while**、**goto**、**activate**、**deactivate**、**stop**、**repeat**、**return**）作为选择对象的可选步骤的直接调用和用户定义的函数的（直接和间接）调用，不得用在 **interleave** 语句中。此外，不允许用布尔表达式来防卫 **interleave** 语句的分支（即“[ ]”应总是空的），也不允许在交叉的行为中指定 **else** 分支。

交叉的行为总可以被一个等价的嵌套选择对象集所代替。该代替过程以及交叉的操作语义在 ITU-T Z.143 建议书[3]中描述。

交叉语句求值的规则如下所述：

a） 不论何时执行一个接收语句，只有到达下一个接收语句或交叉序列结束时，才随后执行之后的非接收语句；

注 — 接收语句是可以在选择对象集合中出现的TTCN-3语句，即**receive**、**check**、**trigger**、**getcall**、**getreply**、**catch**、**done**、**killed**和**timeout**。非接收语句指的是可用在**interleave**语句内的所有其它非控制转移语句。

b） 照下下一个快照后继续进行计算。

交叉的操作语义完整地定义在 ITU-T Z.143 建议书 [3]中。

例如：

```
// 下面的TTCN-3代码段：
 :
interleave {
[] PCO1.receive(MySig1)
   {   PCO1.send(MySig2);
       PCO1.receive(MySig3);
   }
[] PCO2.receive(MySig4)
   {   PCO2.send(MySig5);
       PCO2.send(MySig6);
       PCO2.receive(MySig7);
   }
}
:

// 可以解释为下面代码段的简写：
:
alt {
[] PCO1.receive(MySig1)
   {   PCO1.send(MySig2);
       alt {
       []   PCO1.receive(MySig3)
       {    PCO2.receive(MySig4);
            PCO2.send(MySig5);
            PCO2.send(MySig6);
            PCO2.receive(MySig7)
       }
       []   PCO2.receive(MySig4)
       {    PCO2.send(MySig5);
            PCO2.send(MySig6);
            alt {
            []   PCO1.receive(MySig3) {
                 PCO2.receive(MySig7); }
            []   PCO2.receive(MySig7) {
                 PCO1.receive(MySig3); }
            }
       }
       }
   }
[] PCO2.receive(MySig4)
   {   PCO2.send(MySig5);
       PCO2.send(MySig6);
       alt {
       []   PCO1.receive(MySig1)
       {    PCO1.send(MySig2);
            alt {
            []   PCO1.receive(MySig3)
              {  PCO2.receive(MySig7);
              }
            []   PCO2.receive(MySig7)
              {  PCO1.receive(MySig3);
              }
            }
       }
       []   PCO2.receive(MySig7)
       {    PCO1.receive(MySig1);
            PCO1.send(MySig2);
            PCO1.receive(MySig3);
       }
       }
   }
}

:
```

20.4 return语句

return 语句终止函数或可选步骤的执行，并把控制返回到函数或可选步骤的调用点。当用在函数中时，**return** 语句可以选择是否与一个返回值关联。

注 — 当用在可选步骤中时，**return**语句的作用仿佛是结束语句块以及达到所选选择对象的声明；例如，当从一个**alt**语句调用可选步骤时，从**alt**语句后的第一个语句开始继续执行。

例如：

```
function MyFunction() return boolean {
    :
  if (date == "1.1.2005") {
      return false; // 在1.1.2000执行结束，并返回一个布尔值false。
  }
   :
  return true; //返回true。
}

function MyBehaviour() return verdicttype {
    :
  if (MyFunction()) {
      setverdict(pass); // 在if语句中MyFunction的使用。
  }
  else {
      setverdict(inconc);
  }
   :
  return getverdict; //判定的显性返回。
}
```

21 缺省处理

21.0 概述

TTCN-3 允许激活可选步骤作为缺省（见第 16.2 节）。对每个测试部件，以有序列表的形式来存储缺省（即激活的可选步骤）。按其激活次序的逆序列出各缺省，即最后一个激活的缺省为活动缺省列表的第一个元素。TTCN-3 操作 **activate**（见第 21.3 节） 和 **deactivate**（见第 21.4 节）对缺省列表进行操作。**activate** 语句添加一个新的缺省到列表中，作为第一个元素，**deactivate** 语句从列表中移去一个缺省。缺省列表中的一个缺省可以通过一个缺省引用的方式来确定，它作为对应 **activate** 操作的结果来产生。

表 14/Z.140－用于缺省处理的TTCN-3语句综述

用于缺省处理的语句	
语句	**相关的关键字或符号**
激活一个缺省	**Activate**
停用一个缺省	**deactivate**

21.1 缺省机制

如果由于实际的快照而导致任何指定的选择对象都不能被执行，那么在各 **alt** 语句结束处唤醒缺省机制。被唤醒的缺省机制调用缺省列表中的第一个可选步骤，即最后一个激活的缺省，并等待其终止结果。可以是成功的终止，也可以是不成功的终止。不成功的终止意味着不能选中任何定义缺省行为的 **altstep** 的顶层选择对象（见第 16.2 节），成功的终止意味着已经选中并执行缺省的一个顶层选择对象。

在不成功终止情况下，缺省机制调用列表中的下一个缺省。如果列表中的最后一个缺省未成功终止，那么缺省机制将返回到 **alt** 语句中它的调用点，也就是说，返回到 **alt** 语句的结尾，并指示一个不成功的缺省执行。如果缺省列表为空，那么也将指示一个不成功的缺省执行。

如果测试部件被阻塞，那么一个不成功的缺省执行可能导致一个新的快照或一个动态的错误（见第 20.1 节）。

在成功终止情况下，缺省可以利用一个 **stop** 语句来停止测试部件，或者在调用缺省机制的 **alt** 语句之后立即继续执行测试部件的主控制流，或者测试部件照一个新的快照并对 **alt** 语句重新进行计算。后者需要用 **repeat** 语句进行说明（见第 20.2 节）。如果缺省选中的顶层选择对象不以 **repeat** 语句结束，那么在 **alt** 语句之后立即继续执行测试部件的控制流。

注 — TTCN-3不限制缺省机制的实现。例如，它可以在每个alt语句的结尾处以隐性被调用的进程形式实现，也可以以只对缺省处理负责的单独线程形式实现。当缺省机制已被调用时，唯一的要求是按其激活次序的逆序来调用缺省。

21.2 缺省引用

缺省引用是对已激活缺省的唯一引用，这样的唯一缺省引用是在一个可选步骤作为一个缺省而被激活时产生的，也就是说，一个缺省引用是一个 **activate** 操作的结果（见第 21.3 节）。

缺省引用具有特殊的和预定义的类型 **default**。**default** 类型变量可以用于处理测试部件中的已激活缺省。特殊的 **null** 值可用于指明未定义的缺省引用，例如，用于处理缺省引用的变量初始化。

缺省引用用在 **deactivate** 操作中（见第 21.4 节），以便确定要被去激活的缺省。

将由测试系统在外部解析 **default** 类型的实际数据表示。这允许对抽象测试用例的说明独立于任何实际的 TTCN-3 运行环境，换句话说，TTCN-3 不限制有关缺省处理和识别的测试系统的实现。

例如：

```
// 用于缺省处理的变量声明，并以空值进行初始化。
var default MyDefaultVar := null;
 :
// 用于保存一个激活缺省的MyDefaultVar的用法。
MyDefaultVar := activate(MyDefAltStep()); // 将MyDefAltStep激活为缺省。
 :
// 用于去激活缺省MyDefAltStep的MyDefaultVar的用法。
deactivate(MyDefaultVar);
 :
```

21.3 激活操作

21.3.0 概述

activate 操作用于将可选步骤激活为缺省。**activate** 操作将把被引用的可选步骤作为第一个元素添加到缺省列表中，并返回一个缺省引用。缺省引用是用于缺省的一个唯一标识符，并可以用在 **deactivate** 操作中用于去激活缺省。

activate 操作的影响局限于调用它的测试部件，这意味着，一个测试部件不能激活另一个测试部件中的缺省。

例 1：

```
 :
// 用于缺省处理的变量声明。
var default MyDefaultVar := null;
 :
// 一个缺省引用变量的声明，以及激活一个可选步骤作为缺省。
var default MyDefVarTwo := activate(MySecondAltStep());
 :
// 激活可选步骤MyAltStep作为激活。
MyDefaultVar := activate(MyAltStep()); // 将MyAltStep激活为缺省。
 :
```

无需保存返回的缺省引用就可以调用操作。在不需要显性对已激活缺省去激活的测试用例中，这种形式是有用的；即在 MTC 终止时隐性地完成对缺省的去激活。

例 2：

```
 :
// 激活一个可选步骤作为缺省，不带缺省引用赋值。
activate(MyCommonDefault());
```

21.3.1 激活参数化可选步骤

将在相应的 **activate** 语句中提供应被激活为缺省的参数化可选步骤的实参（见第 16.2.1 节），这意味着，在缺省激活时，实参帮定于缺省绑定（例如，不是在它被缺省机制调用时）。实参列表中的所有定时器实例将被声明为部件类型局部定时器（见第 8.5.1 节）。

例如：

```
altstep MyAltStep2 ( integer    par_value1, MyType par_value2,
                     MyPortType par_port,   timer  par_timer )
{
 :
}

function MyFunc () runs on MyCompType
{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);
   //  MyAltStep2被激活为缺省，带实参5和MyVar值。
   //  通过缺省机制调用MyAltStep2前，myVar的变化将不改变调用的实参。
 :
}
```

21.4 去激活操作

deactivate 操作用于去激活缺省，即之前被激活的可选步骤。**deactivate** 操作将从缺省列表中移去被引用的缺省。

deactivate 操作的影响局限于调用它的测试部件，这意味着，一个测试部件不能去激活另一个测试部件中的缺省。

不带参数的 **deactivate** 操作去激活一个测试部件的所有缺省。

调用一个带特殊值 **null** 的 **deactivate** 操作没有任何作用。调用一个带未定义缺省引用的 **deactivate** 操作，如一个已被去激活缺省的旧引用或未被初始化的缺省引用变量，将导致一个运行错误。

例如：

```
 :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
deactivate(MyDefaultVar); // deactivates MyAltStep   // 去激活MyAltStep。
 :
deactivate; // 去激活所有其它的缺省，即在这种情况下的MySecondAltStep和MyThirdAltStep。
 :
```

22 配置操作

22.0 概述

配置操作（见表 15）用于建立和控制测试部件。这些操作应仅用在 TTCN-3 测试用例、函数和可选步骤中（也就是说，不能用在模块控制部分中）。

表 15/Z.140－TTCN-3配置操作综述

操　作	解　释	语法举例
连接操作		
connect	将一个测试部件的端口连接至另一个测试部件的端口	**connect**(ptc1:p1, ptc2:p2);
disconnect	断开两个或更多个连接的端口	**disconnect**(ptc1:p1, ptc2:p2);
map	将一个测试部件的端口映射至测试系统接口的端口	**map**(ptc1:q, **system**:sutPort1);
unmap	去除两个或更多个映射的端口	**unmap**(ptc1:q, **system**:sutPort1);

表 15/Z.140－TTCN-3配置操作综述

操　作	解　释	语法举例
测试部件操作		
create	创建一个普通的或活动的测试部件；在创建期间区分普通的测试部件和活动的测试部件（MTC 作为普通的测试部件）。	非活动的测试部件： **var** PTCType c := PTCType.**create**; 活动的测试部件： **var** PTCType c := PTCType.**create alive**;
start	启动一个测试部件上的测试行为。启动一个行为不会影响部件变量、定时器或端口的状态。	c.**start**(PTCBehaviour());
stop	停止一个测试部件上的测试行为。	c.**stop**;
kill	致使一个测试部件停止存在。	c.**kill**;
alive	如果已经创建测试部件且准备执行或已经执行一个行为，那么返回 true；否则返回 false。	**if** (c.**alive**) …
running	只要测试部件执行一个行为，那么返回true；否则返回 false。	if (c.**running**) …
done	检查运行于一个测试部件上的函数是否已经终止。	c.**done**;
killed	检查一个测试部件是否已经停止存在。	c.**killed** { … }
引用操作		
mtc	获得到 MTC 的引用	**connect**(**mtc**:p, ptc:p);
system	获得到测试系统接口的引用	**map**(c:p, **system**:sutPort);
self	获得到执行本操作的测试部件的引用	**self.stop;**

22.1　创建操作

MTC 是测试用例启动时自动创建的唯一一个测试部件，所有其它测试部件（PTC）都应在测试执行期间用 **create** 操作显性地创建。一个部件以其输入队列为空的端口全集以及其常量、变量和定时器的全集来创建。此外，如果定义的端口类型为 **in** 或 **inout**，那么它将处于监听状态，随时准备接收连接上的通信流量。

当显性或隐性创建部件时，所有部件变量和定时器将重新设置为其初始值（如果有的话），所有部件常量将重新设置为其指定值。

区分两种类型 PTC：一种 PTC 只能执行一次行为函数，一种 PTC 在行为函数终止后继续保持活动，并因此可以重新用于执行另一个函数。后者使用额外的 **alive** 关键字来创建。必须使用 **kill** 操作来显性地销毁一个活动类型的 PTC（见第 22.9 节），而对非活动的 PTC 则在其行为函数终止后隐性地予以销毁。测试用例的终止，即 MTC，将终止依然存在的所有 PTC（如果有的话）。

由于每个测试用例终止时隐性地销毁所有的测试部件，因此在调用每个测试用例时，应彻底创建它所需的部件和连接配置。

create 操作将返回新建实例的唯一的部件引用。典型地，该部件的唯一引用将保存在一个变量中（见第 8.7 节），且可用于连接实例和通信目的，如发送和接收。

可选地，一个名称可与新创建的部件实例相关联。名称应是一个 **charstring** 值，当指派时，它将作为 **create** 函数的变元出现。测试系统将在创建时自动把名称“MTC”关联至 MTC，把名称“SYSTEM”关联至测试系统接口。不要求关联的部件名称是唯一的。

注 — 部件实例名称只用于日志记录目的（见第19.3节），不得用于指部件实例（部件引用将用于该目的），并对匹配没有任何影响。

例如：

```
// 该例子用于声明MyComponentType类型的变量，用来存储MyComponentType类型新建测试实例的引用，
// 它是创建操作的结果。对某些已创建的部件实例，分配一个相关的名称。
 :
var MyComponentType MyNewComponent;
var MyComponentType MyNewestComponent;
var MyComponentType MyAliveComponent;
var MyComponentType MyAnotherAliveComponent;
 :
MyNewComponent := MyComponentType.create;
MyNewestComponent := MyComponentType.create("Newest");
MyAliveComponent := MyComponentType.create alive;
MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
 :
```

可以在行为定义的任何地方创建部件，提供有关动态配置的全部灵活特性（即任何部件都可以创建任何其它PTC）。部件引用的可见性应遵循与变量相同的范围规则，为了能够在部件创建范围之外引用它们，部件引用应作为消息中的一个参数或一个字段来传递。

22.2 连接和映射操作

22.2.0 概述

一个测试部件的端口可以连接到其它部件或测试系统接口的端口。在两个测试部件连接的情况下，将使用 **connect** 操作。当将一个测试部件连接到一个测试系统接口时，将使用 **map** 操作。**connect** 操作直接将一个端口连接到另一个端口，即将一个端口的 **in** 端连接至另一个端口的 **out** 端，反之亦然。另一方面，**map** 操作可以完全看作是定义应如何引用通信流的名称转换操作。

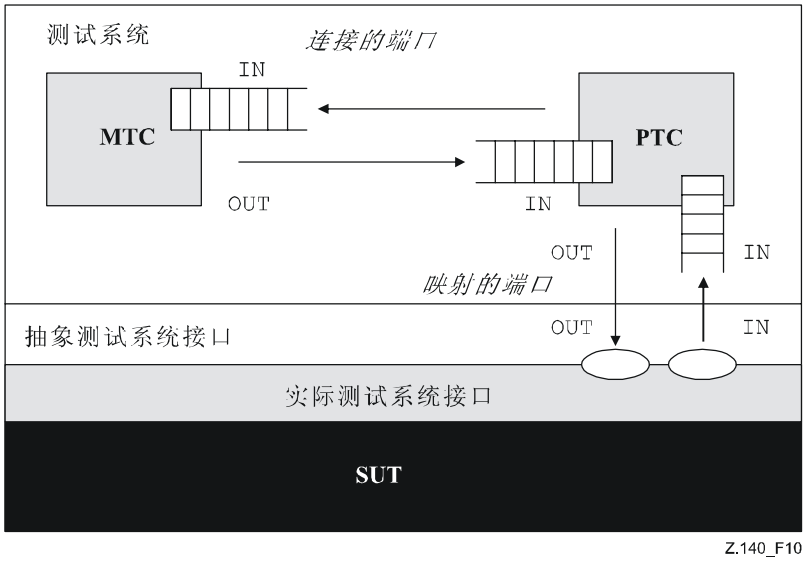


图 10/Z.140－连接和映射操作图示

利用 **connect** 操作和 **map** 操作，通过对连接部件的部件引用和连接端口的名称来确定要连接的端口。

mtc 操作用于确定 MTC，**system** 操作用于确定测试系统接口（见第 22.4 节），**self** 操作用于确定引用 **self** 的测试部件（见第 22.4 节）。所有这些操作都可用于确定和连接端口。

除了模块的控制部分，**connect** 和 **map** 操作可以被任何行为定义所调用。不过，在它们被调用之前，要连接的部件应已创建，且它们的部件引用和相关端口的名称都应是已知的。

map 和 **connect** 操作都允许把一个端口连接至多个其它端口，但不允许连接至一个映射端口或映射至一个连接端口。

例如：

```
// 假设在相应的端口类型和部件类型定义中正确地定义和声明了端口Port1、Port2、Port3和PCO1。
 :
var MyComponentType  MyNewPTC;
 :
MyNewPTC := MyComponentType.create;
 :
 :
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
 :
 :
// 在本例中，创建了一个MyComponentType类型的新部件，并将对它的引用存储在变量MyNewPTC中。
// 然后，在连接操作中，该新部件的Port1端口与MTC的Port3端口连接。
// 然后，利用映射操作，新部件的Port2端口连接至测试系统接口的PCO1端口。
```

22.2.1 一致的连接和映射

对 **connect** 和 **map** 操作，都只允许一致的连接。

假设以下各项：

a） 要连接的端口为端口PORT1和端口PORT2；

b） inlist-PORT1定义端口PORT1输入方向的消息或过程；

c） outlist-PORT1定义端口PORT1输出方向的消息或过程；

d） inlist-PORT2定义端口PORT2输入方向的消息或过程；以及

e） outlist-PORT2定义端口PORT2输出方向的消息或过程。

当且仅当满足以下条件时，才允许进行 **connect** 操作：

- outlist-PORT1 ⊆ inlist-PORT2 且 outlist-PORT2 ⊆ inlist-PORT1;

当且仅当满足以下条件时，才允许进行 **map** 操作（假设 PORT2 为测试系统接口端口）：

- outlist-PORT1 ⊆ outlist-PORT2 且 inlist-PORT2 ⊆ inlist-PORT1.

在所有其它情况下，都不允许 **connect** 和 **map** 操作。

由于 TTCN-3 允许动态配置和寻址，因此在编译时，不是所有的这些一致性检查都可以静态地执行。所有不能在编译时执行的一致性检查，都应在运行时执行，且在发生错误时将导致一个测试用例错误。

22.3 断开连接和取消映射操作

disconnect 和 **unmap** 操作是 **connect** 和 **map** 操作的反操作。它们执行操作来断开（之前连接的）测试部件端口的连接，取消（之前映射的）测试部件端口与测试系统接口中端口之间的映射。

如果相关的部件引用和相关的端口名称是已知的，那么任何部件都可以调用 **disconnect** 和 **unmap** 操作。如果要移去的连接或映射已事先创建，那么 **disconnect** 或 **unmap** 操作只有一个作用。

例 1：

```
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
disconnect(MyNewComponent:Port1, mtc:Port3);    // 断开之前建立的连接。
unmap(MyNewComponent:Port2, system:PCO1);       // 取消之前所做的映射。
```

为了放宽与部件或端口所有连接和映射相关的 **disconnect** 和 **unmap** 操作，允许使用只带一个变元的 **disconnect** 和 **unmap** 操作。这个唯一的变元用于说明要断开连接或取消映射的连接方。关键字 **all port** 可用于表示一个部件的所有端口。

例 2：

```
  :
disconnect(MyNewComponent:Port1);          // 断开部件MyNewComponent拥有的、Port1的所有连接。
unmap(MyNewComponent:all port);            // 取消部件MyNewComponent的所有映射。
  :
```

不带任何参数的 **disconnect** 或 **unmap** 操作用法是使用带参数 **self:all port** 的操作的简写形式。它断开调用操作之部件的所有端口的连接，或者取消调用操作之部件的所有端口的映射。

例 3：

```
  :
disconnect;                                // 是……的所写形式。
disconnect(self:all port);                 // 断开调用操作的部件的所有端口连接。
  :
unmap;                                     // 是……的所写形式。
unmap(self:all port);                      // 取消调用操作的部件的所有端口映射。
  :
```

关键字 **all component** 只能与关键字 **all port** 一起使用，即 **all component:all port**，并只能由 MTC 来使用。此外，**all component:all port** 变元将当作 **disconnect** 或 **unmap** 操作的唯一变元来使用，并且它允许释放测试配置的所有连接和映射。

例 4：

```
  :
  :
disconnect(all component:all port);        // MTC断开测试配置中部件的所有端口连接。
  :
  :
unmap(all component:all port);             // MTC取消测试配置中部件的所有端口映射。
  :
```

22.4 MTC、System和Self 操作

部件引用（见第 8.7 节）有三个操作 — **mtc** 和 **system**，它们分别返回对主测试部件和测试系统接口的引用。**self** 操作可用于返回调用它的部件引用。

例如：

```
var MyComponentType MyAddress;
MyAddress := self; // 存储当前的部件引用。
```

对部件引用仅允许赋值、等于和不等于操作。

22.5 启动测试部件操作

一旦创建和连接了一个 PTC，那么行为必须绑定于该 PTC，并必须开始执行其行为。这通过使用 **start** 操作来完成（原因是，创建 PTC 并不启动部件行为的执行）。区分 **create** 和 **start** 操作的原因是为了允许在实际运行测试部件之前能够完成连接操作。

start 操作将把所需的行为绑定于测试部件。通过对已定义函数的引用来定义该行为。

一个活动类型的 PTC 可以按顺序执行若干行为函数。在一个非活动的 PTC 上启动第二个行为函数或在一个仍在运行的 PTC 上启动一个函数将导致一个测试用例错误。如果在前一个函数终止后在一个活动类型的 PTC 上启动一个函数，那么它使用变量值、定时器、端口和局部判定，原因是，在前一个函数终止后，它们被留下了。尤其是，如果在前一个函数中启动一个定时器，那么应使后续的函数能够处理可能的超时事件。

例如：

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }
 :
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
 :
MyNewPTC := MyComponentType.create;       // 创建一个新的、非活动的测试部件。
MyAlivePTC := MyComponentType.create alive;// 创建一个新的、活动类型的测试部件。
 :
MyNewPTC.start(MyFirstBehaviour());       // 启动一个非活动的部件。
MyNewPTC.done;                            // 等待终止。
MyNewPTC.start(MySecondBehaviour());        // 测试案例错误。
 :
MyAlivePTC.start(MyFirstBehaviour());     // 启动活动类型的部件。
MyAlivePTC.done;                            // 等待终止。
MyAlivePTC.start(MySecondBehaviour());    // 启动同一部件上的下一个函数。
 :
```

对在一个 **start** 测试部件操作中调用的函数应用以下限制：

- 如果这个函数有参数，那么这些参数只能是**in**参数，即带值的参数。
- 这个函数应有一个**runs on**子句定义来引用与新建部件兼容的部件类型（见第6.7.3节）。
- 不得将端口和定时器传入该函数。

注 — 当创建部件时，由于**in**和**inout**端口启动监听，这时它们开始执行，因此在这些端口的输入队列中可能已经有消息在等待处理。

22.6 停止测试行为操作

通过使用 **stop** 测试部件语句，一个测试部件可以停止执行它自身当前运行的测试行为或者停止执行运行于另一个测试部件上的测试行为。如果一个部件不停止其自身行为，而是停止在测试系统中另一个测试部件上运行的行为，那么必须使用其部件引用来确定要停止的部件。一个部件可以通过使用一个简单的 **stop** 执行语句来停止其自身的行为（见第 19.10 节），或者通过在 **stop** 操作中寻址其自身来停止其自身的行为，如通过使用 **self** 操作。

例 1：

注 1 — **create**、**start**、**running**、**done** 和 **killed**操作只能用于PTC，**stop**操作还可以用于MTC。

```
var MyComponentType MyComp := MyComponentType.create; // 创建一个新的测试部件。
MyComp.start(CompBehaviour());                        // 启动这个新的部件。
 :
if (date == "1.1.2005") {
  MyComp.stop;          // 停止部件“MyComp”。
}

 :
if (a < b ) {
   :
  self.stop;       // 当前正在执行的测试部件停止它自身行为。
}
 :
  stop             // 测试部件停止它自身行为。
```

停止测试部件是终止执行当前正在运行行为的显性形式。一旦到达测试用例或函数末尾（测试用例或函数开始于该部件），通过完成其执行或通过一个显性的 **return** 语句，也可以终止一个测试部件行为。这种形式的终止也称为隐性停止。隐性停止具有显性停止的同样作用，即利用被停止测试部件的局部判定来更新全局判定（见第 25 节）。

如果停止的测试部件是 MTC，那么将释放所有当前 PTC 的资源，将从测试系统中移去 PTC，并且测试用例将终止（见第 27.2 节）。

停止一个非活动类型的测试部件（隐性地或显性地）将破坏它，并将释放与测试部件相关联的所有资源。

停止一个活动类型的测试部件将只停止当前正在运行的行为，但部件将继续存在，并可执行新的行为（在其上利用 **start** 操作启动的行为）。在停止其行为后，部件将继续保持在一致的状态上。

注 2 — 例如，如果在向一个已经绑定的变量赋予新值期间停止一个活动类型部件的行为，那么变量将在部件停止后继续保持绑定状态（以旧的或新的值）。同样，如果在重新启动一个已经运行的定时器期间停止部件，那么定时器在行为终止后继续保持运行状态。

终止测试用例的规则以及最终测试判定的计算在第 25 节中进行描述。

关键字 **all** 只能由 MTC 使用，以便停止所有正在运行的 PTC，MTC 自身除外。

注 3 — 一个PTC可以通过停止MTC来停止测试用例的执行。

例 2：

```
  :
all component.stop      // MTC停止测试用例所有的PTC，但不停止它自身。
  :
```

注 4 — 停止PTC的具体机制超出了本建议书的讨论范围。

22.7 运行操作

running 操作允许在一个测试部件上执行行为，以确定在一个不同的测试部件上运行的行为是否已经完成。**running** 操作只能用于 PTC。对已启动但尚未终止或停止的 PTC，**running** 操作返回 **true**，否则返回 **false**。**running** 操作被认为是一个布尔表达式，因此它返回一个 **boolean** 值来指出特定的测试部件（或所有的测试部件）是否已终止。与 **done** 操作相比，**running** 操作可以自由地用在 **boolean** 表达式中。

当关键字 **all** 与 **running** 操作一起使用时，如果所有已启动且尚未被另一个部件显性停止的 PTC 都在执行其行为，那么返回 **true**，否则返回 **false**。

当关键字 **any** 与 **running** 操作一起使用时，如果至少有一个 PTC 在执行其行为，那么返回 **true**，否则返回 **false**。

例如：

```
if (PTC1.running)              // 在一个if语句中running的用法
{
  // 执行……
}

while (all component.running != true) { // 在一个循环条件中running的用法
  MySpecialFunction()
}
```

22.8 Done操作

done 允许在一个测试部件上执行行为，以确定在一个不同的测试部件上运行的行为是否已完成，**done** 操作只能用于 PTC。

done 操作的使用方式应与接收操作或 **timeout** 操作相同，这意味着它不得用在 **boolean** 表达式中，但它可用于决定 **alt** 语句中的一个选择对象或在行为描述中作为一个独立的语句。在后一种情况中，**done** 操作看作是只有一个选择对象的 **alt** 语句的简写形式，也就是说，它具有阻塞语义，因而提供了被动等待测试部件终止的能力。

当 **done** 操作用于 PTC 时，当且仅当该 PTC 的行为已被停止时（隐性地或显性地）或者 PTC 已被取消时，它匹配，否则匹配不成功。

当关键字 **all** 与 **done** 操作一起使用时，如果没有任何 PTC 在执行其行为，那么它匹配。如果没有创建任何 PTC，那么它也匹配。

当关键字 **any** 与 **done** 操作一起使用时，如果至少有一个 PTC 的行为已被停止或取消，那么它匹配，否则匹配不成功。

注 — 停止一个非活动部件的行为也将导致从测试系统中移去该部件，而停止一个活动类型的部件将保持部件在测试系统中的活动状态。在两种情况下，**done**操作都匹配。

例如：

```
// 选择对象中done操作的使用
:
alt {
   [] MyPTC.done {
       setverdict(pass)
      }

   [] any port.receive {
       repeat
      }
}
:

var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
   // 一旦函数MyPTCBehaviour（或它调用的函数/可选步骤）停止，即匹配。
c.done;
   // 函数MyPTCBehaviour（或它调用的函数/可选步骤）结束也匹配。
if(c.running) {c.done}
   // 此处的done只匹配下一个行为结束。

// 下面的done操作作为独立的语句：
 :
all component.done;
 :

//有如下含义：
 :
alt {
   [] all component.done {}
}
 :

// 因此，在所有并行测试部件终止之前，块将一直执行。
```

22.9 Kill测试部件操作

用于测试部件的 **kill** 操作用于停止执行该部件当前正在运行的行为 — 如果有的话，并释放所有与之相关联的资源（包括被取消部件的所有端口连接），以及从测试系统中移去部件。通过一个简单的 **kill** 声明，或者通过结合使用 **self** 操作和 **kill** 操作来寻址其自身，**kill** 操作可以用在当前测试部件自身上。**kill** 操作也可用于另一个测试部件。在这种情况下，要被取消的部件将利用其部件引用来寻址。如果 **kill** 操作用在 MTC 上，即 **mtc.kill**，那么它终止测试用例。

例 1：

```
var PTCType MyAliveComp := PTCType.create alive;     // 创建一个活动类型的测试用例。
MyAliveComp.start(MyFirstBehavior());                // 启动新的部件。
MyAliveComp.done;                                    // 等待终止。
MyAliveComp.start(MySecondBehavior());               // 启动部件一个2nd时间。
MyAliveComp.done;                                    // 等待终止。
MyAliveComp.kill;                                    // 释放其资源。
```

关键字 **all** 只能由 MTC 使用，以便停止和取消所有正在运行的 PTC，但不是 MTC 自身。

例 2：

```
all component.kill;     // MTC首先停止测试用例的所有（活动类型的和普通的）PTC，并释放其资源。
```

22.10 Alive操作

alive 操作是一个布尔操作，它检查测试部件是否已被创建，并检查是否已做好执行准备或者已经开始执行一个行为函数。当用在一个普通测试部件上时，如果部件是非活动的或者正在运行一个函数，那么 **alive** 操作返回 true，否则返回 false。当用在一个活动类型的测试部件上时，如果部件是非活动的、正在运行或已被停止，那么 **alive** 操作返回 true；如果部件已被取消，那么 **alive** 操作返回 false。

类似 **running** 操作，**alive** 操作只能用在 PTC 上（见第 22.7 节）。尤其是，当与关键字 **all** 结合使用时，如果所有（活动类型的或普通的）PTC 都是活动的，那么它返回 true。

alive 操作当与关键字 **any** 结合使用时，如果至少有一个 PTC 是活动的，那么它返回 true。

例如：

```
  :
PTC1.done;                              // 等待部件的终止。
if (PTC1.alive) {                       // 如果部件依然是活动的……
  PTC1.start(AnotherFunction());        // ……执行其上另一个函数。
}
```

22.11 Killed操作

killed 操作允许确定一个不同的测试部件是否是活动的，或者是否已从测试系统中移去。

killed 操作的使用方式同接收操作的使用方式。这意味着它不得用在 **boolean** 表达式中，当可用于确定 **alt** 语句中的一个选择对象，或者作为行为描述中的一个单独语句。在后一种情况中，**done** 操作被认为是只有一个选择对象的 **alt** 语句的简写形式，即它有阻塞语义，因而提供了被动等待测试部件终止的能力。

注 — 当检查普通测试部件时，如果它停止（隐性地或显性地）执行其行为或者已被显性地**killed**，那么**killed**操作匹配，即该操作等同于**done**操作（见第22.8节）。不过，当检查活动类型的测试部件时，当且仅当已用**kill**操作取消部件时，**killed**操作才匹配。否则 **killed**操作不成功。

killed 操作只能用于 PTC。

当关键字 **all** 与 **killed** 操作一起使用时，如果测试用例的所有 PTC 都已停止存在，那么它匹配。如果尚未创建任何 PTC，那么它也匹配。

当关键字 **any** 与 **killed** 操作一起使用时，如果至少有一个 PTC 停止存在，那么它匹配，否则匹配不成功。

例如：

```
var MyPTCType ptc := MyPTCType.create alive;    // 创建一个活动类型的测试部件。
timer T(10.0);                                  // 创建一个定时器。
T.start;                                        // 启动定时器。
ptc.start(MyTestBehavior());                    // 开始执行PTC上的一个函数。
alt {
[] ptc.killed {                                 // 如果在执行期间取消了PTC……
  T.stop;                                       // ……停止定时器并且……
  setverdict(inconc);                           // ……判定设为“inconclusive”。
   }
[] ptc.done {                                   // 如果PTC有规律地终止……
  T.stop;                                       // ……停止定时器并且……
  ptc.start(AnotherFunction());                 // 启动PTC上的另一个函数。
   }
[] T.timeout {                                  // 如果在PTC停止之前发生超时。
  ptc.kill;                                     // ……取消PTC并且……
  setverdict(fail);                             // ……判定设为“fail”。
   }
}
```

22.12 使用部件数组

create、**connect**、**start**、**stop** 和 **kill** 操作不直接工作于部件数组上。相反，应作为这些操作的参数来提供数组的某个特定元素。对部件而言，通过使用一个部件引用的数组，并将相关数组元素赋给 **create** 操作的结果，来实现数组的作用。

例如：

```
// 这个例子表示了如何通过使用循环、在一个部件引用数组中存储所创建的部件引用，
// 来为创建、连接和运行部件数组的结果建模。

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
    :
  var integer i;
  var MyPTCType1  MyPtc[11];
   :
  for (i:= 0; i<=10; i:=i+1)
  {
     MyPtc[i] := MyPTCType1.create;
     connect(self:PtcCoordination, MyPtc[i]:MtcCoordination);
     MyPtc[i].start(MyPtcBehaviour());
  }
   :
}
```

22.13 带部件的any和all用法概述

关键字 **any** 和 **all** 可以和表 16 中给出的配置操作一起使用。

表 16/Z.140－带部件的any 和 all

操　作	允 许 的		例　子	注　释
	any （见注释）	**all** （见注释）		
create				
start				
running	是，但仅来自于 MTC	是，但仅来自于 MTC	**any component.running;** **all component.running;**	是否存在执行测试行为的 PTC? 是否所有的 PTC 都执行测试行为?
alive	是，但仅来自于 MTC	是，但仅来自于 MTC	**any component.alive;** **all component.alive;**	是否存在活动的 PTC? 是否所有的 PTC 都是活动的?
done	是，但仅来自于 MTC	是，但仅来自于 MTC	**any component.done;** **all component.done;**	是否存在已完成执行的 PTC? 是否所有的 PTC 都完成了其执行?
killed	是，但仅来自于 MTC	是，但仅来自于 MTC	**any component.killed;** **all component.killed;**	是否存在停止了存在的 PTC? 是否所有的 PTC 都停止了存在?
stop		是，但仅来自于 MTC	**all component.stop;**	停止所有 PTC 上的行为。
kill		是，但仅来自于 MTC	**all component.kill;**	取消所有的 PTC，即它们停止了存在。
注 — **any** 和 **all** 只指 PTC，即不考虑 MTC。				

23 通信操作

23.0 概述

TTCN-3 支持基于消息的和基于过程的通信。此外，TTCN-3 允许检查输入端口队列的顶层元素，并利用控制操作的方式来控制对端口的访问。

表 17/Z.140－TTCN-3通信操作综述

通 信 操 作			
通信操作	**关键字**	**可用于基于消息的端口上**	**可用于基于过程的端口上**
基于消息的通信			
发送消息	**send**	是	
接收消息	**receive**	是	
消息触发	**trigger**	是	
基于过程的消息			
调用过程调用	**call**		是
接受来自远程实体的过程调用	**getcall**		是
回答来自远程实体的过程调用	**reply**		是
（对一个已接受的调用）引起异常	**raise**		是
处理来自之前调用的响应	**getreply**		是
捕获异常（从被调用的实体）	**catch**		是
检查输入端口队列的顶层元素			
检查接收到的消息/调用/异常/应答	**check**	是	是
控制操作			
清除端口队列	**clear**	是	是
清理队列，并且使得能够在端口上进行发送和接收。	**start**	是	是
使得不能发送，并且不允许在端口上匹配接收操作。	**stop**	是	是
使得不能发送，并且不允许接收操作匹配新的消息/调用。	**halt**	是	是

23.1 通信操作的通用格式

23.1.0 概述

像 **send** 和 **call** 这样的操作，用于测试部件之间以及 SUT 与测试部件之间的交换信息。为了解释这些操作的通用格式，可以将它们分成两组：

a） 一个测试部件发送一个消息（**send**操作）、调用一个过程（**call**操作）、应答一个已接受的调用（**reply**操作）或引起一个异常（**raise**操作）。这些动作合起来称为发送操作；

b） 一个部件接受一个消息（**receive**操作）、等待一个消息（**trigger**操作）、接受一个过程调用（**getcall**操作）、接受一个之前被调用过程的应答（**getreply**操作）或捕获一个异常（**catch**操作）。这些动作合起来称为接收操作。

23.1.1 发送操作的通用格式

发送操作由一个发送部分组成，在一个阻塞的、基于过程的 **call** 操作情况下，由一个响应部分和异常处理部分组成。

发送部分：

- 指定将发生指定操作的端口；
- 定义要传输的消息或过程调用；
- 给定一个（可选的）地址部分，它唯一地确定一个或多个通信伙伴，消息、调用、应答或异常将发送给这些通信伙伴。

端口名、操作名和值将出现在所有的发送操作中。地址部分（用关键字 **to** 表示）是可选的，且只在一对多连接的情况下才需要进行说明，当中：

- 使用单播通信，并将显性确定一个接收实体；
- 使用多播通信，并需要显性确定一系列接收实体；
- 使用广播通信，并需要对所有连接至规定端口的实体进行寻址。

注 — 术语“单播”、“多播”和“广播”通信用于指端口通信。这意味着只可能对一个、若干或所有连接至规定端口的测试部件进行寻址。单播、多播、广播也可以用于映射的端口。在这种情况下，可以通过规定的映射端口到达SUT内的一个、若干或所有实体。

例 1：

发送部分			**（可选的）响应和异常处理部分**
端口和操作	**值部分**	**（可选的）地址部分**	
MyP1.**send**	(MyVariable + YourVariable - 2)	**to** MyPartner;	

仅在基于过程的通信的情况下，才需要做出响应并进行异常处理。**call** 操作的响应和异常处理部分是可选的，它们只有在以下情况下才是需要的：即被调用的过程返回一个值或者拥有 **out** 或 **inout** 参数，在调用部件中需要这些参数的值，以及被调用的过程可能引起需要由调用部件来处理的异常。

调用操作的响应和异常处理部分利用 **getreply** 和 **catch** 操作来提供要求的功能性。

例 2：

发送部分			**（可选的）响应和异常处理部分**
端口和操作	**值部分**	**（可选的）地址部分**	

MyP1.**call**	(MyProc:{MyVar1})		{
			[] MyP1.**getreply**(MyProc:{MyVar2}) {}
			[] MyP1.**catch**(MyProc, ExceptionOne) {}
			}

23.1.2 接收操作的通用格式

接收操作由一个接收部分和一个（可选的）赋值部分组成。

接收部分：

a） 指定将发生操作的端口；

b） 定义一个匹配部分，它指定将匹配该语句的可接受的输入；

c） 给定一个（可选的）地址表达式，它唯一地确定通信伙伴（在一对多连接的情况下）。

所有接收操作的端口名、操作名和值部分都应出现。确定通信伙伴（用关键字 **from** 表示）是可选的，且只在一对多连接的情况下才需要进行说明，当中需要显性地确定接收实体。

接收操作中的赋值部分是可选的。对基于消息的端口，当需要存储接收到的消息时，将用到赋值部分。在基于过程的端口的情况下，它用于存储一个已接受调用的 **in** 和 **inout** 参数，用于存储返回值，或者用于存储异常。对赋值部分，需要强类型机制，例如，用于存储消息的变量类型应与输入消息的类型相同。

另外，赋值部分也可以用于把一个消息、异常、**reply** 或 **call** 的 **sender** 地址赋给一个变量。这对一对多连接是有用的，例如，可以接收来自不同部件的相同消息或调用，但该消息、异常、应答或调用必须发送回最初的发送部件。

例如：

接收部分				（可选的）赋值部分		
端口和操作	匹配部分	（可选的）地址表达式		（可选的）值赋值	（可选的）参数值赋值	（可选的）发送方值赋值
MyP1.getreply	(AProc:{?} value 5)		->		param (V1)	sender APeer

接收部分				（可选的）赋值部分		
端口和操作	匹配部分	（可选的）地址表达式		（可选的）值赋值	（可选的）参数值赋值	（可选的）发送方值赋值
MyP2.receive	(MyTemplate(5,7))	from APeer	->	value MyVar		

23.2 基于消息的通信

23.2.0 概述

基于消息的通信是基于异步消息交换的通信。基于消息的通信在 **send** 操作上是非阻塞的，如图 11 所示，当中发送方在 **send** 操作发生之后立即继续进行它的处理过程。接收方在 **receive** 操作上被阻塞，直到它对接收到的消息进行处理。

除了 **receive** 操作外，TTCN-3 还提供了一个 **trigger** 操作，利用一定的匹配准则来过滤来自给定输入端口上接收信息流的消息。从端口移去队列顶层不满足匹配准则的消息，而不采取进一步行动。

图 11/Z.140－异步发送和接收的图示

23.2.1 发送操作

23.2.1.0 概述

send 操作用于将一个消息置于一个输出消息端口上。可以通过引用一个已定义的模板来规定该消息，或者可定义为一个内置的模板。如果要发送的消息的类型存在模糊性，那么在内置的消息时，将使用可选的类型部分。

send 操作将只用在基于消息的（或混合型的）端口上，并且要发送的模板的类型将位于该端口类型定义的输出类型列表中。

例如：

```
MyPort.send(MyTemplate(5,MyVar));        // 通过MyPort发送带实参5和MyVar的模板MyTemplate。

MyPort.send(5);                          // 发送整数值5（它是一个内置的模板）。
```

23.2.1.1 单播、多播或广播发送

TTCN-3 支持单播、多播和广播通信。可以由 **send** 操作中可选的 **to** 子句来确定所用的通信机制。在一对一连接的情况下，可以省略 **to** 子句，当中使用单播通信，并由测试系统结构来唯一地确定消息接收者。在一对一连接的情况下，**to** 子句将出现。

如果 **to** 子句只访问一个通信伙伴，那么规定单播通信。如果 **to** 子句包括一个通信伙伴列表，那么使用多播通信。通过使用带关键字 **all component** 的 **to** 子句来定义广播。

例如：

```
MyPort.send(charstring:"My string") to MyPartner;
                        // 发送串“My string”给部件，部件引用保存在变量MyPartner中。

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
                    // 发送算术表达式结果给MyPartner。

MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                    // 说明一个多播通信，当中MyTemplate的值发送给
                    // 保存在变量MyPeerOne和MyPeerTwo中的两个部件引用。

MyPCO3.send(MyTemplate) to all component;
                    // 广播通信：MyTemplate的值发送给
                    // 可通过该端口寻址的所有部件。
                    // 如果MyPCO3为映射的端口，那么各部件可驻留在SUT内。
```

23.2.2 接收操作

23.2.2.0 概述

receive 操作用于从一个输入消息端口队列接收一个消息。可以通过引用一个已定义的模板来指定该消息，或者可将之定义为一个内置的模板。当内嵌地定义消息时，当要接收的消息的类型是含糊的时，可选的类型部分将出现。**receive** 操作将只能用在基于消息的（或混合型的）端口上，且要接收的值的类型将被包括在该端口类型定义的输入类型列表中。

当且仅当输入端口队列的顶层消息满足所有 **receive** 操作相关的匹配准则时，**receive** 操作才从相关的输入端口队列中移去顶层消息。不会出现任何将输入值绑定于表达式术语或模板的情况。

如果匹配不成功，那么输入端口队列的顶层消息不得从端口队列中移去，也就是说，如果 **receive** 操作当作 **alt** 语句的一个选择对象来使用，且它不成功的话，那么测试用例的执行将从 **alt** 语句的下一个选择对象开始继续进行。

匹配准则与要接收消息的类型和值有关。要接收消息的类型和值由 **receive** 操作的变元来确定，也就是说，可以源自定义的模板，也可以内置地规定。将使用 **receive** 操作匹配准则中的可选类型字段，以避免要接收值的类型的任何模糊性。

注 1 — 通过防止解码器从接收到的、以不同于属性规定之方式进行编码的消息产生一个抽象值，编码属性也以一种隐性的方式来参加匹配。

在一对多连接的情况下，可以限制 **receive** 操作到一个特定的通信伙伴，使用关键字 **from** 来表示这种限制。

例 1：

```
MyPort.receive(MyTemplate(5, MyVar));   // 在端口MyPort上匹配一个满足
                                        // 由模板MyTemplate定义之条件的消息。

MyPort.receive(A<B);                    // 匹配一个依赖于A<B结果的布尔值。

MyPort.receive(integer:MyVar);          // 在端口MyPort上匹配一个整数值与MyVar的值。

MyPort.receive(MyVar);                  // 为前一个例子的一个替换。

MyPort.receive(charstring:"Hello")from MyPeer; // 匹配来自MyPeer的字符串“Hello”。
```

如果匹配成功，那么从端口队列中移去的值可以存储在一个变量中，该变量可以在一个变量中恢复和保存。这用符号“->”和关键字 **value** 来表示。

也有可能恢复和保存消息发送者的部件引用或地址。这用关键字 **sender** 来表示。

注 2 — 当在一个连接的端口上接收消息时，只有部件引用保存在以下关键字**sender**中，但测试系统也将内部保存部件名称，如果有的话（将用在日记记录中）。

例 2：

```
MyPort.receive(MyType:?) -> value MyVar; // 将接收到的消息的值赋给MyVar。

MyPort.receive(A<B) -> sender MyPeer;    // 将发送者的地址赋给MyPeer。

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// 将接收到的消息的值保存在MyVar中，将发送者的地址保存在MyPeer中。
```

23.2.2.1 接收任何消息

不带任何有关待接收消息类型和值匹配准则变元列表的 **receive** 操作，如果满足所有其它匹配准则，那么移去输入端口队列顶层的消息（如果有的话）。

通过 *ReceiveAnyMessage* 接收到的消息不得赋值给一个变量。

例如：

```
MyPort.receive;                          // 从MyPort移去队列顶层值。

MyPort.receive from MyPeer;              // 如果发送者是MyPeer的话，那么从MyPort移去队列顶层值。

MyPort.receive -> sender MySenderVar;    // 从MyPort移去队列顶层消息，并将发送者地址赋给MySenderVar。
```

23.2.2.2 在任何端口上接收

为了在任何端口上 **receive** 消息，应使用关键字 **any port**。

例如：

```
any port.receive(MyMessage);
```

23.2.3 触发操作

23.2.3.0 概述

trigger 操作从相关的输入端口队列中移去顶层消息。如果该顶层消息满足匹配准则，那么 **trigger** 操作的行为方式同 **receive** 操作。如果该顶层消息不满足匹配准则，那么 **trigger** 操作将从输入端口队列中被移去，而不采取任何进一步的动作。**trigger** 操作将只能用在基于消息的（或混合型的）端口上，且要接收的值的类型将被包括在端口类型定义的输入类型列表中。

注 — 第22.2.2.0节中的注1也适用于**trigger**操作。

在一个行为描述中，可以将 **trigger** 操作用作一个独立的语句。在后一种情况下，**trigger** 操作可以被看作是只有一个选择对象的 **alt** 语句的简写形式，也就是说，它有阻塞语义，并因此提供等待下一个消息匹配规定之模板或该队列上值的能力。

例 1：

```
MyPort.trigger(MyType:?);
// 说明在端口MyPort上，对观测到的、第一个带任意值的MyType类型的消息的接收将触发该操作。
```

trigger 操作需要端口名、有关类型和值的匹配准则、一个可选的 **from** 限制（即通信伙伴的选择）以及一个可选的赋值（将匹配消息和发送者部件赋给变量）。

例 2：

```
MyPort.trigger(MyType:?) from MyPartner;
// 在端口MyPort上，在接收来自MyPartner的、第一个MyType类型的消息时触发。

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// 这个例子与上一个例子几乎是相同的。另外，触发的消息（即满足所有匹配准则的消息）存储在变量MyRecMessage中。

MyPort.trigger(MyType:?) -> sender MyPartner;
// 这个例子与第一个例子几乎是相同的。另外，将恢复发送者部件的引用，并存储在变量MyPartner中。
```

```
MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// 在接收一个之后存储在变量MyVar中的任意整数值时触发。发送者部件的引用将存储在变量MyPartner中。
```

23.2.3.1 在任何消息上的触发

不带任何变元列表的 **trigger** 操作将在收到任何消息后触发。因此，其含义与接收任何消息的含义是相同的。由 *TriggerOnAnyMessag* 接收的消息不得赋给一个变量。

例如：

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 在任何端口上的触发

为了在任何端口上 **trigger** 消息，应使用关键字 **any port**。

例如：

```
any port.trigger
```

23.3 基于过程的通信

23.3.0 概述

基于过程的通信的原理是调用远程实体中的过程。TTCN-3 支持阻塞的和非阻塞的、基于过程的通信。阻塞的、基于过程的通信在调用方和被调用方是阻塞的，而非阻塞的、基于过程的通信仅在被调用方是阻塞的。将根据第 13 节中的规则来描述用于非阻塞的、基于过程的通信的过程特征。

阻塞的、基于过程的通信的通信方案如图 12 所示。调用方通过使用 **call** 操作来调用被调用方中的一个远程过程，被调用方通过 **getcall** 操作的方式来接受该调用，并通过使用 **reply** 操作应答调用或引起（**raise** 操作）一个异常的方式来响应调用。调用方通过使用 **getreply**（获得应答）或 **catch**（捕获）操作来处理应答或异常。在图 12 中，用虚线来表示调用方和被调用方的阻塞。

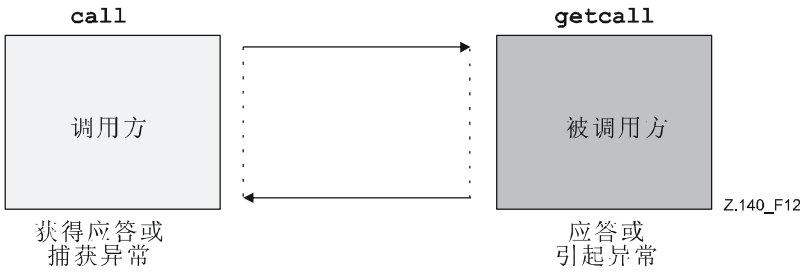


图 12/Z.140－阻塞的基于过程的通信图示

非阻塞的、基于过程的通信的通信方案如图 13 所示。调用方通过使用 **call** 操作来调用被调用方中的一个远程过程，并继续其自身的执行，即不等待一个应答或异常。被调用方通过 **getcall** 操作方式来接受该调用，并执行被请求的过程。如果执行不成功，那么被调用方可以引起一个异常，来告知调用方。调用方可以通过使用 **alt** 语句中的 **catch** 操作来处理异常。在图 13 中，用虚线来表示被调用方的阻塞（直到调用处理结束并可能引起一个异常）。

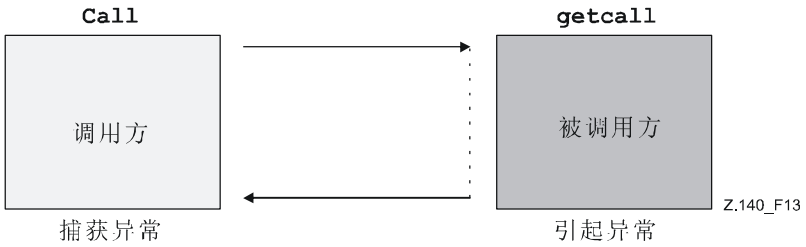


图 13/Z.140－非阻塞的基于过程的通信图示

23.3.1 调用操作

23.3.1.0 概述

call 操作用于说明一个测试部件调用 SUT 或另一个测试部件中的过程。**call** 操作将只用在基于过程的（或混合型的）端口上。发生调用操作的端口的类型定义将在其 **out** 或 **inout** 列表中包括过程名，也就是说，必须允许它在该端口上调用这个过程。

在 **call** 操作发送部分中传输的信息是一个可以在特征模板格式中定义的特征，也可以内嵌地进行定义。该特征的所有 **in** 和 **inout** 参数将有一个特定的值，也就是说，不允许使用匹配机制，如 *AnyValue*。

不使用 **call** 操作的特征变元来为 **out** 或 **inout** 参数恢复变量名。过程返回值以及 **out** 和 **inout** 参数值对变量的实际赋值将在 **call** 操作的响应和异常处理部分中，通过 **getreply** 和 **catch** 操作方式来显性地进行。这允许 **call** 操作中特征模板的使用（与模板的使用方式相同）可以用于多种类型。

例 1：

```
// 假定……
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// MyProc的一个调用
MyPort.call(MyProc:{ -, MyVar2}) {      // 用于调用MyProc的内置特征模板。
   [] MyPort.getreply(MyProc:{?, ?}) { }
}

//……以及MyProc的另一个调用。
MyPort.call(MyProcTemplate) {           // 为调用MyProc而使用的特征模板。
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

在一对多连接的情况下，应唯一指定通信伙伴，这用关键字 **to** 来表示。

例 2：

```
MyPort.call(MyProcTemplate) to MyPeer {     // 在MyPeer处调用MyProc
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

23.3.1.1 处理一个调用的响应和异常

在非阻塞的、基于过程的通信的情况下（见第 23.3.1.4 节），通过将 **catch** 操作（见第 23.3.6 节）用作 **alt** 语句中的选择对象，来完成对 **call** 操作异常的处理。

如果使用 **nowait** 选项的话（见第 23.3.1.2 节），那么通过将 **getreply** 操作（见第 23.3.4 节）和 **catch** 操作（见第 23.3.6 节）用作 **alt** 语句中的选择对象，来完成对 **call** 操作响应或异常的处理。

在阻塞的、基于过程的通信的情况下，在 **call** 操作的响应和异常处理部分中，通过 **getreply** 操作（见第 23.3.4 节）和 **catch** 操作（见第 23.3.6 节），来完成对 **call** 操作响应或异常的处理。

call 操作的响应和异常处理部分看起来与一个 **alt** 语句体相似，它定义了一个选择对象集合，用于描述对该调用可能的响应和异常。对选择对象的选择仅仅基于用于被调用过程的 **getreply** 和 **catch** 操作。无限制的 **getreply** 和 **catch** 操作只能处理来自被调用过程的应答以及由被调用过程引起的异常。不允许使用 **else** 分支和调用可选步骤。

如果必要的话，可以通过置于选择对象方括号“[]”之间的一个 **boolean** 表达式来激活/去激活一个选择对象。

call 操作的响应和异常处理部分的执行象一个没有任何活动的缺省的 **alt** 语句。这意味着一个对应的快照包括计算（可选的）布尔防卫所需的所有信息，可以包括端口的顶层元素（如果有的话），过程在该端口上被调用，也可以包括一个超时例外，它由监视该调用的（可选的）定时器产生（见第 23.3.1.2 节）。

在响应和异常处理部分中对防卫选择对象的布尔表达式的求值可能有副作用。为了避免不期望出现的副作用，将使用与 **alt** 语句中一样的布尔防卫规则（见第 20.1.1 节）。

例如：

```
// 假定
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
  exception (ExceptionTypeOne, ExceptionTypeTwo);
 :

// 调用MyProc3。
MyPort.call(MyProc3:{ -, true }) to MyPartner {

  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

  [] MyPort.catch(MyProc3, MyExceptionOne) {
      setverdict(fail);
      stop;
     }
  [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
      setverdict(inconc);
     }
  [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

23.3.1.2 处理调用的超时异常

call 操作可以选择是否包括一个超时，它定义为一个明确的值或 **float** 类型的一个常量，并在启动 **call** 操作后定义时间的长度。**timeout** 异常将由测试系统产生。如果在 **call** 操作中没有出现任何超时值部分，那么不会产生任何 **timeout** 异常。

例 1：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

  [] MyPort.getreply(MyProc:{?, ?}) { }

  [] MyPort.catch(timeout) {            // 20ms后超时异常。
      setverdict(fail);
      stop;
     }
}
```

在 **call** 操作中使用关键字 **nowait** 而不是一个超时异常值，允许不用等待响应或被调用过程引起的异常或超时异常就可继续调用一个过程。

例 2：

```
MyPort.call(MyProc:{5, MyVar}, nowait);  // 调用测试部件将继续其执行，而不等待MyProc的终止。
```

如果使用关键字 **nowait**，那么必须通过在随后的 **alt** 语句使用一个 **getreply** 或 **catch** 操作来从端口队列中移去被调用过程的一个可能的响应或异常。

23.3.1.3 调用不带返回值、输出参数、输入/输出参数和异常的阻塞过程

阻塞的过程可以没有任何返回值，没有任何 **out** 和 **inout** 参数，也可以不引起任何异常。用于这样过程实例的调用操作也会有一个响应和异常处理部分，以统一的方式来处理阻塞。

例如：

```
// 假定
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// MyBlockingProc的一个调用
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

23.3.1.4 调用非阻塞过程

非阻塞的过程没有任何 **out** 和 **inout** 参数，没有任何返回值，通过关键字 **noblock** 在对应的特征定义中指明非阻塞的特性。

有关非阻塞的过程的 **call** 操作应没有任何响应和异常处理部分，不会引起任何超时异常，也不使用关键字 **nowait**。

非阻塞的过程可能引起的异常必须通过在随后的 **alt** 或 **interleave** 语句使用 **catch** 操作来从端口队列中移去。

23.3.1.5 过程的单播、多播和广播调用

如同 **send** 操作，TTCN-3 也支持对过程的单播、多播和广播调用。这可以用第 23.2.1.1 节中所述的相同方式来实现；也就是说，对单播调用，**call** 操作的 **to** 子句的变元是一个接收实体的地址（或者在一对一连接的情况下可以省略）；对多播调用，**call** 操作的 **to** 子句的变元是接收方集合的一个地址列表；对广播调用，**call** 操作的 **to** 子句的变元是关键字 **all component**。在一对一连接的情况下，可以省略 **to** 子句，原因是由系统结构来唯一地确定接收实体。

已在第 23.3.1.1 节和第 23.3.1.4 节中对阻塞或非阻塞单播 **call** 操作的响应和异常处理做了解释。一个多播或广播的 **call** 操作可以从不同的通信伙伴引起若干个响应和异常。

在非阻塞过程的多播或广播 **call** 操作的情况下，可以从不同通信伙伴引起的所有异常都可以在后续的 **catch**、**alt** 或 **interleave** 语句中进行处理。

在阻塞过程的多播或广播 **call** 操作的情况下，存在两种选择：第一种选择是，在 **call** 操作的响应和异常处理部分，只处理一个响应或异常，然后在后续的 **alt** 或 **interleave** 语句中对更多的响应和异常进行处理。第二种选择是，通过使用重复语句，在 **call** 操作的响应和异常处理部分的一个或多个语句块和声明中，处理若干个响应或异常：重复语句的执行将引起对调用体的重新计算。

注 — 在第二种情况中，用户需要处理重复次数。

例 1：

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to (MyPeerOne, MyPeerTwo) { // MyProc的多播调用。
   // 处理来自MyPeerOne的响应。
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
       if (first) { first := false; repeat; }
       :
   }
   // 处理来自MyPeerTwo的响应。
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
       if (first) { first := false; repeat; }
       :
   }
   [] MyPort.catch(timeout) {            // 20ms后超时异常。
       setverdict(fail);
       stop;
   }
}

alt {
   [] MyPort.getreply(MyProc:{?, ?}) {     // 处理所有其它对广播调用的响应。
       repeat
   }
}
```

在阻塞过程的多播或广播 **call** 操作的情况中，使用关键字 **nowait**，需要在后续的 **alt** 或 **interleave** 语句中对所有响应和异常进行处理。

例 2：

```
MyPort.call(MyProc:{5,MyVar}) to (MyPeer1, MyPeer2) nowait; // MyProc的多播调用。

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { } // 处理MyPeer1的响应。
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { } // 处理MyPeer2的响应。
}
```

23.3.2 Getcall操作

23.3.2.0 概述

getcall 操作用于说明一个测试部件接受来自 SUT 或另一个测试部件的调用。**getcall** 操作将只能用在基于过程的（或混合型的）端口上，要被接受的过程调用特征将包括在端口类型定义的允许引入过程列表中。

当且仅当满足 **getcall** 操作相关的匹配准则时，**getcall** 操作将从输入端口队列中移去顶层调用。这些匹配准则与要处理的调用特征和通信伙伴有关，有关特征的匹配准则可以内置地予以说明，也可以源自一个特征模板。

在一对多连接的情况下，**getcall** 操作可能会限制于一个特定的通信伙伴，该限制通过使用关键字 **from** 来表示。

例 1：

```
MyPort.getcall(MyProc: MyProcTemplate(5, MyVar));   // 在MyPort处接受MyProc的一个调用。

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer; // 在MyPort处接受一个来自MyPeer的MyProc调用。
```

getcall 操作的特征变元不得用于为 **in** 和 **inout** 参数传入变量名。**in** 和 **inout** 参数值到变量的赋值应在 **getcall** 操作的赋值部分中进行。这使得 **getcall** 操作中特征模板的使用方式同模板对类型的使用方式。

getcall 操作的赋值部分（可选的）包含 **in** 和 **inout** 参数值到变量的赋值以及恢复调用部件的地址。值赋值部分不得与 **getcall** 操作一起使用。关键字 **param** 用于恢复调用的参数值。

当要求恢复发送方的地址时（例如，在一对多配置中，用于寻址一个对调用方的 **reply**，或者对调用方的异常），使用关键字 **sender**。

例 2：

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// MyProc的in或inout参数值赋给MyPar1Var和MyPar2Var。

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// 在MyPort处接受MyProc的一个带有in或inout参数5和MyVar的调用。
// 恢复调用方的地址并存储在MySenderVar中。

// 下面的getcall例子说明了使用匹配属性以及省略可能对测试说明不重要的可选部分的可能性。

MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;

MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);

MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// 第一个inout参数值是不重要的或者未被使用。

// 下面的例子解释将in或inout参数值赋给变量的可能性。假设下面的特征用于被调用的过程：

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// 将参数A、B和E赋给变量MyVarA、MyVarB和MyVarE。无需考虑out参数D。
```

```
MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// 用于将in或inout参数赋值给变量的替代记法。
// 注意：赋值列表中的名称指的是用在MyProc2特征中的名称。

MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
// 在进一步的测试用例执行中，只需要inout参数值。
```

23.3.2.1 接受任何调用

不带任何有关特征匹配准则变元列表的 **getcall** 操作，如果满足所有其它匹配准则，那么移去输入端口队列顶层的调用（如果有的话）。通过 *AcceptAnyCall* 接受的调用的参数不得赋值给一个变量。

例如：

```
MyPort.getcall;                          // 从MyPort移去顶层调用

MyPort.getcall from MyPartner;           // 从端口MyPort移去一个来自MyPartner的调用

MyPort.getcall -> sender MySenderVar;    //  从MyPort移去一个调用，并恢复调用实体的地址。
```

23.3.2.2 在任何端口上的Getcall

用关键字 **any** 表示在任何端口上的 **getcall** 操作。

例如：

```
any port.getcall(MyProc)
```

23.3.3 应答操作

reply 操作用于依据过程特征来应答一个之前接受的调用。**reply** 操作仅用在基于过程的（或混合型的）端口上。端口的类型定义应包括 **reply** 操作所属的过程名。

注 — 不能总是静态地检查一个已接受调用于一个**reply**操作之间的关系。对测试而言，允许指定一个不带相关**getcall**操作的**reply**操作。

reply 操作的值部分由一个带相关实参列表的特征引用和（可选的）返回值组成。特征可以以特征模板的格式进行定义，或者内置地定义。特征的所有 **out** 和 **inout** 参数都将拥有一个特定的值，也就是说，不允许使用像 *AnyValue* 这样的匹配机制。

对一个或多个 **call** 操作的响应可发送给连接于所寻址端口的一个、若干或所有对等实体。这可以用第 23.2.1.1 节中所述的相同方式来说明。这意味着：对单播响应，**reply** 操作的 **to** 子句的变元是一个接收实体的地址；对多播响应，**reply** 操作的 **to** 子句的变元是接收方集合的一个地址列表；对广播响应，**reply** 操作的 **to** 子句的变元是关键字 **all component**。

在一对多连接的情况下，可以省略 **to** 子句，原因是由系统结构来唯一地确定接收实体。

如果要返回一个值给调用方，那么要用关键字 **value** 显性地予以声明。

例如：

```
MyPort.reply(MyProc2:{ - ,5});           // 应答MyProc2的一个接受的调用。

MyPort.reply(MyProc2:{ - ,5}) to MyPeer; // 应答来自MyPeer的、MyProc2的一个接受的调用。

MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2); // 多播应答MyPeer1 和 MyPeer2。

MyPort.reply(MyProc2:{ - ,5}) to all component;// 广播应答所有连接至MyPort的实体。

MyPort.reply(MyProc3:{5,MyVar} value 20);// 应答MyProc3的一个接受的调用。
```

23.3.4 Getreply操作

23.3.4.0 概述

getreply 操作用于处理来自之前被调用过程的应答。**getreply** 操作仅能用在基于过程的（或混合型的）端口上。端口的类型定义将包括 **getreply** 操作所属的过程的名称。

当且仅当满足与 **getreply** 操作相关的匹配准则时，**getreply** 操作才会从输入端口队列中移去顶层应答。这些匹配准则与待处理过程的特征和通信伙伴有关，用于特征的匹配准则可以内置地说明，也可以源自一个特征模板。

可以用关键字 **value** 来说明对一个收到的返回值的匹配。

在一对多连接的情况下，可以将 **getreply** 操作限制于某个特定的通信伙伴，用关键字 **from** 来表示该限制。

例 1：

```
MyPort.getreply(MyProc:{5, ?} value 20); // 接受一个MyProc的应答，该应答带有两个out或
                                         // inout参数和一个返回值20。

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer; // 接受来自MyPeer的一个MyProc2的应答。
```

getreply 操作的特征变元不得用来为 **out** 和 **inout** 参数传入变量名。**out** 和 **inout** 参数值到变量的赋值将在 **getreply** 操作的赋值部分中进行。这允许以模板用于类型的相同方式来使用 **getreply** 操作中的特征模板。

getreply 操作的赋值部分（可选的）包含 **out** 和 **inout** 参数值到变量的赋值以及恢复应答发送方的地址。关键字 **value** 用 于恢复返回值，关键字 **param** 用于恢复应答的参数值。当需要恢复发送方的地址时，使用关键字 **sender**。

例 2：

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
// 返回值赋给变量MyRetValue，两个out或inout参数的值赋给变量MyPar1和MyPar2。

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender MySender;
// 在进一步的测试执行中，不考虑第一个参数的值。恢复发送方部件的地址并存储在变量MySender中。

// 下面的例子描述了将out或inout参数值赋给变量的可能性。为已被调用的过程假设如下特征。

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);

MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);

MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(MyVarInout1:=E);
```

23.3.4.1 获得任何应答

如果满足所有其它匹配准则，那么不带任何有关特征匹配准则的 **getreply** 操作将从输入端口队列的顶层移去应答消息（如果有的话）。*GetAnyReply* 接受的参数或返回值不得分配给变量。如果 *GetAnyReply* 用在响应和 **call** 操作的异常处理部分，那么它只能处理来自 **call** 操作所调用之过程的应答。

例如：

```
MyPort.getreply;              // 从MyPort移去顶层应答。

MyPort.getreply from MyPeer;  // 从MyPort移去接收自MyPeer的顶层应答。

MyPort.getreply -> sender MySenderVar;  // 从MyPort移去顶层应答，并恢复发送方实体的地址。
```

23.3.4.2 获得任何端口上的应答

使用关键字 **any port** 来获得任何端口上的应答。

例如：

```
any port.getreply(Myproc)
```

23.3.5 引起操作

raise 操作用于引起一个异常，仅能在基于过程的（或混合型的）端口上引起异常。异常是对已接受过程调用的一个反应，其结果是导致一个异常事件。将在被调用过程的特征中对异常的类型进行说明。端口的类型定义将在其已接受过程调用的列表中包含异常所属的过程名。

注 — 不可能总是静态地检查一个已接受调用和**raise**操作的关系。对测试而言，允许规定一个不带相关**getcall**操作的**raise**操作。

raise 操作的值部分由跟在异常值后的特征引用组成。

异常描述为类型，因此，异常值可以来自一个模板，也可以来自一个表达式的结果（当然，表达式可以是一个确定的值）。在需要避免发送值类型出现任何含糊的情况下，将使用针对 **raise** 操作的值说明中的可选类型字段。

一个或多个 **call** 操作的异常可以发送给连接于所寻址端口的一个、若干或全有对等实体。这可以用第 23.2.1.1 节中所述的相同方式来说明。这意味着：对单播异常，**raise** 操作的 **to** 子句的变元是一个接收实体的地址；对多播异常，**raise** 操作的 **to** 子句的变元是接收方集合的一个地址列表；对广播响应，**raise** 操作的 **to** 子句的变元是关键字 **all component**。

在一对一连接的情况下，可以省略 **to** 子句，原因是由系统结构来唯一地确定接收实体。

例如：

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
// 引起一个异常，带一个值，为MyPort处算术表达式的结果。

MyPort.raise(MyProc, integer:5});
//   引起一个异常，对MyProc，值为整数5。

MyPort.raise(MySignature, "My string") to MyPartner;
//   引起一个异常，对MySignature，值为MyPort处的“My string”，并将之发送给MyPartner。

MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
//   引起一个异常，值为MyPort处的“My string”，并将之发送给MyPartnerOne和MyPartnerTwo（即多播通信）。

MyPort.raise(MySignature, "My string") to all component;
//   引起一个异常，值为MyPort处的“My string”，并将之发送给连接至MyPort的所有实体（即广播通信）。
```

23.3.6 捕获操作

23.3.6.0 概述

catch 操作用于捕获由测试部件或 SUT 引起的异常，作为对过程调用的响应。**catch** 操作仅能用在基于过程的（或混合型的）端口上。将在引起异常的过程特征中说明被捕获异常的类型。异常被说明为类型，因此可以像消息那样对待它，例如，可以用模板来区分相同异常类型的不同值。

当且仅当顶层异常满足与 **catch** 操作相关的所有匹配准则时，**catch** 操作才从相关的输入端口队列中移去顶层异常。不会出现任何将输入值绑定于表达式术语或模板的情况。将异常值赋给变量将在 **catch** 操作的赋值部分中进行。

在一对多连接的情况下，**catch** 操作可限制于一个特定的通信伙伴。该限制用关键字 **from** 来表示。

例 1：

```
MyPort.catch(MyProc, integer: MyVar);   // 在端口MyPort处捕获MyProc引起的一个值为MyVar的整数异常。

MyPort.catch(MyProc, MyVar);            // 是之前例子的一个替换。

MyPort.catch(MyProc, A<B);              // 捕获一个布尔型异常。

MyPort.catch(MyProc, MyType:{5, MyVar}); // 一个异常值的内嵌式模板。

MyPort.catch(MyProc, charstring:"Hello")from MyPeer; // 捕获来自MyPeer的“Hello”异常。
```

catch 操作的（可选的）赋值部分包含异常值的赋值和调用部件地址的恢复。关键字 **value** 用于恢复一个异常的值，当需要恢复发送方的地址时，使用关键字 **sender**。

例 2：

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// 捕获来自MyPartner的一个异常，并将它的值赋给MyVar。

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// 捕获一个异常，将它的值赋给MyVarTwo，并恢复发送方的地址。
```

catch 操作可以是 **call** 操作响应和异常处理部分的一部分，或者用于确定 **alt** 语句的选择对象。如果 **catch** 操作用在 **call** 操作的接受部分，那么关于端口名的信息以及用于指示引起异常的过程的特征引用将是多余的，原因是该信息跟在 **call** 操作后。不过，由于可读性原因（如在复杂的 **call** 语句情况下），将重复该信息。

23.3.6.1 超时异常

存在一个由 **catch** 操作捕获的特殊的 **timeout** 异常。对被调用过程在预定时间内既不响应也不引起异常的情况，**timeout** 异常是一个紧急出口（见第 23.3.1.2 节）。

例如：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {            // 20ms后超时异常。
      setverdict(fail);
      stop;
     }
}
```

捕获 **timeout** 异常将限制于一个调用的异常处理部分。对处理 **timeout** 异常的 **catch** 操作来说，不允许有更多的匹配准则（包括一个 **from** 部分）和赋值部分。

23.3.6.2 捕获任何异常

没有任何变元列表的 **catch** 操作允许捕获任何有效的异常。 最通常的情况是不使用关键字 **from**。*CatchAnyException* 接受的异常值不得赋给一个变量。如果 *CatchAnyException* 用在一个 **call** 操作的响应和异常处理部分，那么它将只处理由 **call** 操作所调用的过程引起的异常。*CatchAnyException* 也将捕获 **timeout** 异常。

例如：

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

23.3.6.3 对任何端口的捕获

使用关键字 **any** 来 **catch** 任何端口上的异常。

例如：

```
any port.catch;
```

23.4 检查操作

23.4.0 概述

check 操作是一个普通的操作，它允许读取基于消息的和基于过程的输入端口队列的顶层元素，但不从该队列移去顶层元素。**check** 操作必须在基于消息的端口上处理某种类型的值，在基于过程的端口上区分要接受的调用、要捕获的异常以及来自之前调用的应答。

check 操作使用接收操作（**receive**、**getcall**、**getreply** 和 **catch**）及其匹配和赋值部分一起来定义必须要检查的条件，并在必要时提取它参数的值。

要检查的是输入端口队列的顶层元素（不可能检查队列）。如果队列是空的，那么 **check** 操作失败。如果队列不为空，那么复制队列的顶层元素，并在这个复制的顶层元素上执行在 **check** 操作中指定的接收操作。如果接收操作失败，那么 **check** 操作失败，也就是说不满足匹配准则。在这种情况下，抛弃复制的队列顶层元素，并按正常方式继续执行测试，也就是说计算 **check** 操作的下一条语句或选择对象。如果接收操作成功，那么 **check** 操作成功。

以错误的方式使用 **check** 操作，如在一个基于消息的端口上检查异常，将导致一个测试用例错误。

注 — 在大多数情况下，可以静态地检查**check**操作的正确用法，即在编译之前/之间进行检查。

例如：

```
MyPort1.check(receive(5)); // 检查一个值为5的整数消息。

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// 检查端口MyPort2上、来自MyPartner的、对MyProc的一个调用。

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// 检查端口MyPort2上返回值为20且两个out或inout参数的值为5和MyVar值的、来自过程MyProc的应答。

MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));

MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue param(MyPar1,-));

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

23.4.1 检查任何操作

不带任何变元列表的 **check** 操作允许检查在输入端口队列中是否有元素在等待处理。**check** 任何操作允许通过使用一个 **from** 子句来区分不同的发送方（在一对多连接的情况下），以及通过使用带有一个 **sender** 子句的简写形式的赋值部分来恢复发送方。在混合型端口的情况下，**check** 任何操作将检查基于消息的和基于过程的混合型端口输入队列。如果 **check** 任何操作对这两种混合型端口的输入队列都匹配，那么将对与基于过程的队列相关的信息赋予优先级，即作为 **check** 任何操作的结果返回。例如，如果混合型端口的、基于消息的和基于过程的输入队列不为空，并且由 **check** 任何操作来恢复发送方信息，那么将返回基于过程的输入队列中的调用、应答或异常的发送方。

注 — 通过结合使用**check**操作和**receive**任何操作，可以方便地恢复与混合型端口的、基于消息的输入队列相关的信息。例如：

```
MyPort.check(receive) -> sender Mysender.
```

例如：

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 对任何端口的检查

为了对任何端口进行 **check**，使用关键字 **any port**。在对任何不带变元的端口操作进行 **check** 的情况下，将对混合型端口的输入队列进行检查，如第 23.4.1 节所规定的那样。

例如：

```
any port.check;
```

23.5 控制通信端口

23.5.0 概述

TTCN-3 中用于控制基于消息的、基于过程的和混合型的端口的控制是：

- **clear**：移去输入端口队列的内容；
- **start**：移去输入端口队列的内容，并使能端口上的发送和接收操作；
- **stop**：使发送不能进行，且不允许接收操作匹配端口；
- **halt**：立即使端口上的发送操作不能进行，且不允许接收操作匹配在执行挂起操作后进入端口队列的、新的消息/调用/应答/异常。可继续处理已在队列中的条目。

23.5.1 清除端口操作

clear 操作移去指定端口的输入队列的内容。如果端口队列已为空，那么该操作不起任何作用。

例如：

```
MyPort.clear;  // 清除端口MyPort。
```

23.5.2 启动端口操作

如果一个端口定义为允许接收操作，如 **receive**、**getcall** 等，那么 **start** 操作清除指定端口的输入队列，并启动对端口上通信流的监听。如果定义端口允许发送操作，那么也允许在该端口上执行如 **send**、**call**、**raise** 等操作。

例如：

```
MyPort.start;  // 启动MyPort。
```

缺省地，在创建一个部件时，部件的所有端口都将被隐性地启动。启动端口操作将通过移去在输入队列中等待的所有消息来重新启动未被停止的端口。

23.5.3 停止端口操作

如果定义一个端口允许接收操作，如 **receive** 和 **getcall**，那么 **stop** 操作将导致停止对指定端口的监听。如果定义端口允许发送操作，那么 **stop** 端口不允许执行如 **send**、**call**、**raise** 等操作。

例 1：

```
MyPort.stop;  // 停止MyPort。
```

注 — 停止端口上的监听意味着，在stop操作之前定义的所有接收操作，将在端口的工作被挂起之前全部予以执行。

例 2：

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                // 对接收到的值进行解码，并与MyTemplate1匹配，
                                // 将匹配值保存在变量RecPDU中。
MyPort.stop;                    // 未执行任何在stop操作后定义的接收操作
                                // （除非通过后续的start操作重新启动端口）。
MyPort.receive (MyTemplate2);   // 该操作不匹配，将阻塞（假设没有任何缺省被激活）。
```

23.5.4 挂起端口操作

如果端口允许接收操作，如 **receive**、**trigger** 和 **getcall**，那么在该端口上执行 **halt** 操作后，**halt** 操作不允许接收操作紧跟在进入端口队列的消息和过程调用元素之后。对在 **halt** 操作之前已经存在于队列中的消息和过程调用元素，依然可以利用接收操作进行处理。如果端口允许发送操作，那么 **halt** 端口立即不允许执行发送操作，如 **send**、**call**、**raise** 等。后续的 **halt** 操作对端口的状态或其队列没有任何影响。

注 1 — 当执行操作时，端口**halt**操作实质上是在收到队列的最后一个条目后打上一个标记。标记之前的条目可以正常进行处理。在处理完标记之前的所有队列条目后，端口的状态等同停止状态。

注 2 — 如果在处理完标记之前的所有队列条目前，在一个挂起的端口上执行了一个端口**stop**操作，那么不允许立即执行更多的接收操作（也就是说，标记实质上已移至队列的顶层）。

注 3 — 挂起端口上的端口**start**操作清除队列中的所有条目，而不管它们是在执行端口**halt**操作之前还是之后到达。它也移去标记。

注 4 — 挂起端口上的一个端口**clear**操作清楚队列中的所有条目，而不管它们在执行端口**halt**操作之前或之后是否到达。事实上，它也在队列顶层打上标记。

例如：

```
MyPort.halt;                    // 从该时刻起，在MyPort上不允许任何发送；
                                // 仍可以处理队列中的消息。
MyPort.receive (MyTemplate1);   // 如果在挂起操作之前一个消息已在队列中，并且它匹配MyTemplate1，
                                // 那么对它进行处理；否则接收操作阻塞。
```

23.6 与端口一起使用any和all

如表 18 所示，关键字 **any** 和 **all** 可以与配置和通信操作一起使用。

表 18/Z.140－带端口的Any和All

操　作	允 许 的		例　子
	any	**all**	
receive、trigger、getcall、getreply、catch、check	是		any port.receive
connect/map			
Start、stop、clear、halt		是	all port.start

24 定时器操作

24.0 概述

TTCN-3 支持许多定时器操作，这些操作可以用在测试用例、函数、可选步骤和模板控制中。

假设声明定时器的每个 TTCN-3 范围单元都负责维护其自身的运行定时器列表和超时列表，即所有实际运行定时器的列表和所有超时定时器的列表。超时列表是测试用例执行时所照的快照的一部分。如果启动、停止、超时了在范围单元中的一个定时器或者如果执行了一个 **timeout** 操作，那么更新超时列表。

注 1 — 运行定时器列表和超时列表只是一个概念上的列表，并不限制定时器的实现。也可以使用其它的数据结构，如集合，此时，对超时事件的访问不受如超时事件发生的次序限制。

注 2 — 对每个测试部件，假设在对应的部件类型定义中声明了用于处理定时器启动/停止和定时器超时事件的一个特殊的运行定时器列表和超时列表。

当一个定时器到期时（从概念上讲，紧接在选择对象事件集合的快照处理之前），将超时事件置于声明定时器的范围单元的超时列表中。定时器立即变为非活动的。在任何时候，对任何特定的定时器而言，只有一个条目可以出现在声明定时器的范围单元的超时列表中。

在显性或隐性停止部件时，所有运行中的定时器都将被自动取消。

表 19/Z.140－TTCN-3定时器操作概述

定时器操作	
语句	**相关的关键字或符号**
启动定时器	**start**
停止定时器	**stop**
读取定时器已经过的时间	**read**
检查定时器是否正在运行	**running**
超时事件	**timeout**

24.1 启动定时器操作

start 定时器操作用于指明一个定时器应开始运行，定时器的值应是非负 **float** 数（即大于等于 0.0）。当启动一个定时器时，它的名称将被添加到运行定时器的列表中（对给定的范围单元）。

例如：

```
MyTimer1.start;         // 以缺省的持续时间来启动MyTimer1。
MyTimer2.start(20E-3); // 启动MyTimer2，持续时间为20ms。

// 也可以在一个循环中启动定时器数组的元素，例如：
timer t_Mytimer [5];
var float v_timerValues [5];

for (var integer i := 0; i<=4; i:=i+1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i<=4; i:=i+1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

如果没有给定任何缺省的持续时间，或者如果想要覆盖定时器声明中所述的缺省值，那么将使用可选的定时器值参数。当覆盖定时器持续时间时，新的值仅用于当前的定时器实例，该定时器任何之后的 **start** 操作，如果未指定持续时间，那么将使用缺省的持续时间。

启动一个定时器值为 0.0 的定时器意味着该定时器立刻就超时。启动一个带有负定时器值的定时器，如定时器的值是一个表达式的结果，或者不带指定的定时器值，将导致一个运行时间错误。

定时器时钟运行从浮点数值零（0.0）开始，直至持续时间参数所述的最大值。

可以对一个正在运行的定时器应用 **start** 操作，在这种情况下，定时器被停止并被重新启动。将从超时列表中移去该定时器超时列表中的任何条目。

24.2 停止定时器操作

stop 操作用于停止一个正在运行的定时器，并从正在运行的定时器列表中将之移去。一个被停止的定时器变成非活动的，它消耗的时间被设为浮点数值零（0.0）。

尽管没有任何影响，但停止一个非活动的定时器是一个有效的操作。停止一个超时的定时器将导致移去超时列表中有关该定时器的条目。关键字 **all** 可以用来停止调用 **stop** 操作的范围单元中可见的所有定时器。

例如：

```
MyTimer1.stop;      // 停止MyTimer1。
all timer.stop;     // 停止所有正在运行的定时器。
```

24.3 读定时器操作

read 操作用来恢复自指定定时器启动后消耗的时间，该返回值的类型应为 **float** 类型。

例如：

```
var float Myvar;
MyVar := MyTimer1.read; // 把MyTimer1启动后已经过的时间赋给MyVar。
```

对一个非活动的定时器应用 **read** 操作，即在一个未列于运行定时器列表中的定时器上，将返回浮点数值零（0.0）。

24.4 运行定时器操作

running 定时器操作用来检查一个定时器是否列于给定范围单元的运行定时器列表上（即定时器已被启动，且没有超时或没有被停止）。如果定时器列于列表中，那么操作返回值 **true**，否则返回 **false**。

例如：

```
if (MyTimer1.running) { … }
```

24.5 超时操作

timeout 操作允许检查一个测试部件范围单元中或调用超时操作的模块控制中的一个定时器或所有定时器是否已到期。

当处理 **timeout** 操作时，如果指定了定时器名称，那么根据 TTCN-3 的范围规则来搜索超时列表。如果存在与该定时器名称相匹配的超时事件，那么从超时列表中移去该事件，**timeout** 操作成功。**timeout** 操作不得用在一个 **boolean** 表达式中，但它可用来确定 **alt** 语句中的一个选择对象，或者作为行为描述中的一个独立语句。在后一种情况中，**timeout** 操作可认为是只有一个选择对象的 **alt** 语句的简写形式，也就是说，它具有阻塞语义，并因此提供被动等待定时器超时的能力。

例 1：

```
MyTimer1.timeout; // 检查之前启动的定时器MyTimer1的超时情况。
```

如果超时列表不为空，那么与 **timeout** 操作一起使用的关键字 **any** 跟在气候（而不是一个显性命名的定时器）。

例 2：

```
any timer.timeout; // 检查任何之前启动的定时器的超时情况。
```

24.6 和定时器一起使用的any与all概述

如表 20 所示，关键字 **any** 和 **all** 可以和定时器操作一起使用。

表 20/Z.140－带定时器的Any和All

操　作	允 许 的		例　子
	any	**all**	
start			
stop		是	**all timer.stop**
read			
running	是		**if (any timer.running) {…}**
timeout	是		**any timer.timeout**

25 测试判定操作

25.0 概述

判定操作允许分别使用 **setverdict** 和 **getverdict** 操作来设置和恢复判定。这些操作只能用在测试用例、可选步骤和函数中。

表 21/Z.140－TTCN-3测试判定操作概述

测试判定操作	
语句	**相关的关键字或符号**
设置本地判定	**setverdict**
获得本地判定	**getverdict**

活动配置的每个测试部件都应维护其自身的局部判定。局部判定是每个测试部件在创建之时为自己创建的一个对象，用于在每个测试部件中（即在 MTC 和每个 PTC 中）追踪单个的判定。

25.1 测试用例判定

另外，在每个测试部件（即 MTC 和每个 PTC）终止执行时，会有一个由测试系统实例化和处理的全局测试用例判定被更新。该判定对 **getverdict** 和 **setverdict** 操作而言是不可访问的。当测试用例终止执行时，测试用例将返回该判定的值。如果未显性地将返回的判定保存在控制部分中（如赋值给一个变量），那么丢弃它。

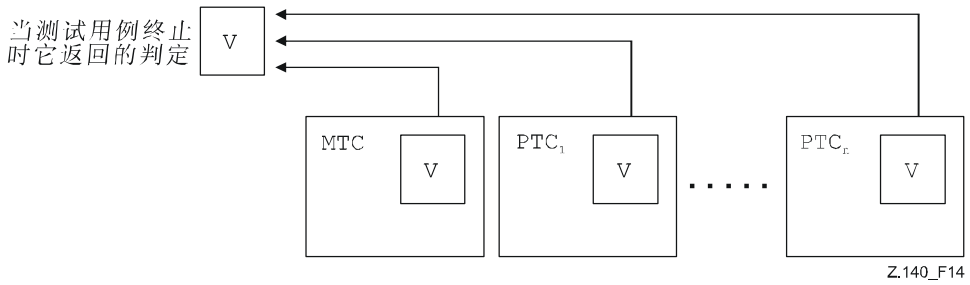


图 14/Z.140－判定之间关系的图示

注 — TTCN-3不描述执行本地和测试用例判定更新的实际机制。这些机制依赖于具体的实现方案。

25.2 判定值和覆盖规则

25.2.0 概述

判定可以有五个不同的值：**pass**、**fail**、**inconc**、**none** 和 **error**，即 **verdicttype** 的不同值（见第6.1 节）。

注 — **inconc**意味着一个不确定的判定。

只能使用值 **pass**、**fail**、**inconc** 和 **none** 来使用 **setverdict** 操作。

例 1：

```
setverdict(pass);
setverdict(inconc);
```

可以通过使用 **getverdict** 操作来恢复本地判定的值。

例 2：

```
MyResult := getverdict; // 当中MyResult是判定类型verdicttype的一个变量。
```

当一个测试部件被实例化时，创建其本地判定对象且设为 **none** 值。

当改变局部判定的值时（即使用 **setverdict** 操作），那么该改变的结果将遵循表 22 中所列的覆盖规则。测试用例判定在一个测试部件终止时隐性地予以更新。该隐性操作的结果也将遵循表 22 中所列的覆盖规则。

表 22/Z.140－判定的覆盖规则

判定的当前值	新判定指派值			
	pass	**inconc**	**fail**	**none**
none	**pass**	**inconc**	**fail**	**none**
pass	**pass**	**inconc**	**fail**	**pass**
inconc	**inconc**	**inconc**	**fail**	**inconc**
fail	**fail**	**fail**	**fail**	**fail**

例 3：

```
 :
setverdict(pass); // 本地判定设为pass。
 :
setverdict(fail); // 直至执行到这行，将导致本地判定值被改写为fail。
 :                 // 当ptc终止测试用例时，判定设为fail。
```

25.2.1 错误判定

error 判定是特别的，因为它由测试系统设置，用以指明发生了一个测试用例（即运行时间）错误。它不得由 **setverdict** 操作来设置，也不会通过 **getverdict** 操作来返回。没有任何其它判定值可以覆盖一个 **error** 判定，这意味着一个 **error** 判定只能是一个 **execute** 测试用例操作的结果。

26 外部动作

在某些测试情况下，可以缺少某些到 SUT 的接口，或者因此是未知的（如管理接口），但可能有必要刺激 SUT，以完成特定行为（如发送一条消息给测试系统）。另外，测试执行人员也可能要求特定的动作（如改变测试的环境条件，如温度、供电电压等）。

要求的行为可以描述为一个串表达式；即使用文字串、串类型的变量和参数等，并允许是它们的任何并联形式。

例如：

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString);  // 外部动作的非形式化描述。
```

对执行什么动作来触发该动作或由 SUT 来触发该动作未做任何规定，只对要求的动作本身有一个非形式化的描述。

可以在测试用例、函数、可选步骤和模板控制中使用外部动作。

27 模块控制部分

27.0 概述

测试用例在模块定义部分中定义，而模块控制部分负责管理其执行。如果要在该测试用例的行为定义中使用在模块控制部分中定义的所有变量（如果有的话），那么应通过参数化将它们传入测试用例；也就是说，TTCN-3 不支持任何种类的全局变量。

在每个测试用例启动之时，将重新设置测试配置。这意味着当停止测试用例时，毁掉在之前测试用例中由 **create**、**connect** 等操作管理的所有部件和端口（因此，对新的测试用例来说不是“可见的”）。

27.1 测试用例的执行

使用 **execute** 语句调用一个测试用例。作为测试用例的执行结果，返回 **none**、**pass**、**inconc**、**fail** 或 **error** 的一个测试用例判定，并可将之赋值给一个变量，以做进一步处理。

可选地，**execute** 语句允许通过定时器持续时间的方式来监管测试用例（见第 27.5 节）。

例如：

```
execute(MyTestCase1());                    // 执行MyTestCase1，不保存返回的测试判定，没有时间监管。

MyVerdict := execute(MyTestCase2());    // 执行MyTestCase2，并将结果判定存储在变量MyVerdict中。

MyVerdict := execute(MyTestCase3(),5E-3);
                                          // 执行MyTestCase3，并将结果判定存储在变量MyVerdict中。
                                          // 如果测试用例在5ms内不终止，那么MyVerdict将得到值“error”。
```

27.2 测试用例的终止

随着 MTC 的终止，测试用例终止。一旦 MTC（显性地或隐性地）终止，所有正在运行的并行测试部件都将被测试系统移去。

注 1 — 停止所有PTC的具体机制与特定的工具有关，因此超出了本建议书的讨论范围。

根据第 25 节中定义的规则，基于不同测试部件的最终局部判定，来计算测试用例的最终判定。当测试部件终止自己或者被自己、另一个测试部件或测试系统停止时，该测试部件的实际局部判定变为它的最终局部判定。

注 2 — 为了避免因被延时的PTC停止而引起的测试判定计算竞争条件，MTC应确保所有的PTC在停止自己之前都已停止（通过**done**或**killed**语句）。

27.3 测试用例的控制执行

限制于表 11 和表 12 中定义的那些编程语句，可以用在模块的控制部分中，用以说明如测试用例的执行次序或测试用例应该运行的次数等事项。

例如：

```
module MyTestSuite () {
    :
  control {
       :
      // 做10次该测试
      count:=0;
      while (count < 10)
      {    execute (MySimpleTestCase1());
       count := count+1;
      }
  }
}
```

如果没有使用任何编程语句，那么缺省地，以其在模块控制部分中的出现次序来执行测试用例。

注 — 这并不排除某些工具可能希望覆盖该缺省次序，以便允许用户或工具选择一个不同的执行次序。

测试用例的选择和取消选择也可用于控制测试用例的执行（见第 27.4 节）。

27.4 测试用例选择

在 TTN-3 中有不同的方法来选择和取消选择测试用例。例如，可以使用布尔表达式来选择和取消选择要执行的测试用例。当然，这包括返回 **boolean** 值的函数的使用。

例 1：

```
module MyTestSuite () {
    :
  control {
       :
     if (MySelectionExpression1()) {
          execute(MySimpleTestCase1());
          execute(MySimpleTestCase2());
          execute(MySimpleTestCase3());
     }
     if (MySelectionExpression2()) {
          execute(MySimpleTestCase4());
          execute(MySimpleTestCase5());
          execute(MySimpleTestCase6());
     }
      :
  }
}
```

另一种作为一个组来执行测试用例的方法是将它们集中在一个函数中，并从模块的控制部分执行该函数。

例 2：

```
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
}
 :
control
{  if (MySelectionExpression1()) { MyTestCaseGroup1(); }
   if (MySelectionExpression2()) { MyTestCaseGroup2(); }
    :
}
```

由于测试用例返回一个单个的 **verdicttype** 类型值，因此它也可能依据测试用例的结果来控制测试用例的执行次序。TTCN-3 **verdicttype** 的使用是另一种选择测试用例的方法。

例 3：

```
if ( execute (MySimpleTestCase()) == pass )
 { execute (MyGoOnTestCase()) }
else
 { execute (MyErrorRecoveryTestCase()) };
```

27.5 控制中定时器的使用

可以用定时器来监管测试用例的执行，可以在 **execute** 语句中使用一个显性的超时来完成这种监管。如果测试用例没有在该时间期限内结束，那么测试用例的执行结果将是一个错误的判定，测试系统将终止测试用例。用于测试用例监管的定时器是一个系统定时器，不必声明或启动它。

例 1：

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// 如果在7ms内MyTestCase没有完成执行的话，那么返回的判定将是错误的。
```

也可以显性地使用定时器操作来控制测试用例的执行。

例 2：

```
// 使用正在运行的定时器操作的例子。
while (T1.running or x<10) // 当中T1是一个之前已启动的定时器。
{  execute(MyTestCase());
   x := x+1;
}
```

```
// 使用启动和超时操作的例子。

timer T1 := 1.0;
 :
execute(MyTestCase1());
T1.start;
T1.timeout; // 在执行下一个测试用例之前暂停。
execute(MyTestCase2());
```

28 说明属性

28.0 概述

可以使用 **with** 语句来关联属性和 TTCN-3 语言要素。有关 **with** 语句参数（即实际属性）的语法定义为一个自由文本串。

有四种属性：

a） **display**：允许与特定的表示格式有关的显示属性说明；

b） **encode**：允许引用特定的编码规则；

c） **variant**：允许引用特定的编码变量；

d） **extension**：允许用户定义的属性说明。

28.1 显示属性

所有的 TTCN-3 语言要素都可以具有用于说明如何显示特殊语言要素的显示 **display** 属性，例如，以表格格式。

可以在 ITU-T Z.141 建议书 [1]中找到用于表格（一致性）表示格式的、与显示属性有关的专用属性串。

可以在 ITU-T Z.142 建议书 [2]中找到用于图形表示格式的、与显示属性有关的专用属性串。

其它的 **display** 属性可以由用户来定义。

注 — 由于用户定义的属性未标准化，因此不同工具对这些属性的解释可能不同，或甚至不支持用户定义的属性。

28.2 值的编码

28.2.0 概述

编码规则定义如何对一个特殊的值、模板等进行编码、如何在一个通信 **port** 上进行传输以及如何对接收到的信号进行解码。TTCN-3 没有缺省的编码机制，这意味着以 TTCN-3 之外某种方式来定义编码规则或编码指令。

在 TTCN-3 中，可以使用 **encode** 属性和 **variant** 属性来说明通用的或特殊的编码规则。

28.2.1 编码属性

encode 属性允许用于 TTCN-3 定义中的某些引用的编码规则或编码指令相关联。

定义实际编码规则的方式（如散文诗形式、函数形式等）超出了本建议书的讨论范围。如果没有引用任何特定的规则，那么编码将是一个单个实现的问题。

大多数情况下，将用层次法来使用编码属性。顶层是整个模块，下一层是一个组，最下层是一个单个的类型或定义：

a） **module**：编码适用于模块中定义的所有类型，包括TTCN-3类型（内置式类型）；

b） **group**：编码适用于一组用户定义的类型定义；

c） **type**或**definition**：编码适用于一个单个的用户定义的类型或定义；

d） **field**：编码适用于**record**类型、**set**类型或**template**中的一个字段。

例如：

```
module MyTTCNmodule
{ :
  import from MySecondModule {
      type MyRecord
  }
  with { encode "MyRule 1" } // 将依据MyRule 1来对MyRecord实例进行编码。

  :
  type charstring MyType;  //通常依据“全局编码规则”来编码。
  :
  group MyRecords
  {   :
      type record MyPDU1
      {
           integer field1,    // 将依据Rule 3来对field1进行编码。
           boolean field2,    // 将依据Rule 3来对field2进行编码。
           Mytype  field3     // 将依据Rule 2来对field3进行编码。
      }
      with { encode (field1, field2) "Rule 3" }
      :
  }
  with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.2 变量属性

使用 **variant** 属性来说明对当前所描述编码方案的改进，而不是替代它。因此，对变量属性，应用额外的覆盖规则（见第 28.5.1 节）。

例如：

```
module MyTTCNmodule1
{ :
  type charstring MyType;  //通常依据“全局编码规则”来编码。
  :
  group MyRecords
  {   :
      type record MyPDU1
      {
       integer    field1,
                              // 将依据“Rule 2”来对field1进行编码，使用编码变量“length form 3”。
       Mytype     field3
                              // 将依据“Rule 2”来对field3进行编码，使用任何可能长度的编码格式。
      }
      with { variant (field1) "length form 3" }
      :
  }
  with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.3 专用串

下列串是为简单的基本类型预定义的（标准化的）**variant** 属性（见第 E.2.1 节）：

a） 当用于整数和枚举类型时，“8位”和“无符号的8位”意味着整数值或者与枚举类型关联的整数在系统中将按8位（单字节）表示来处理。

b） 当用于整数和枚举类型时，“16位”和“无符号的16位”意味着整数值或者与枚举类型关联的整数在系统中将按16位（双字节）表示来处理。

c） 当用于整数和枚举类型时，“32位”和“无符号的32位”意味着整数值或者与枚举类型关联的整数在系统中将按32位（四字节）表示来处理。

d） 当用于整数和枚举类型时，“64位”和“无符号的64位”意味着整数值或者与枚举类型关联的整数在系统中将按64位（八字节）表示来处理。

e）当用于浮点数类型时，“IEEE754 浮点数”、“IEEE754 双精度数”、“IEEE754 扩展浮点数”和“IEEE754 扩展双精度数”意味着值将按标准的IEEE 754进行编码和解码（见参考文献）。

下列串是为 **charstring** 和 **universal charstring** 预定义的（标准化的）**variant** 属性（见第 E.2.2 节）：

a） 当用于通用字符串类型时，“UTF-8”意味着应依据ISO/IEC 10646 [10] 附件R中所定义的UCS转换格式8（UTF-8），分别对值的每一个字符进行编码和解码。

b） 当用于通用字符串类型时，“UCS-2”意味着应依据UCS-2编码的表示形式（见ISO/IEC 10646 [10] 的第14.1节），分别对值的每一个字符进行编码和解码。

c） 当用于通用字符串类型时，“UTF-16”意味着应依据ISO/IEC 10646 [10] 附件Q中所定义的UCS转换格式16（UTF-16），分别对值的每一个字符进行编码和解码。

d） 当用于字符串和通用字符串类型时，“8位”意味着应依据ISO/IEC 8859（一种8位编码方式）中描述的编码表示方式，分别对值的每一个字符进行编码和解码。

下列串是为结构化类型预定义的（标准化的）**variant** 属性（见第 E.2.3 节）：

a） 当用于一个记录类型时，“IDL:fixed FORMAL/01-12-01 v.2.6”意味着将值作为一个IDL定点十进制值来处理（见参考文献）。

这些变量属性可以与更多的、在高层描述的编码属性结合使用。例如，在一个自身具有全局编码属性“BER:1997”（见第 12.2 节/Z.146 [6]）的模块中，利用 **variant** 属性“UTF-8”描述的 **universal charstring**，将导致串中值的每个字符首先依据 UTF-8 规则进行编码，而后依据更全局的 BER 规则来对该 UTF-8 值进行编码。

28.2.4 无效的编码

如果想要描述无效的编码规则，那么可以按照与引用有效编码规则同样的方式，在模块外部的可引用源中，对这些无效的编码规则进行描述。

28.3 扩展属性

所有 TTCN-3 语言要素都可以拥有用户描述的 **extension** 属性。

注 — 由于用户定义的属性未被标准化，因此不同厂商提供的工具对这些属性的解释可能不同或甚至不支持这些属性。

28.4 属性的范围

with 语句可以将属性和一个单个的语言要素关联起来，也可能通过诸如列出属性语句中结构类型字段之类的方法，或者通过将一个 **with** 语句与周边范围单元或语言要素 **group** 关联起来的方法，将属性和多个语言要素关联起来，该属性语句与一个单个类型定义相关联。

例如：

```
// MyPDU1将显示为PDU。
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2将显示为带有应用特定的扩展属性MyRule的PDU。
type record MyPDU2 { … }
with
{
  display "PDU";
  extension "MyRule"
}

// 下列组定义……
group MyPDUs {
  type record MyPDU3 { … }
  type record MyPDU4 { … }
}
with {display "PDU"}   // 组MyPDUs的所有类型将显示为PDU。
```

```
// 等同于
group MyPDUs {
   type record MyPDU3 { … } with { display "PDU"}
   type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 属性的覆盖规则

较低作用范围单元中的属性定义将覆盖较高作用范围中的通用属性定义。有关变量属性的额外覆盖规则在第28.5.1 节中定义。

例 1：

```
type record MyRecordA
{
   :
} with { encode "RuleA" }

// 在下面，MyRecordA依据规则RuleA进行编码，而不是依据RuleB进行编码。
type record MyRecordB
{
    :
   field    MyRecordA
} with { encode "RuleB" }
```

放在另一个 **with** 语句范围内的 **with** 语句将覆盖最外面的 **with** 语句，这同样适用于带有组的 **with** 语句的使用情况。当覆盖方案用在引用与单个定义结合时，应格外小心。通用规则指的是应依据属性出现的次序来指定和覆盖属性。

```
// with语句覆盖方案使用举例
group MyPDUs
{
   type record MyPDU1 { … }
   type record MyPDU2 { … }

   group MySpecialPDUs
   {
       type record MyPDU3 { … }
       type record MyPDU4 { … }
   }
   with {extension "MySpecialRule"} // MyPDU3和MyPDU4将具有应用特定的扩展属性MySpecialRule。
}
with
{
   display "PDU";        //   组MyPDU的所有类型都将显示为PDU，并且
   extension "MyRule";  // （如果未被覆盖）将具有扩展属性MyRule。
}

// 等同于……
group MyPDUs
{
   type record MyPDU1 { … } with {display "PDU"; extension "MyRule" }
   type record MyPDU2 { … } with {display "PDU"; extension "MyRule" }
   group MySpecialPDUs {
       type record MyPDU3 { … } with {display "PDU"; extension "MySpecialRule" }
       type record MyPDU4 { … } with {display "PDU"; extension "MySpecialRule" }
   }
}
```

通过使用 **override** 指令，可在较高范围中覆盖较低范围中的属性定义 。

例 2：

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// 在下面，依据RuleB对MyRecordA进行编码。
type record MyRecordB
{
    :
   fieldA   MyRecordA
} with { encode override "RuleB" }
```

override 指令迫使所有较低范围内所含的类型都强制具有指定的属性。

28.5.1 变量属性的额外覆盖规则

variant 属性总是与 **encode** 属性相关。尽管编码的变量可以改变，但如果没有覆盖所有当前的变量属性，那么编码不得改变。因此，对变量属性，应用以下覆盖规则：

- 依据第28.5节中定义的规则，**variant**属性覆盖当前的**variant**属性；
- 依据第28.5节中定义的规则，覆盖当前**encoding**属性的**encoding**属性也将覆盖对应的当前**variant**属性；即不提供任何新的**variant**属性，但当前**variant**属性变为非活动的；
- 依据第28.6节中定义的规则，改变引入语言要素当前**encoding**属性的**encoding**属性也将改变对应的当前**variant**属性；即不提供任何新的**variant**属性，但当前**variant**属性变为非活动的。

例如：

```
module MyVariantEncodingModule {
   :
   type charstring MyCharString;     // 通常依据“Encoding 1”进行编码。
   :
   group MyVariantsOne {
       :
       type record MyPDUone
       {
        integer     field1,   // field1将只依据“Encoding 2”进行编码。
                              // “Encoding 2”覆盖“Encoding 1”和变量“Variant 1”
        Mytype      field3    // field3将依据“Encoding 1”进行编码，变量为“Variant 1”。
       }
       with { encoding (field1) "Encoding 2" }
       :
   }
   with { variant "Variant 1" }

   group MyVariantsTwo
   {   :
       type record MyPDUtwo
       {
        integer     field1,   // field1将依据“Encoding 3”进行编码，
                              // 使用编码变量“Variant 3”。
        Mytype      field3    // field3将依据“Encoding 3”进行编码，
                              // 使用编码变量“Variant 2”。
       }
       with { variant (field1) "Variant 3" }
       :
   }
   with { encode "Encoding 3"; variant 'Variant 2'}

}
with { encode "Encoding 1" }
```

28.6 改变引入语言要素的属性

通常，一个语言要素与其属性一起被引入。在某些情况下，当引入语言要素时，可能得改变这些属性，例如，可以在一个模块中将一个类型显示为 ASP，而后由另一个模块引入，在该模块中它应显示为 PDU。对此类情况，允许在 **import** 语句上改变属性。

例如：

```
import from MyModule {
   type MyType
}
with { display "ASP" }      // MyType将被显示为ASP。

import from MyModule {
   group MyGroup
}
with {
   display "PDU";       //   缺省地，所有类型都将被显示为PDU。
   extension "MyRule"
}
```

附 件 A

BNF和静态语义

A.1 TTCN-3 BNF

A.1.0 概述

本附件通过扩展 BNF（此后简称 BNF）来定义 TTCN-3 的语法。

A.1.1 语法描述的转换

表 A.1 为 TTCN-3 定义了用于描述扩展 BNF 语法的元记法。

表 A.1/Z.140－语法元记法

`::=`	被定义为
`abc xyz`	Abc 后跟 xyz
|	选择对象
`[abc]`	abc 的 0 或 1 个实例
`{abc}`	abc 的 0 或多个实例
`{abc}+`	abc 的 1 或多个实例
`(...)`	文本分组
`Abc`	非终结符 abc
`"abc"`	终结符 abc

A.1.2 语句终结符

通常，所有 TTCN-3 语言构件（即定义、声明、语句和操作）由分号（;）终止。如果语言构件用右括号（}）结束或后接一个右括号，即该语言构件是语句块、操作或声明中的最后一个语句，那么分号是可选的。

A.1.3 标识符

TTCN-3 标识符是大小写敏感的，只能包含小写字母（a-z）、大写字母（A-Z）和数字（0-9），也允许使用下划线（_）。一个标识符应以字母开头（即不能以数字或下划线开头）。

A.1.4 注释

以自由文本描写的注释可以出现在 TTCN-3 规格说明中的任意位置。

注释块应以符号对/*开始，以符号对*/结束。

例 1：

```
/* 这是一个注释块
占两行 */
```

注释块不得嵌套。

```
/* 这不是 /* 一个合法的 */ 注释 */
```

注释行应以符号对//开始，以一个<新行>结束。

例 2：

```
// 这是一个行注释
// 占两行
```

注释行可以接在 TTCN-3 程序语句的后面，但不得嵌入一个语句中。

例 3：

```
// 下面是不合法的
const // This is MyConst integer MyConst := 1;

//下面是合法的
const integer MyConst := 1; // This is MyConst
```

A.1.5 TTCN-3终结符

TTCN-3 终结符和保留字列于表 A.2 和表 A.3 中。

表 A.2/Z.140－TTCN-3特殊的终结符列表

块开始/结束符	**{ }**
列表开始/结束符	**( )**
可选对象符	**[ ]**
到达符号（在一个值域内）	**..**
行注释和块注释	**/* */ //**
行/语句终结符	**;**
算术运算符	**+ / -**
串连接运算符	**&**
等值运算符	**!= == >= <=**
串开始结束符	**" '**
通配符/匹配符	**? ***
赋值符	**:=**
通信操作赋值	**->**
比特串、十六进制串和八位字节串值	**B H O**
浮点指数	**E**

也应将表 10 中定义的、附件 C 描述的预定义函数标识符看作是保留字。

表 A.3/Z.140－作为保留字的TTCN-3终结符列表

action	fail	noblock	select
activate	false	none	self
address	float	not	send
alive	for	not4b	sender
all	from	nowait	set
alt	function	null	setverdict
altstep			signature
and	getverdict	octetstring	start
and4b	getcall	of	stop
any	getreply	omit	subset
anytype	goto	on	superset
	group	optional	system
bitstring		or	
boolean	hexstring	or4b	template
		out	testcase
case	if	override	timeout
call	ifpresent		timer
catch	import	param	to
char	in	pass	trigger
charstring	inconc	pattern	true
check	infinity	port	type
clear	inout	procedure	
complement	integer		union
component	interleave	raise	universal
connect		read	unmap
const	kill	receive	
control	killed	record	value
create			valueof
	label	rem	var
deactivate	language	repeat	variant
default	length	reply	verdicttype
disconnect	log	return	
display		running	while
do	map	runs	with
done	match		
	message		xor
else	mixed		xor4b
encode	mod		
enumerated	modifies		
error	module		
except	modulepar		
exception	mtc		
execute			
extends			
extension			
external			

表 A.3 中列出的 TTCN-3 终结符不得用作 TTCN-3 模块中的标识符。这些终结符都应全部小写。

A.1.6 TTCN-3语法BNF形式

A.1.6.0 TTCN-3模块

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                   "{"
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   "}"
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
/* 静态语义－只有当引用的模块包含TTCN-3记法时，才可以省略LanguageSpec。 */
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 模块定义部分

A.1.6.1.0 概述

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
                          TemplateDef |
                          ModuleParDef |
                          FunctionDef |
                          SignatureDef |
                          TestcaseDef |
                          AltstepDef |
                          ImportDef |
                          GroupDef |
                          ExtFunctionDef |
                          ExtConstDef) [WithStatement]
```

A.1.6.1.1 Typedef定义

```
12. TypeDef ::= TypeDefKeyword TypeDefBody
13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
14. TypeDefKeyword ::= "type"
15. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
16. RecordDef ::= RecordKeyword StructDefBody
17. RecordKeyword ::= "record"
18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
19. StructTypeIdentifier ::= Identifier
20. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
21. StructDefFormalPar ::=  FormalValuePar
/*    静态语义－FormalValuePar应解析为一个in参数。 */
22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                       [OptionalKeyword]
23. NestedTypeDef ::= NestedRecordDef |
                      NestedUnionDef |
                      NestedSetDef |
                      NestedRecordOfDef |
                      NestedSetOfDef |
                      NestedEnumDef
24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef} "}"
26. NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
30. StructFieldIdentifier ::= Identifier
31. OptionalKeyword ::= "optional"
32. UnionDef ::= UnionKeyword UnionDefBody
33. UnionKeyword ::= "union"
34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
```

```
                    "{" UnionFieldDef {"," UnionFieldDef} "}"
35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
36. SetDef ::= SetKeyword StructDefBody
37. SetKeyword ::= "set"
38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
39. OfKeyword ::= "of"
40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                "{" EnumerationList "}"
43. EnumKeyword ::= "enumerated"
44. EnumTypeIdentifier ::= Identifier
45. EnumerationList ::= Enumeration {"," Enumeration}
46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
47. EnumerationIdentifier ::= Identifier
48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
49. SubTypeIdentifier ::= Identifier
50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
/*    静态语义－AllowedValues应与子类型字段具有相同的类型 */
51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch ")"
52. ValueOrRange ::= RangeDef | ConstantExpression
/*    静态语义－RangeDef产生式应只能与基于integer, charstring, universal charstring或float的类型一起使用。 */
/*    静态语义－当细分字符串或通用字符串类型时，在同一SubTypeSpec中不得混合其值域和值。 */
53. RangeDef ::= LowerBound ".." UpperBound
54. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/*    静态语义－StringLength应只能与串类型一起使用，或者用于限制set of和record of类型。SingleConstExpression和
UpperBound应取非负整数值（当UpperBound包含无限大值时）。 */
55.   LengthKeyword ::= "length"
56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
57. PortDef ::= PortKeyword PortDefBody
58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
59. PortKeyword ::= "port"
60. PortTypeIdentifier ::= Identifier
61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
62. MessageAttribs ::= MessageKeyword
                       "{" {MessageList [SemiColon]}+ "}"
63. MessageList ::= Direction AllOrTypeList
64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
65. MessageKeyword ::= "message"
66. AllOrTypeList ::= AllKeyword | TypeList
/* 注：不赞成在端口定义中使用AllKeyword。 */
67. AllKeyword ::= "all"
68. TypeList ::= Type {"," Type}
69. ProcedureAttribs ::= ProcedureKeyword
                         "{" {ProcedureList  [SemiColon]}+ "}"
70. ProcedureKeyword ::= "procedure"
71. ProcedureList ::= Direction AllOrSignatureList
72. AllOrSignatureList ::= AllKeyword | SignatureList
73. SignatureList ::= Signature {"," Signature}
74. MixedAttribs ::= MixedKeyword
                     "{" {MixedList [SemiColon]}+ "}"
75. MixedKeyword ::= "mixed"
76. MixedList ::= Direction ProcOrTypeList
77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
78. ProcOrType ::= Signature | Type
79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                     [ExtendsKeyword ComponentType {"," ComponentType}]
                     "{" [ComponentDefList] "}"
80. ComponentKeyword ::= "component"
81. ExtendsKeyword ::= "extends"
82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
83. ComponentTypeIdentifier ::= Identifier
84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
85. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
86. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
87. PortElement ::= PortIdentifier [ArrayDef]
88. PortIdentifier ::= Identifier
```

A.1.6.1.2 常量定义

```
89. ConstDef ::= ConstKeyword Type ConstList
/* 静态语义－类型应遵循第9节给出的规则。 */
90. ConstList ::= SingleConstDef  {"," SingleConstDef}
91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/*    静态语义－ConstantExpression的值应与该常量已声明的类型相同。 */
92. ConstKeyword ::= "const"
```

```
93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 模板定义

```
94. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef] AssignmentChar TemplateBody
95. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
96. TemplateKeyword ::= "template"
97. TemplateIdentifier ::= Identifier
98. DerivedDef ::= ModifiesKeyword TemplateRef
99. ModifiesKeyword ::= "modifies"
100. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/*    静态语义－FormalValuePar应解析为一个in参数。 */
102. TemplateBody ::= (SimpleSpec | FieldSpecList | ArrayValueOrAttrib) | [ExtraMatchingAttributes]
/*    静态语义－在TeplateBody内，ArrayValueOrAttrib可用于数组、记录、record of和set of类型中。 */
103. SimpleSpec ::= SingleValueOrAttrib
104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
/*    静态语义－在FieldReference内，ArrayOrBitRef只可用于已修改模板中的record of和set of模板/模板字段。 */
107. StructFieldRef ::= StructFieldIdentifier| PredefinedType | TypeReference
/*    静态语义－PredefinedType和TypeReference只能用于anytype值记法。PredefinedType不得为AnyTypeKeyword。 */
108. ParRef ::= SignatureParIdentifier
/*    静态语义－SignatureParIdentifier将是来自相关特征定义的一个形参标识符。 */
109. SignatureParIdentifier ::= ValueParIdentifier
110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/*    静态语义－ArrayRef可选择地用于数组类型和ASN.1 SET OF和SEQUENCE OF类型以及TTCN-3 record of和set of类型。在一个ASN.1
或TTCN-3比特串类型内，Bit引用可使用相同的记法。 */
111. FieldOrBitNumber ::= SingleExpression
/*    静态语义－SingleExpression将解析为一个整数类型值。 */
112. SingleValueOrAttrib ::= MatchingSymbol |
                             SingleExpression |
                             TemplateRefWithParList
/*    静态语义－VariableIdentifier（通过singleExpression访问）仅可用在内置的模板定义中，用来引用当前范围内的变量。 */
113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
114. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch | TemplateBody
116. NotUsedSymbol ::= Dash
117. MatchingSymbol ::= Complement |
                        AnyValue |
                        AnyOrOmit |
                        ValueOrAttribList |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch
118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch  IfPresentMatch
119. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
121. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
125. CharStringMatch ::= PatternKeyword Cstring
126. PatternKeyword ::= "pattern"
127. Complement ::= ComplementKeyword ValueList
128. ComplementKeyword ::= "complement"
129. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
130. SubsetMatch ::= SubsetKeyword ValueList
/*    静态语义－Subset匹配只能与set of类型一起使用。 */
131. SubsetKeyword ::= "subset"
132. SupersetMatch ::= SupersetKeyword ValueList
/*    静态语义－Superset匹配只能与set of类型一起使用。 */
133. SupersetKeyword ::= "superset"
134. PermutationMatch ::= PermutationKeyword PermutationList
135. PermutationKeyword ::= "permutation"
136. PermutationList ::= "(" TemplateBody { "," TemplateBody } ")"
/*    静态语义－对TemplateBody 内容的限制在第B.1.3.3节中给出。 */
137. AnyValue ::= "?"
138. AnyOrOmit ::= "*"
```

```
139. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
140. LengthMatch ::= StringLength
141. IfPresentMatch ::= IfPresentKeyword
142. IfPresentKeyword ::= "ifpresent"
143. Range ::= "(" LowerBound ".." UpperBound ")"
144. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
145. UpperBound ::= SingleConstExpression | InfinityKeyword
/*    静态语义－LowerBound和UpperBound将当作整数、字符串、通用字符串或浮点数类型进行计算。当LowerBound或UpperBound当作
字符串或通用字符串类型进行计算时，只能出现SingleConstExpression，且串长度为1。 */
146. InfinityKeyword ::= "infinity"
147. TemplateInstance ::= InLineTemplate
148. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList] |
                                TemplateParIdentifier
149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
150. InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                        TemplateBody
/*    静态语义－当类型隐性地清晰时，type字段才能被省略。 */
151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
152. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
153. TemplateActualPar ::= TemplateInstance
/*    静态语义－当相应的形参不是模板类型时，TemplateInstance产生式将解析为一个或多个SingleExpressions。 */
154. TemplateOps ::= MatchOp | ValueofOp
155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/*    静态语义－表达式返回的值的类型必须与模板类型相同，且模板的每个字段都将解析为一个单个的值。 */
156. MatchKeyword ::= "match"
157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 函数定义

```
159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                     "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                     StatementBlock
160. FunctionKeyword ::= "function"
161. FunctionIdentifier ::= Identifier
162. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
163. FunctionFormalPar ::= FormalValuePar |
                           FormalTimerPar |
                           FormalTemplatePar |
                           FormalPortPar
164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
/*    静态语义－模板关键字的使用应遵循第16.1.0节中的限制。 */
165. ReturnKeyword ::= "return"
166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
167. RunsKeyword ::= "runs"
168. OnKeyword ::= "on"
169. MTCKeyword ::= "mtc"
170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
171. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
172. FunctionStatementOrDef ::= FunctionLocalDef |
                                FunctionLocalInst |
                                FunctionStatement
173. FunctionLocalInst ::= VarInstance | TimerInstance
174. FunctionLocalDef ::= ConstDef | TemplateDef
175. FunctionStatement ::= ConfigurationStatements |
                           TimerStatements |
                           CommunicationStatements |
                           BasicStatements |
                           BehaviourStatements |
                           VerdictStatements |
                           SUTStatements
176. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
                     PreDefFunctionIdentifier
178. PreDefFunctionIdentifier ::= Identifier
/*    静态语义－标识符是附件C中预定义TTCN-3函数标识符之一。 */
179. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
180. FunctionActualPar ::= TimerRef |
                           TemplateInstance |
                           Port |
                           ComponentRef
/* 静态语义－当相应的形参不是模板类型时，TemplateInstance产生式将解析为一个或多个SingleExpressions，即等同于表达式产生式。 */
```

A.1.6.1.5 特征定义

```
181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                      "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                      [ExceptionSpec]
182. SignatureKeyword ::= "signature"
183. SignatureIdentifier ::= Identifier
184. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
185. SignatureFormalPar ::=  FormalValuePar
186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
187. ExceptionKeyword ::= "exception"
188. ExceptionTypeList ::= Type {"," Type}
189. NoBlockKeyword ::= "noblock"
190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 测试用例定义

```
191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                     "("[TestcaseFormalParList] ")" ConfigSpec
                     StatementBlock
192. TestcaseKeyword ::= "testcase"
193. TestcaseIdentifier ::= Identifier
194. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
195. TestcaseFormalPar ::= FormalValuePar |
                           FormalTemplatePar
196. ConfigSpec ::= RunsOnSpec [SystemSpec]
197. SystemSpec ::= SystemKeyword ComponentType
198. SystemKeyword ::= "system"
199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")"
                          ["," TimerValue] ")"
200. ExecuteKeyword ::= "execute"
201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
202. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
203. TestcaseActualPar ::= TemplateInstance
/*    静态语义－当相应的形参不是模板类型时，TemplateInstance产生式将解析为一个或多个SingleExpressions，即等同于表达式产生式。 */
```

A.1.6.1.7 可选步骤定义

```
204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                    "("[AltstepFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
205. AltstepKeyword ::= "altstep"
206. AltstepIdentifier ::= Identifier
207. AltstepFormalParList ::= FunctionFormalParList
/*    静态语义－激活为缺省的可选步骤只能有in参数、端口参数或定时器参数。 */
/*    静态语义－只有在alt语句中作为可选对象或在TTCN-3行为描述中作为独立语句被调用的可选步骤才能有in、out和inout参数。 */
208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
209. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef | TemplateDef
/*    静态语义－AltstepLocalDef应满足第16.2.2.1节中的限制。 */
210. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
/* STATIC SEMANTICS - all timer instances in FunctionActualParList shall be declared as component local timers
(see also production ComponentElementDef) */
/*    静态语义－FunctionActualParList中的所有定时器实例都将声明为部件局部定时器（也可参见产生式ComponentElementDef）。 */
211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 引入定义

```
212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
213. ImportKeyword ::= "import"
214. AllWithExcepts ::= AllKeyword [ExceptsDef]
215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
216. ExceptKeyword ::= "except"
217. ExceptSpec ::= {ExceptElement [SemiColon]}
/*    静态语义－任意一个产生式部件（ExceptGroupSpec、ExceptTypeDefSpec等）都只能在ExceptSpec产生式中出现一次。 */
218. ExceptElement ::= ExceptGroupSpec |
                      ExceptTypeDefSpec |
                      ExceptTemplateSpec |
                      ExceptConstSpec |
                      ExceptTestcaseSpec |
                      ExceptAltstepSpec |
                      ExceptFunctionSpec |
                      ExceptSignatureSpec |
                      ExceptModuleParSpec
219. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
220. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
```

```
221. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
222. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
223. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
224. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
225. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
226. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
227. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
228. ImportSpec ::= {ImportElement [SemiColon]}
229. ImportElement ::= ImportGroupSpec |
                       ImportTypeDefSpec |
                       ImportTemplateSpec |
                       ImportConstSpec |
                       ImportTestcaseSpec |
                       ImportAltstepSpec |
                       ImportFunctionSpec |
                       ImportSignatureSpec |
                       ImportModuleParSpec
230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
/* 注：不赞成使用RecursiveKeyword。 */
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {"," FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                          EnumTypeIdentifier |
                          PortTypeIdentifier |
                          ComponentTypeIdentifier |
                          SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
247. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 组定义

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 外部函数定义

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")" [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 外部常量定义

```
271. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
/*    静态语义—类型应遵循第9节中给出的规则。 */
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 模块参数定义

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar | ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
/*    静态语义－ConstantExpression的值的类型应与为参数声明的类型相同。 */
277. ModuleParType ::= Type
/*    静态语义－类型不能是部件、缺省或anytype类型。如果地址类型的定义在模块内定义，那么类型将只解析为地址类型。 */
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar ConstantExpression]
                      {","ModuleParIdentifier [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 控制部分

A.1.6.2.0 概述

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
/* 静态语义－控制部分中有关语句使用的限制在表11中给出。 */
```

A.1.6.2.1 变量实例化

```
286. VarInstance ::= VarKeyword ((Type VarList) | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 定时器实例化

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
/*    静态语义－当表达式解析为SingleExpression时，它必须解析为一个浮点数类型值。在将缺省定时器值赋给定时器数组的初始化过程中
，表达式只能解析为CompoundExpression。 */
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier) {ArrayOrBitRef}
```

A.1.6.2.3 部件操作

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword ["(" SingleExpession ")"] [AliveKeyword]
/*    静态语义－对SingleExpression的限制参见第22.1节。 */
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
```

```
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
323. ComponentRef ::= ComponentOrDefaultReference | SystemOp | SelfOp | MTCOp
324. DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                    AllConnectionsSpec |
                                    AllPortsSpec |
                                    AllCompsAllPortsSpec]
326. AllConnectionsSpec ::= "(" PortRef ")"
327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword PortKeyword ")"
329. DisconnectKeyword ::= "disconnect"
330. MapStatement ::= MapKeyword SingleConnectionSpec
331. MapKeyword ::= "map"
332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
333. UnmapKeyword ::= "unmap"
334. StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword "(" FunctionInstance ")"
/* 静态语义－函数实例只能有in参数。 */
/* 静态语义－函数实例不能有定时器参数。 */
335. StartKeyword ::= "start"
336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral Dot StopKeyword) |
                        (AllKeyword ComponentKeyword Dot StopKeyword)
337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference | MTCOp | SelfOp
338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral Dot KillKeyword) |
                         (AllKeyword ComponentKeyword Dot KillKeyword)
339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
/* 静态语义－当在配置语句中使用时，与VariableRef关联的变量或与FunctionInstance关联的返回类型必须是部件类型；当在非活动的语句
中使用时，它们必须是缺省类型。 */
340. KillKeyword ::= "kill"
```

A.1.6.2.4 端口操作

```
341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
342. CommunicationStatements ::= SendStatement |
                                 CallStatement |
                                 ReplyStatement |
                                 RaiseStatement |
                                 ReceiveStatement |
                                 TriggerStatement |
                                 GetCallStatement |
                                 GetReplyStatement |
                                 CatchStatement |
                                 CheckStatement |
                                 ClearStatement |
                                 StartStatement |
                                 StopStatement
343. SendStatement ::= Port Dot PortSendOp
344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
345. SendOpKeyword ::= "send"
346. SendParameter ::= TemplateInstance
347. ToClause ::= ToKeyword AddressRef |
                  AddressRefList |
                  AllKeyword ComponentKeyword
/* 静态语义－AddressRef不得包含匹配机制。 */
348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
349. ToKeyword ::= "to"
350. AddressRef ::= TemplateInstance
/* 静态语义－TemplateInstance必须是地址或部件类型。 */
351. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
353. CallOpKeyword ::= "call"
354. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* 静态语义－只有out参数可以省略或用某个匹配的属性加以规定。 */
355. CallTimerValue ::= TimerValue | NowaitKeyword
/* 静态语义－值必须是浮点数类型。 */
356. NowaitKeyword ::= "nowait"
357. PortCallBody ::= "{" CallBodyStatementList "}"
```

```
358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
359. CallBodyStatement ::= CallBodyGuard  StatementBlock
360. CallBodyGuard ::= AltGuardChar CallBodyOps
361. CallBodyOps ::= GetReplyStatement | CatchStatement
362. ReplyStatement ::= Port Dot PortReplyOp
363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
364. ReplyKeyword ::= "reply"
365. ReplyValue ::= ValueKeyword Expression
366. RaiseStatement ::= Port Dot PortRaiseOp
367. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
368. RaiseKeyword ::= "raise"
369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
370. PortOrAny ::= Port | AnyKeyword PortKeyword
371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－当且仅当ReceiveParameter选项也存在时，PortRedirect选项才出现。 */
372. ReceiveOpKeyword ::= "receive"
373. ReceiveParameter ::= TemplateInstance
374. FromClause ::= FromKeyword AddressRef
375. FromKeyword ::= "from"
376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
377. PortRedirectSymbol ::= "->"
378. ValueSpec ::= ValueKeyword VariableRef
379. ValueKeyword ::= "value"
380. SenderSpec ::= SenderKeyword VariableRef
/* 静态语义－变量引用必须是地址或部件类型。 */
381. SenderKeyword ::= "sender"
382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－当且仅当ReceiveParameter选项也出现时，PortRedirect选项才出现。 */
384. TriggerOpKeyword ::= "trigger"
385. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                      [PortRedirectWithParam]
/* 静态语义－当且仅当ReceiveParameter选项也出现时，PortRedirectWithParam选项才出现。 */
387. GetCallOpKeyword ::= "getcall"
388. PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                               SenderSpec
390. ParamSpec ::= ParamKeyword ParamAssignmentList
391. ParamKeyword ::= "param"
392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
394. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* 静态语义－parameterIdentifiers必须来自相应的特征定义。 */
395. ParameterIdentifier ::= ValueParIdentifier
396. VariableList ::= VariableEntry  {"," VariableEntry}
397. VariableEntry ::= VariableRef | NotUsedSymbol
398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
                        [FromClause] [PortRedirectWithValueAndParam]
/* 静态语义－当且仅当ReceiveParameter选项也出现时，PortRedirectWithParam选项才出现。 */
400. PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                       RedirectWithParamSpec
402. GetReplyOpKeyword ::= "getreply"
403. ValueMatchSpec ::= ValueKeyword TemplateInstance
404. CheckStatement ::= PortOrAny Dot PortCheckOp
405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
406. CheckOpKeyword ::= "check"
407. CheckParameter ::= CheckPortOpsPresent | FromClausePresent | RedirectPresent
408. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
409. RedirectPresent ::= PortRedirectSymbol SenderSpec
410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
411. CatchStatement ::= PortOrAny Dot PortCatchOp
412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－当且仅当CatchOpParameter选项也出现时，PortRedirect选项才出现。 */
413. CatchOpKeyword ::= "catch"
414. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
415. ClearStatement ::= PortOrAll Dot PortClearOp
416. PortOrAll ::= Port | AllKeyword PortKeyword
417. PortClearOp ::= ClearOpKeyword
418. ClearOpKeyword ::= "clear"
419. StartStatement ::= PortOrAll Dot PortStartOp
420. PortStartOp ::= StartKeyword
```

```
421. StopStatement ::= PortOrAll Dot PortStopOp
422. PortStopOp ::= StopKeyword
423. StopKeyword ::= "stop"
424. AnyKeyword ::= "any"
```

A.1.6.2.5 定时器操作

```
425. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
426. TimerOps ::= ReadTimerOp | RunningTimerOp
427. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
430. ReadTimerOp ::= TimerRef Dot ReadKeyword
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 类型

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                       EnumTypeIdentifier |
                       SubTypeIdentifier |
                       ComponentTypeIdentifier
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds ["..." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
/* 静态语义—ArrayBounds将解析为一个非负的整数类型值。 */
```

A.1.6.4 值

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
                         OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
```

```
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier |
                        ModuleParIdentifier ) |
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" "B"
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" "H"
486. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* 参考一由相关的CharacterString类型定义的一个字符。对字符串类型，字符来自ITU-T T.50建议书中定义的字符集。
对通用字符串类型，字符来自ISO/IEC 10646中定义的任何字符集。 */
491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
493. AlphaNum ::= Alpha | Num
494. UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" |
"P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
495. LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" |
"p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
496. ExtendedAlphaNum ::= /* 参考一来自基本拉丁文或来自ISO/IEC 10646中定义的LATIN-1补充字符集（从字符(0,0,0,32)到字符
(0,0,0,126)，从字符 (0,0,0,161)到字符(0,0,0,172)以及从字符(0,0,0,174)到字符(0,0,0,255)）的图形字符。 */
497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 参数化

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
509. TimerParIdentifier ::= Identifier
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           TemplateKeyword Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 With语句

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
```

```
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
/* 静态语义－DefOrFieldRef必须指的是模块、组或与with语句相关的定义中的定义或字段。 */
526. DefinitionRef ::= StructTypeIdentifier |
                      EnumTypeIdentifier |
                      PortTypeIdentifier |
                      ComponentTypeIdentifier |
                      SubTypeIdentifier |
                      ConstIdentifier |
                      TemplateIdentifier |
                      AltstepIdentifier |
                      TestcaseIdentifier |
                      FunctionIdentifier |
                      SignatureIdentifier |
                      VarIdentifier |
                      TimerIdentifier |
                      PortIdentifier |
                      ModuleParIdentifier |
                      FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{" GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{" TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{" TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{" ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{" AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{" TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{" FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{" SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{" ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 行为语句

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
/* 静态语义－不能从一个已有的、正在执行的测试用例或从测试用例调用的函数链中调用TestcaseInstance，即只能从控制部分或直接调用自控制部分的函数来初始化测试用例。 */
/* 静态语义－不能从模块控制部分内调用ActivateOp。 */
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* 静态语义－SingleExpression必须解析为一个判定类型的值。 */
/* 静态语义－SetLocalVerdict不能用于赋error值。 */
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ] {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
537. ActionText ::= FreeText | Expression
/* 静态语义－表达式将拥有基本类型字符串或通用字符串。 */
538. ReturnStatement ::= ReturnKeyword  [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] | GuardOp StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
/* 静态语义－BooleanExpression应满足第20.1.2节中的限制。 */
545. GuardOp ::= TimeoutStatement |
                ReceiveStatement |
                TriggerStatement |
                GetCallStatement |
                CatchStatement |
                CheckStatement |
                GetReplyStatement |
                DoneStatement |
                KilledStatement
/* 静态语义－在模块控制部分内部使用的GuardOp只能包含timeoutStatement。 */
546. InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
```

```
549. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
/* 静态语义－StatementBlock不能包含loop语句、跳转（goto）、激活（activate）、去激活（deactivate）、停止（stop）、返回（return）或对函数的调用。 */
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
559. ActivateKeyword ::= "activate"
560. DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference ")"]
561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 基本语句

```
562. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct |
                         SelectCaseConstruct
563. Expression ::= SingleExpression | CompoundExpression
/* 静态语义－表达式不能包含模块控制部分内的配置、激活操作或判定操作。 */
564. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* 静态语义－在CompoundExpression内，ArrayExpression可用于数组、记录、record of和set of类型。 */
565. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
566. FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
568. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
569. NotUsedOrExpression ::= Expression | NotUsedSymbol
570. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
571. SingleConstExpression ::= SingleExpression
/* 静态语义－SingleConstExpression不能包含变量或模块参数，并应在编译时解析为一个固定值。 */
572. BooleanExpression ::= SingleExpression
/* 静态语义－BooleanExpression将解析为一个布尔类型值。 */
573. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* 静态语义－在CompoundConstExpression内，ArrayConstExpression可用于数组、记录、record of和set of类型。 */
574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
575. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
576. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
577. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
578. Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)
/* 静态语义－对值变量，赋值符号右侧的表达式将当作与赋值符号左侧之类型兼容的类型的一个明确值；对模板变量，赋值符号右侧的表达式将当作与赋值符号左侧之类型兼容的一个明确值、模板（文字或模板实例）或匹配机制。 */
579. SingleExpression ::= XorExpression { "or" XorExpression }
/* 静态语义－如果存在多个XorExpression，那么XorExpression将当作兼容类型的特定值。 */
580. XorExpression ::= AndExpression { "xor" AndExpression }
/* 静态语义－如果存在多个AndExpression，那么AndExpression将当作兼容类型的特定值。 */
581. AndExpression ::= NotExpression { "and" NotExpression }
/* 静态语义－如果存在多个NotExpression，那么NotExpression将当作兼容类型的特定值。 */
582. NotExpression ::= [ "not" ] EqualExpression
/* 静态语义－not运算符的操作数应是布尔类型（TTCN或ASN.1）或布尔类型的派生类型。 */
583. EqualExpression ::= RelExpression { EqualOp RelExpression }
/* 静态语义－如果存在多个RelExpression，那么RelExpression将当作兼容类型的特定值。 */
584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
/* 静态语义－如果两个ShiftExpressions都存在，那么每个ShiftExpressions都将当作是一个特定的整数、枚举或浮点数值（这些值可以是TTCN或ASN.1值）或这些类型的派生类型值。 */
585. ShiftExpression ::= BitOrExpression { ShiftOp BitOrExpression }
/* 静态语义－每个结果都解析为一个特定的值。如果存在多个结果，那么右侧操作数应是整数类型或其派生类型，并且如果移动运算符是'<<'或'>>'，那么左侧操作数应解析为bitstring、hexstring或octetstring类型或是这些类型的派生类型。如果移动运算符是'<@'或'@>'，那么左侧操作数应是bitstring、hexstring、charstring或universal charstring类型或是这些类型的派生类型。 */
586. BitOrExpression ::= BitXorExpression { "or4b" BitXorExpression }
/* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall evaluate to specific values of compatible types */
/* 静态语义－如果存在多个BitXorExpression，则BitXorExpression会计算为兼容类型的特定值*/
587. BitXorExpression ::= BitAndExpression { "xor4b" BitAndExpression }
```

```
/* 静态语义－如果存在多个BitAndExpression，那么BitAndExpression将当作兼容类型的特定值。 */
588. BitAndExpression ::= BitNotExpression { "and4b" BitNotExpression }
/* 静态语义－如果存在多个BitNotExpression，那么BitNotExpression将当作兼容类型的特定值。 */
589. BitNotExpression ::= [ "not4b" ] AddExpression
/* 静态语义－如果存在not4b运算符，那么操作数应是bitstring、octetstring或hexstring类型或这些类型的派生类型。 */
590. AddExpression ::= MulExpression { AddOp MulExpression }
/* 静态语义－每个MulExpression都解析为一个特定的值。如果存在多个MulExpression，且AddOp解析为StringOp，那么
MulExpressions应解析为bitstring、hexstring、octetstring、charstring或 universal charstring类型或这些类型的派生类型
。如果存在多个MulExpression，但AddOp不是解析为StringOp，那么MulExpression将解析为整数类型或浮点数类型或这些类型的派生类型
。 */
591. MulExpression ::= UnaryExpression { MultiplyOp UnaryExpression }
/* 静态语义－每个UnaryExpression都将解析为一个特定的值。如果存在多个UnaryExpression，那么UnaryExpression将解析为整数类型
或浮点数类型或这些类型的派生类型。 */
592. UnaryExpression ::= [ UnaryOp ] Primary
/* 静态语义－Primary将解析为整数类型或浮点数类型或这些类型派生类型的一个特定值。 */
593. Primary ::= OpCall | Value | "(" SingleExpression ")"
594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | TypeDefIdentifier))
                                | ArrayOrBitRef }+
/* 静态语义－当且仅当使用ExtendedFieldReference的VarInstance或ReferencedValue的类型为anytype时，才能使用
TypeDefIdentifier。 */
595. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                FunctionInstance |
                TemplateOps |
                ActivateOp
596. AddOp ::= "+" | "-" | StringOp
/* 静态语义－“+”或“-”运算符的操作数应是整数类型或浮点数类型，或者是整数类型或浮点数类型的派生类型（即子域）。 */
597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* 静态语义－“*”、“/”、rem或mod运算符的操作数应是整数类型或浮点数类型，或者是整数类型或浮点数类型的派生类型（即子域）。 */
598. UnaryOp ::= "+" | "-"
/* 静态语义－“+”或“-”运算符的操作数应是整数类型或浮点数类型，或者是整数类型或浮点数类型的派生类型（即子域）。 */
599. RelOp ::= "<" | ">" | ">=" | "<="
/* 静态语义－运算符的优先级在表7中定义。 */
600. EqualOp ::= "==" | "!="
601. StringOp ::= "&"
/* 静态语义－串运算符的操作数应是bitstring、hexstring、octetstring或character string类型。 */
602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
603. LogStatement ::= LogKeyword "(" LogItem { "," LogItem } ")"
604. LogKeyword ::= "log"
605. LogItem ::= FreeText | TemplateInstance
606. LoopConstruct ::= ForStatement |
                      WhileStatement |
                      DoWhileStatement
607. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                      StatementBlock
608. ForKeyword ::= "for"
609. Initial ::= VarInstance | Assignment
610. Final ::= BooleanExpression
611. Step ::= Assignment
612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                        StatementBlock
613. WhileKeyword ::= "while"
614. DoWhileStatement ::= DoKeyword StatementBlock
                          WhileKeyword "(" BooleanExpression ")"
615. DoKeyword ::= "do"
616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                              StatementBlock
                              {ElseIfClause}[ElseClause]
617. IfKeyword ::= "if"
618. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
619. ElseKeyword ::= "else"
620. ElseClause ::= ElseKeyword  StatementBlock
621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
622. SelectKeyword ::= "select"
623. SelectCaseBody ::= "{" { SelectCase }+ "}"
```

```
624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {"," TemplateInstance } ')' | ElseKeyword )
                    StatementBlock
625. CaseKeyword ::= "case"
```

A.1.6.9 其它产生式

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

附 件 B

匹配输入值

B.1 模板匹配机制

B.1.0 概述

本附件描述了在 TTCN-3 模板中可以使用的匹配机制（且仅用在模板中）。

B.1.1 匹配特定值

特定值是 TTCN-3 模板的基本匹配机制。模板中的特定值是不包含任何匹配机制或通配符的表达式。除非另有规定，否则当且仅当输入字段值与模板中表达式的值完全相同时，模板字段才匹配相应的输入字段值。

例如：

```
// 假定消息类型定义
type record MyMessageType
{
  integer      field1,
  charstring   field2,
  boolean      field3 optional,
  integer[4]   field4
}

// 使用特定值的消息模板可以是
template MyMessageType MyTemplate:=
{
  field1 := 3+2,           // 整数类型的特定值
  field2 := "My string",   // 字符串类型的特定值
  field3 := true,          // 布尔类型的特定值
  field4 := {1,2,3}        // 整数数组的特定值
}
```

B.1.1.1 省略值

关键字 **omit** 表示应忽略一个可选的模板字段。倘若模板字段是可选的，那么它可用于所有类型的值。

例如：

```
template Mymessage MyTemplate:=
{ :
  :
  field3 := omit, // omit this field
  :
}
```

B.1.2 代替值的匹配机制

B.1.2.0 概述

下面的匹配机制可以用来代替明确的值。

B.1.2.1 值列表

值列表用列表方式列出可接受的输入值，它可用于所有类型的值。当且仅当输入字段值匹配值列表中的任何一个值时，使用值列表的模板字段才匹配相应的输入字段。值列表中每个值的类型应与为使用该匹配机制的模板字段所声明的类型相同。

例如：

```
template Mymessage MyTemplate:=
{
  field1 := (2,4,6),                // 整数值列表
  field2 := ("String1", "String2"),// 字符串值列表
  :
  :
}
```

B.1.2.2 值列表的补集

关键字 **complement** 表示列表中的值不能接受为输入值（即它是值列表的补集）。它可以用于所有类型的所有值。

列表中每个值的类型应与使用补集的模板字段声明的类型一致。当且仅当输入字段不匹配值列表中列出的任何值时，使用补集的模板字段才匹配对应的输入字段。当然，值列表可以是一个单个的值。

例如：

```
template Mymessage MyTemplate:=
{
  complement (1,3,5),  // 无法接受的整数值列表
  :
  field3 not(true)     // 将匹配false
  :
}
```

B.1.2.3 任何值

匹配符“?”（*AnyValue*）是指可以接受的任何有效的输入值，它可用于所有类型的值。当且仅当输入字段取指定类型的一个单个值时，使用任何值机制的模板字段才匹配对应的输入字段。

例如：

```
template Mymessage MyTemplate:=
{
  field1 := ?,    // 将匹配任何整数
  field2 := ?,    // 将匹配任何非空的字符串值
  field3 := ?,    // 将匹配true或false
  field4 := ?     // 将匹配任何整数序列
}
```

B.1.2.4 任何值或无

匹配符号“*”（*AnyValueOrNone*）是指可以接受的任何有效的输入值，包括省略该值。如果模板字段声明为可选的，那么它可用于所有类型的值。

当且仅当输入字段取值为指定类型的任何元素或输入字段不存在时，使用该符号的模板字段才匹配对应的输入字段。

例如：

```
template Mymessage MyTemplate:=
{ :
  field3 := *,    // 将匹配true、flase或被省略的字段
  :
}
```

B.1.2.5 值域

当用于 **integer** 或 **float** 类型（以及 integer 或 float 类型的子类型）时，值域指明可接受值的有界作用范围。边界值将为：

a） 无穷大或负无穷大；

b） 取值为一个特定整数值或浮点数值的表达式。

下边界将置于值域的左侧，上边界将置于值域的右侧，下边界将小于上边界。当且仅当输入字段值等于值域中的一个值时，使用值域的模板字段才匹配对应的输入字段。

当边界用在 **charstring** 或 **universal charstring** 类型的模板或模板字段中时，边界将根据类型的编码字符集表取有效的字符位置（如给定的位置不得为空）。上下边界之间的空位置被认为不是特定值域的有效值。

例如：

```
template Mymessage MyTemplate:=
{
  field1 := (1 .. 6),  // 整数类型的域
  :
  :
  :
}
// field1的其它条目可以是（-infinity 到 8）或（12 到 infinity)
```

B.1.2.6 超集

超集是一个只能用于 **set of** 类型值的匹配操作，超集用关键字 **superset** 来表示。当且仅当输入字段至少包含超集中定义的所有元素时，可以包含更多元素，超集才匹配对应的输入字段。超集的变元类型将与使用超集机制的字段中所声明的类型一致。

例如：

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// 不管次序和位置，只要整数序列中至少出现一次数字1、2和3，则匹配成功。
```

B.1.2.7 子集

子集是一个只能用于 **set of** 类型值的匹配操作，子集用关键字 **subset** 来表示。

当且仅当输入字段只包含子集中定义的元素时，可以包含更少元素，使用子集的字段才匹配对应的输入字段。子集的变元类型将与使用子集机制的字段中所声明的类型一致。

例如：

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// 不管次序和位置，只要整数序列中出现0次或一次数字1、2和3，则匹配成功。
```

B.1.3 值内部的匹配机制

B.1.3.0 概述

下面的匹配机制可以用在串、记录、record of、set、set of 和数组类型的明确值中。

B.1.3.1 任何元素

匹配符号“?”（*AnyElement*）表示该符号可以替代串（字符串除外）、**record of**、**set of** 或数组中的单个元素。它只能用于串类型、**record of** 类型、**set of** 类型或数组类型的值中。

例如：

```
template Mymessage MyTemplate:=
{ :
  field2 := "abcxyz",
  field3 := '10???'B,  // 每个“?”都可以是0或1
  field4 := {1, ?, 3}  // ?可以是任何整数值
}
```

注 — field4中的“?”可以被解释为作为整数值的*AnyValue*，或是**record of**、**set of**或数组中的*AnyElement*。由于这两种解释都将导致同样的匹配，因此不会引起任何问题。

B.1.3.1.1 使用单个字符通配符

如果要求在字符串中表达通配符“?”，那么它将使用字符样式来实现（见第 B.1.5 节）。例如，“abcdxyz”、“abccxyz”和“abcxxyz”等都与 **pattern**“abc?xyz”相匹配，但“abcxyz”和“abcdefxyz”等与 **pattern**“abc?xyz”不匹配。

B.1.3.2 任何个数的元素或无元素

匹配符“*”（*AnyElementsOrNone*）表示它可以替代串（字符串除外）、**record of**、**set of** 或数组类型的任意多个连续元素。“*”符号根据“*”前后符号所描述的样式来匹配尽可能长的元素序列。

例如：

```
template Mymessage MyTemplate:=
{ :
  field2 := "abcxyz",
  field3 := '10*11'B,  // “*”可以是任何比特序列（可以为空）
  field4 := {*, 2, 3}  // “*”可以是任何数目的整数或被省略
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

如果“*”出现在串、**record of**、**set of** 或数组内的最高层，那么它将被解释为 *AnyElementsOrNone*。

注 — 该规则用于防止其它可能的解释（将“*”解释为*AnyValueOrNone*），即替代串、**record of**、**set of**或数组内的一个元素。

B.1.3.2.1 使用多字符通配符

如果要求在字符串中表达通配符“*”，那么它将使用字符样式来实现（见第 B.1.5 节）。例如，“abcxyz”、“abcdefxyz”和“abcabcxyz”等都将与 **pattern**“abc*xyz”相匹配。

B.1.3.3 数列

数列是一种匹配操作，它只能用于 **record of** 类型的值。数列用关键字 **permutation** 表示。表达式、*AnyElement* 和 *AnyElementsOrNone* 都可以作为数列元素。数列中列出的每个元素的类型都应与 **record of** 类型相同。

代替单个元素的数列是指：只要它包含与数列中值列表相同的元素，虽然次序可能不同，这些元素串都是可接受的。如果值中同时使用了数列和 *AnyElementsOrNone*，那么它们应结合使用。

数列内部使用的 *AnyElementsOrNone* 可代替数列匹配的值记录段内的 0 个或多个元素。数列内部的 *AnyElementsOrNone* 将最后计算（当数列列表的所有其它元素都已匹配已计算列表中的一个元素时）。

注 1 — 数列内部使用的*AnyElementsOrNone*在以下两种情况下效果不同，一种情况是*AnyElementsOrNone*与数列结合使用，一种情况是 *AnyElementsOrNone* 只代替连续的元素。例如：{**permutation**(1,2,*)} 相当于 ({*,1,*,2,*},{*,2,*,1,*})，而{**permutation**(1,2),*}相当于 ({1,2},{2,1},*)。

注 2 — 当*AnyElementsOrNone*与数列结合使用时，对*AnyElementsOrNone*可能会用一个长度属性来限制*AnyElementsOrNone*匹配的元素个数（见第B.1.4.1节）。相反，对数列内部使用的*AnyElementsOrNone*不会施加任何长度属性（而对整个数列会用长度属性）。

例如：

```
type record of integer MySequenceOfType;

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
// 匹配下列任何一个4整数序列：1,2,3,5；1,3,2,5；2,1,3,5；2,3,1,5；3,1,2,5 或 3,2,1,5。

template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
// 匹配任何一个以5结尾、在其它位置上至少出现一次1和2的4整数序列。

template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
// 匹配任一个以1,2,3；1,3,2；2,1,3；2,3,1；3,1,2 或 3,2,1开头的整数序列。

template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
// 匹配任一个以1,2,3；1,3,2；2,1,3；2,3,1；3,1,2 或 3,2,1结尾的整数序列。

template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
// 匹配任一个在任意位置上包含1,2,3；1,3,2；2,1,3；2,3,1；3,1,2 或 3,2,1中任一子串的整数序列。

template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
// 匹配任一个以5结尾、在其它位置至少出现一次1和2的整数序列。
```

```
template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
// 匹配任一个以1,2,3；1,3,2；2,1,3；2,3,1；3,1,2 或 3,2,1开头、包含3～8个整数的序列。

template MySequenceOfType MyTemplate9 := { permutation ( 1, 2, *) length (3..5), 5 };
// 匹配任一个以5结尾、在其它位置至少出现一次1和2的、包含4～6个整数的序列。
```

B.1.4 值的匹配属性

B.1.4.0 概述

下面是与匹配机制相关的属性。

B.1.4.1 长度限制

长度限制属性用来限制串值的长度以及 **set of**、**record of** 或数组结构中的元素个数。它只能用作以下机制的一个属性：*AnyValue*、*AnyValueOrNone*、*AnyElement* 和 *AnyElementsOrNone*（但不在数列内）、数列、超集和子集。它也可以与补集匹配机制和 **ifpresent** 属性结合使用。**Length** 的语法见第 6.2.3 节和第 6.3.3 节。

注 — 当对模板或模板字段同时使用补集和长度限制匹配机制时，这两种限制应独立地应用于模板或模板字段。

对串值，要根据表 4 来解释长度单位。对 **set of**、**record of** 类型和数组，其长度单位是相同的类型。边界将由解析为特定的非负 **integer** 值的表达式来表示。可选地，对上边值可以使用关键字 **infinity** 作为一个值，以表示没有任何长度上限。

模板的长度说明不得与对应类型的长度限制（如果有的话）相矛盾。当且仅当输入字段同时匹配符号及其相关的属性时，使用长度作为符号一个属性的模板字段才与对应的输入字段相匹配。如果输入字段的长度大于等于指定的下限且小于等于指定的上限，那么长度属性匹配。仅有一个长度值的时候，只有接收到的字段的长度正好等于指定值的时候，长度属性才匹配。

长度限制允许与特殊值 **omit** 结合使用，不过在这种情况下，长度属性没有任何作用（即带上 **omit** 是多余的）。如果带 *AnyValueOrNone* 和 **ifpresent**，那么它对输入值施加了一个限制。

例如：

```
template Mymessage MyTemplate:=
{
  field1 := complement ({4,5},{1,4,8,9}) length (1 .. 6),
                        // 除{4,5} 和 {1,4,8,9}外，任一包含1个、2个、3个、4个、5个或6个元素的值都是可接受的。
  field2 := "ab*ab" length(13) // AnyElementsOrNone串的最大长度是9个字符。
  :
}
```

B.1.4.2 IfPresent指示器

如果存在一个可选的字段，那么 **ifpresent** 表示可以实现一个匹配（即不省略）。如果类型声明为可选的，那么该属性可以与所有的匹配机制一起使用。

当且仅当输入字段满足相关匹配机制或输入字段不存在时，使用 **ifpresent** 的模板字段才匹配对应的输入字段。

例如：

```
template Mymessage:MyTemplate:=
{ :
  field2 := "abcd" ifpresent, // 如果未被省略，那么匹配“abcd”。
  :
  :
}
```

注 — *AnyValueOrNone* 与**? ifpresent**具有完全相同的含义。

B.1.5 匹配字符样式

B.1.5.0 概述

字符样式可用在模板中以定义需要接收的字符串的格式。字符样式可用来匹配 **charstring** 和 **universal charstring** 值。除了文字字符外，字符样式允许使用元字符（例如，字符样式中的“?”和“*”分别表示匹配任何一个字符以及任何多个的任何字符。

例 1：

```
template charstring MyTemplate:= pattern "ab??xyz*0";
```

该模板将匹配任何由“ab”组成的字符串，后接任何两个字符，后接字符“xyz”，后接任何多个的任何字符（包括任何多个的“0”），最后以字符“0”结束

如果要求照字面来解释任何元字符，那么应在其前加上元字符“\”。

例 2：

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

该模板将匹配任何这样的字符串，即由字符“ab”组成，后接任何字符，后接字符“?xyz”，后接任意多个字符。

用于 TTCN-3 样式的元字符列表如表 B.1 所示。元字符不得包含空白，除非是在集合表达式之前或之内跟在换行符之后的空白。

表 B.1/Z.140－TTCN-3样式元字符列表

元 字 符	描 述
?	匹配任何字符（见注 1 和注 2）。
*	匹配任何字符零次或多次；将匹配尽可能多个数的字符（见上面例 1）（见注 1 和注 2）。
\	使其后的元字符解释为一个文字（见注 3）。当位于某字符前面而没有定义的元字符含义“\”且字符匹配“\”后的字符时（见注 4）。
[ ]	匹配指定集合内的任何字符（更多细节请参见第 B.1.5.1 节）。
-	仅在一对方括号（“[”和“]”）内才有元字符含义，方括号内的第一个和最后一个位置除外。允许指定字符范围（更多细节请参见第 B.1.5.1 节）。
^	仅在一对方括号（“[”和“]”）内“[”后的第一个字符时才有元字符含义，导致匹配该元字符后字符集补集中的任何字符（更多细节请参见第 B.1.5.1 节）。
\q{group,plane,row,cell}	匹配四元组所描述的通用字符。
{reference}	插入所引用的用户定义串，并将之解释为一个通常的表达式（更多细节请参见第 B.1.5.2 节）。
\N{reference}	匹配字符集内的任何字符，字符集通过所引用的定义进行定义(更多细节请参见 B.1.5.4 节)。
\d	匹配任何数字位（等价于[0-9]）。
\w	匹配任何字母数字的字符（等价于[0-9a-zA-Z]）。
\t	匹配 C0 控制字符 HT(9)（见 ISO/IEC 6429 [11]）。
\n	匹配以下任何 C0 控制字符：LF(10)、VT(11)、FF(12)、CR(13)（见 ISO/IEC 6429 [11]）（合称为换行字符）
\r	匹配 C0 控制字符 CR（见 ISO/IEC 6429 [11]）。
\s	匹配以下任何 C0 控制字符：HT(9)、LF(10)、VT(11)、FF(12)、CR(13)、SP(32)（见 ISO/IEC 6429 [11]、ITU-T T.50 建议书 [9]）（合称为空白符）。
\b	匹配一个字界（除 SP 和 DEL 之外的任何图形字符都会在其前或其后跟上空白或换行符）。
\"	匹配双引号符
""	匹配双引号符

表 B.1/Z.140－TTCN-3样式元字符列表

元 字 符	描 述
|	用来表示两个可选的表达式
( )	用于组合一个表达式
#(n, m)	匹配前面的表达式至少 n 次、但至多 m 次（后缀）（更多细节请参见第 B.1.5.3 节）。
#n	匹配前面的表达式正好 n 次（n 是一个单个的数字）（后缀）；等同于#(n)。
+	匹配前面的表达式一次或多次（后缀）；等同于#(1,)。
注 1 — 元字符?和*能匹配使用它们的模板或模板字段的源类型字符集的任何字符（即不考虑所用的类型约束）。不过，不能忘记接收操作在试图匹配接收消息之前需要检查接收到的消息。因此，不符合模板或模板字段的子类型规范要求的接收值永远不会去匹配。 注 2 — 在其它一些语言/记法中，?和*作为元字符时有不同的含义。不过，在 TTCN 中，这些字符还是像本表中所规定的那样按传统方式用于匹配。 注 3 — 因此，反斜号字符可以被一对其间不带空格的反斜号字符匹配（\\），例如，样式“\\d”将匹配串“\d”；左方括号和右方括号可分别被“\[”和“\]”匹配等。 注 4 — 不赞成元字符“\”的这种使用，因为之后会定义更多的元字符。	

B.1.5.1 集合表达式

用一对“[”和“]”括起来的字符列表匹配该列表中的任何单个字符。集合表达式用“[”、“]”符号划定界限。除了字符文字外，还可以用连字号“-”来指定字符范围。字符范围包括紧位于连字号之前的字符、紧位于连字号之后的字符以及字符编码在这两个边界字符编码之间的所有字符。列表内没有之前或之后字符的连字号“-”没有特殊含义。

也可以在左方括号之后，通过将补字号“^”作为第一个字符的方式，来将集合表达式表示为补集。补集操作优先级比字符范围优先级高。因此，紧跟在补字号“^”后的连字号“-”将作为文字字符来处理。

不允许有空列表和空列表补集。因此，紧跟在左方括号“[”后的右方括号“]”，或者紧跟在左方括号“[”后且其后紧跟右方括号“]”的补字号“^”，都将作为文字字符来处理。

除下面列出的情况外，列表内的所有元字符都将失去其特殊含义：

- “]”不在第一个位置，且不直接跟在第一个位置的“^”后；
- “-”不在列表的第一个和最后一个位置；
- “^”在列表的第一个位置，但其后不是紧跟着一个右方括号；
- “\”、“\d”、“\t”、“\w”、“\r”、“\n”、“\s” 和“\b”；
- \q{group,plane,row,cell}；
- **\N**{引用}。

注 1 — 不允许使用嵌入式列表（例如，在样式“[ab[r-z]]”中，第二个“[”表示文字“[”，第一个“]”结束列表，第二个“]”将引发一个错误，原因是样式中没有相关的左方括号）。

注 2 — 为包括文字补字号“**^**”，可将之置于除第一个位置外的其它任何地方，或者在其前面加上反斜号。为包括文字连字号“**-**”，可将之置于列表的第一个或最后一个位置，或者在其前面加上反斜号。为包括文字右方括号“**]**”，可将之置于列表的第一个位置，或者在其前面加上反斜号。如果列表中的第一个字符是补字号“**^**”，那么当字符“**-**”和“**]**”直接跟在补字号后时，也可以与其自身匹配。

例如：

```
template charstring RegExp1:= pattern '[a-z]';  // 这将匹配从a到z的任何字符。

template charstring RegExp2:= pattern '[^a-z]';  // 这将匹配除a到z之外的任何字符。

template charstring RegExp3:= pattern '[AC-E][0-9][0-9][0-9]YKE';

// RegExp3将匹配以A或C与E之间的字母（但不是如B）开头的、后接三个数字以及字母YKE的字符串。
```

B.1.5.2 引用表达式

除了直接的串值，也可以在样式中使用对已有模板、常量、变量或模板参数的引用。引用用“{”和“}”字符括起来，引用将解析为字符串类型之一。所引用模板、常量或变量的内容将作为普通表达式来处理。每个表达式只能被解除引用一次。

例如：

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern '{MyString}';
```

该模板将匹配由字符“ab”组成的任何字符串，后接任何字符。实际上，不管是显性还是通过引用方式跟在关键字 **pattern** 后的任何字符串都将按照本节中定义的规则进行解释。

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

该模板将匹配由字符“ab”组成的任何字符串，后接任何字符，后接字符“de”，后接 ISO/IEC 10646 中的字符，其组＝1，面＝1，行＝13，单元＝7。

如果引用表达式引用一个包含一个或多个引用表达式的模板、常量或变量，那么被引用模板、常量或变量中的引用将在把其内容插入引用样式之前递归地解除引用。

例如：

```
const charstring MyConst2 := pattern "ab";
template charstring RegExp1 := pattern "{MyConst2}";
// 匹配串"ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
// 匹配串"abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
// 匹配串"cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
   // 只匹配串"{RegExp1}"（即不能处理成对模板RegExp1的引用表达式）。
```

B.1.5.3 匹配表达式n次

为说明对前面的表达式应匹配多少次，将使用以下语法之一：“#(n, m)”、“#(n,)”、“#(, m)”、“#(n)”、“#n”或“+”。形式“#(n, m)”是指必须至少匹配前面的表达式 n 次，但又不能多于 m 次。元字符后缀“#(n,)”是指必须至少匹配前面的表达式 n 次，而“#(, m)”是指至多匹配前面的表达式 m 次。元字符后缀“#(n)”和“#n”是指必须正好匹配前面的表达式 n 次（等同于“#(n, n)”）。在形式“#n”中，n 是一个单个的数字。元字符后缀“+”是指必须至少匹配前面的表达式 1 次（等同于“#(1, 1)”）。

例如：

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)';  // 至少匹配9个但不大于11个、从a到z的字符。
template charstring RegExp5a:= pattern '[a-z]#(9)';     // 刚好匹配9个从a到z的字符。
template charstring RegExp5b:= pattern '[a-z]#9';       // 刚好匹配9个从a到z的字符。
template charstring RegExp6:= pattern '[a-z]#(9, )';    // 至少匹配9个从a到z的字符。
template charstring RegExp7:= pattern '[a-z]#(, 11)';   // 匹配不大于11个从a到z的字符。
template charstring RegExp8:= pattern '[a-z]+';         // 至少匹配1个从a到z的字符。
```

B.1.5.4 匹配一个引用的字符集

“\N{引用}”这种形式的记法（其中引用表示一个单字符长度的模板、常量、变量或模板参数）匹配所引用值或模板中的字符。

引用一个长度不为 1 的模板、常量、变量或模板参数会引发一个错误。

"\N{类型引用}**"**这种形式的记法（其中“类型引用”是对 **charstring** 或 **universal charstring** 类型的一个引用），将匹配所引用类型表示的字符集中的任何字符。

注 1 — 当所引用的字符集不是使用字符样式的模板或模板字段的类型定义允许的值真子集时，不能看作是一种错误（但是，例如，如果两个集合没有交集，那么永远不会发生匹配）。

注 2 — \N{**charstring**}等价于应用于**charstring**类型模板或模板字段的?，\N{**universal charstring**}等价于应用于**universal charstring**类型模板或模板字段的?（但如果将之用于**charstring**类型的模板或模板字段，那么会引发一个错误）。

例如：

```
type charstring MyCharRange ('a'..'z');
type charstring MyCharList ('a', 'z');
const MyCharRange myCharR := 'r';

template charstring myTempPatt1 := pattern '\N { myCharR }';
// myTempPatt1将只匹配串“r”。

template charstring myTempPatt2 := pattern '\N { MyCharRange }';
// myTempPatt2将匹配包含a～z单个字符的任何串。

template MyCharRange myTempPatt3 := pattern '\N { MyCharList }';
// myTempPatt3将只匹配串“a”和“r”。

template MyCharList myTempPatt4 := pattern '\N { MyCharRange }';
// myTempPatt4将只匹配串“a”和“r”。
```

B.1.5.5 有关样式的类型兼容性规则

对所引用的样式（见第 B.1.5.2 节）和所引用的字符集（见第 B.1.5.4 节），应用特殊的兼容性规则：**charstring** 类型的引用类型、模板、常量、变量或模块参数总是用在 **universal charstring** 类型的模板或模板字段的样式规格说明中；如果引用模板或值中使用的所有字符以及引用类型允许的字符集在 **charstring** 类型中都有对应的字符（见第 6.7.1 节中对应字符的定义），那么 **universal charstring** 类型的引用类型、模板或值可用于 **charstring** 类型的模板或模板字段的样式规格说明中。

附 件 C

TTCN-3预定义函数

本附件定义了 TTCN-3 预定义函数。

C.0 异常处理程序概述

本附件相关子句中没有显性定义异常处理规则的错误情况（例如，输入参数超出允许的范围、输入参数类型错误、输入值包含不正确的字符等）将引起一个 TTCN-3 编译时或运行时错误。哪种错误情况会引起编译时错以及哪种错误会引起运行时错与工具的执行选项有关。

C.1 integer到character

```
int2char(integer value) return charstring
```

该函数将一个 0～127（8 比特编码）之间的 **integer** 值转换为一个单字符长度的 **charstring** 值。该整数值是该字符的 8 比特编码。

C.2 character到integer

```
char2int(charstring value) return integer
```

该函数将一个单字符长度的 **charstring** 值转换为一个 0～127 之间的整数值。该整数值是该字符的 8 比特编码。

C.3 integer到universal character

```
int2unichar(integer value) return universal charstring
```

该函数将 0～2147483 647（32 位编码）间的一个 **integer** 值转换为一个单字符长度的 **universal charstring** 值。该整数值用于描述该字符的 32 位编码。

C.4 universal character到integer

```
unichar2int(universal charstring value) return integer
```

该函数将一个单字符长度的 **universal charstring** 值转换为一个 0～2147483647 之间的整数值。该整数值是该字符的 32 比特编码。

C.5 bitstring到integer

```
bit2int(bitstring value) return integer
```

该函数将一个单个的 **bitstring** 值转换为一个单个的 **integer** 值。

在该转换中，**bitstring** 将被解释为一个基数为 2 的正 **integer** 值。最右边的位为最低有效位，最左边的位为最高有效位，位 0 和 1 分别表示十进制的值 0 和 1。

C.6 hexstring到integer

```
hex2int(hexstring value) return integer
```

该函数将一个单个的 **hexstring** 值转换为一个单个的 **integer** 值。

在该转换中，**hexstring** 将被解释为一个基数为 16 的正 **integer** 值。最右边的十六进制位为最低有效位，最左边的十六进制位为最高有效位，十六进制数字 0～F 分别表示十进制的值 0～15。

C.7 octetstring到integer

```
oct2int(octetstring value) return integer
```

该函数将一个单个的 **octetstring** 值转换为一个单个的 **integer** 值。

在该转换中，**octetstring** 将被解释为一个基数为 16 的正 **integer** 值。最右边的十六进制位为最低有效位，最左边的十六进制位为最高有效位。由于一个八位字节由 2 个十六进制的数字组成，因此给出的十六进制数字的个数应是 2 的倍数。十六进制数字 0～F 分别表示十进制的值 0～15。

C.8 charstring到integer

```
str2int(charstring value) return integer
```

该函数将一个表示 **integer** 值的 **charstring** 值转换为一个等价的 **integer**。

例如：

```
str2int("66")     // 将返回integer值66。

str2int("-66")    // 将返回integer值-66。

str2int("abc")    // 将产生编译或测试用例错误。

str2int("0")      // 将返回integer值0。
```

C.9 integer到bitstring

```
int2bit(in integer value, in integer length) return bitstring
```

该函数将一个单个的 **integer** 值转换为一个单个的 **bitstring** 值。结果是长度为 length 的比特串。

在该转换中，**bitstring** 将被解释为一个基数为 2 的正 **integer** 值。最右边的位为最低有效位，最左边的位为最高有效位，比特 0 和 1 分别表示十进制的值 0 和 1。如果转换产生的值的长度小于 length 参数所指定的长度，那么应在该 **bitstring** 的左边添加 0。

C.10 integer到hexstring

```
int2hex(in integer value, in integer length) return hexstring
```

该函数将一个单个的 **integer** 值转换为一个单个的 **hexstring** 值。结果是长度为 length 的十六进制数字串。

在该转换中，**hexstring** 将被解释为一个基数为 16 的正 **integer** 值。最右边的十六进制位为最低有效位，最左边的十六进制位为最高有效位，十六进制数字 0～F 分别表示十进制的值 0～15。如果转换产生的值的长度小于 length 参数所指定的长度，那么应在该 **hexstring** 的左边添加 0。

C.11 integer到octetstring

```
int2oct(in integer value, in integer length) return octetstring
```

该函数将一个单个的 **integer** 值转换为一个单个的 **octetstring** 值。结果串是长度为 length 的八位字节。

在该转换中，**octetstring** 将被解释为一个基数为 16 的正 **integer** 值。最右边的十六进制位为最低有效位，最左边的十六进制位为最高有效位。由于一个八位字节由 2 个十六进制的数字组成，因此给出的十六进制数字的个数应是 2 的倍数。十六进制数字 0～F 分别表示十进制的值 0～15。如果转换产生的值的长度小于 length 参数所指定的长度，那么应在该 **hexstring** 的左边添加 0。

C.12 integer到charstring

```
int2str(integer value) return charstring
```

该函数将一个整数值转换为与其等价的串（该返回串的基数总为十进制）。

例如：

```
int2str(66)    // 将返回charstring值"66"。
int2str(-66)   // 将返回charstring值"-66"。
int2str(0)     // 将返回charstring值"0"。
```

C.13 串长度

```
lengthof(any_string_type value) return integer
```

该函数返回 **bitstring**、**hexstring**、**octetstring** 或任何字符串类型值的长度，每个串类型的长度单位在表 4 中定义。

universal charstring 的长度将通过对每个合成字符和合成音节字符（包括填充符）的计数来计算（见 ISO/IEC 10646[10]，第 23 节和第 24 节）。

例如：

```
lengthof('010'B)  // 返回3
lengthof('F3'H)  // 返回2
lengthof('F2'O)  // 返回1
lengthof (universal charstring : "Length_of_Example")  // 返回17
```

C.14 结构化值中的元素个数

```
sizeof(any_type value) return integer
```

该函数返回 **record**、**record of**、**set**、**set of** 类型或数组的模块参数、常量、变量或 **template** 的实际元素个数（见注）。对 **record of** 和 **set of** 类型的值、模板或数组，返回的实际值是最后定义元素的序列号（该元素的下标值加 1）。

注 — 只对作为函数参数的TTCN-3对象的元素才做计算，也就是说，任何嵌套类型/值的元素在确定返回值时都不予考虑。

例如：

```
// 假定
type record MyPDU
  {   boolean field1  optional,
      integer field2
  };

template MyPDU  MyTemplate
  { field1  omit,
    field2 5
  };

var integer numElements;

// 则

numElements := sizeof(MyTemplate); // 返回1

// 假定
type record length(0..10) of integer MyList;
var MyList MyRecordVar;
MyRecordVar := { 0, 1, omit, 2, omit };


// 则
numElements := sizeof(MyRecordVar);
//返回4，不考虑元素MyRecordVar[2]未定义这一事实。
```

C.15 IsPresent函数

```
ispresent(any_type value) return boolean
```

该函数只适用于 **record** 和 **set** 类型，当且仅当所引用字段的值在所引用数据对象的实际实例中出现时，该函数才返回 **true**。**ispresent** 的变元将是一个对 **record** 或 **set** 类型的字段的引用。

```
// 假定
type record MyRecord
  {   boolean  field1 optional,
      integer field2
  }
// 并假定MyPDU是一个MyRecord类型的模板，且received_PDU也是MyRecord类型，那么
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// 如果MyPDU实际实例中的field1出现，那么返回true。
```

C.16 IsChosen函数

```
ischosen(any_type value) return boolean
```

当且仅当数据对象引用描述了为给定数据对象实际选中的 **union** 类型的变体，该函数才返回值 **true**。

例如：

```
// 假定
type union MyUnion
  {   PDU_type1   p1,
      PDU_type2   p2,
      PDU_type p3
  }

// 并假定MyPDU是一个MyUnion类型的模板，且received_PDU也是MyUnion类型，那么
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// 如果实际的MyPDU实例承载了一个PDU_type2类型的PDU，那么返回true。
```

C.17 regexp函数

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
character_string_type
```

该函数返回输入字符串 instr 的子串，它是对 expression 第 n 组匹配的内容。输入串 instr 可以是任何字符串类型。返回的字符串类型是 instr 的源类型。表达式为第 B.1.5 节中所描述的字符样式。待返回的组编号由 groupno 指定，它将是一个正整数。组编号根据它们在组的括号中出现的次序进行赋值，从 0 开始开始计，步长为 1。如果在输入串中没有任何子串满足所有的条件（即样式和组编号），那么返回空串。

例如：

```
// 假定
var charstring mypattern2 := "
var charstring  myinput := '      date: 2001-10-20 ;  msgno: 17; exp  '
var charstring mypattern := '[ /t]#(,)date:[ \d\-]#(,);[ /t]#(,)msgno: (\d#(1,3)); [exp]#(0,1)'

// 则表达式为：
var charstring mystring := regexp(myinput, mypattern,1)
//将返回值'17'。
```

C.18 bitstring到charstring

```
bit2str (bitstring value) return charstring
```

该函数将一个单个的 **bitstring** 值转换为一个单个的 **charstring** 值。结果 **charstring** 的长度与 **bitstring** 的长度相同，并只包含字符 '0' 和 '1'。

在该转换中，**bitstring** 被转换为一个 **charstring**。**bitstring** 的每一位根据其值为 0 或为 1，分别转换为字符 '0' 或字符 '1'。结果 **charstring** 中的字符连续次序与 **bitstring** 中的位次序相同。

例如：

```
bit2str ('1110101'B) will return "1110101"
```

C.19 hexstring到charstring

```
hex2str (hexstring value) return charstring
```

该函数将一个单个的 **hexstring** 值转换为一个 **charstring** 值。结果 **charstring** 的长度与 **hexstring** 的长度相同，且仅包含字符 '0' ～ '9' 和 'A' ～ 'F' 。

在该转换中，一个 **hexstring** 应被转换为一个 **charstring**。**hexstring** 的每一个十六进制数字将根据其值为 0～9 或为 A～F 的十六进制数字，分别转换为一个 '0' ～ '9' 或 'A' ～ 'F' 的字符。结果 **charstring** 中的字符连续次序与 **hexstring** 中的数字次序相同。

例如：

```
hex2str ('AB801'H) will return "AB801"
```

C.20 octetstring到charstring

```
oct2str (octetstring invalue) return charstring
```

该函数将一个 **octetstring** 类型的 invalue 转换为一个 **charstring**，它代表一个等价于输入值的串。结果 **charstring** 的长度将与输入 **octetstring** 的长度相同。

在该转换中，invalue 的每个十六进制数字将转换为字符 '0' 、 '1' 、 '2' 、 '3' 、 '4' 、 '5' 、 '6' 、 '7' 、 '8' 、 '9' 、'A' 、'B' 、'C' 、'D' 、'E' 或 'F' ，对应十六进制数字的值。结果 **charstring** 中的字符连续次序与 **octetstring** 中的十六进制数字次序相同。

例如：

```
oct2str ('4469707379'O) = "4469707379"
```

C.21 charstring到octetstring

```
str2oct (charstring invalue) return octetstring
```

该函数将一个 **charstring** 类型的串转换为一个 **octetstring**。串 invalue 将包含偶数个字符，且只能是 '0' 、 '1' 、 '2' 、 '3' 、 '4' 、 '5' 、 '6' 、 '7' 、 '8' 、 '9' 、 'a' 、 'b' 、 'c' 、 'd' 、 'e' 、 'f' 、 'A' 、 'B' 、 'C' 、 'D' 、 'E' 或 'F' 这些图形字符中的一个。结果 **octetstring** 的长度将与输入 **charstring** 的长度相同。

例如：

```
str2oct ("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
```

C.22 bitstring到hexstring

```
bit2hex (bitstring value) return hexstring
```

该函数将一个单个的**bitstring**值转换为一个单个的**hexstring**值。结果**hexstring**表示与**bitstring**相同的值。

在该转换中，**bitstring** 将被转换为 **hexstring**。转换时，**bitstring** 从最右边开始分组，每四位分成一组。每个四位组转换为一个十六进制数字，如下所示：

'0000'B → '0'H, '0001'B → '1'H, '0010'B → '2'H, '0011'B → '3'H, '0100'B → '4'H, '0101'B → '5'H,
'0110'B → '6'H, '0111'B → '7'H, '1000'B → '8'H, '1001'B → '9'H, '1010'B → 'A'H, '1011'B → 'B'H,
'1100'B → 'C'H, '1101'B → 'D'H,'1110'B → 'E'H, and '1111'B → 'F'H.

当最左边的比特组不足4位时，则从左边开始填充 '0' B，制导正好满4位，然后对它进行转换。结果 **hexstring** 中，十六进制数字的连续次序与 **bitstring** 中 4 位分组的次序相同。

例如：

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 hexstring到octetstring

```
hex2oct (hexstring value) return octetstring
```

该函数将一个单个的 **hexstring** 值转换为一个 **octetstring** 值。结果 **octetstring** 表示与 **hexstring** 相同的值。

在该转换中，**hexstring** 将被转换为 **octetstring**。转换时，如果 **hexstring** 的长度模 2 为 0，那么 **octetstring** 包含与 **hexstring** 相同的十六进制数字序列。否则，结果 **octetstring** 包含 0，作为最左边十六进制数字 0，后跟与 **hexstring** 中相同的十六进制数字序列。

例如：

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 bitstring到octetstring

```
bit2oct (bitstring value) return octetstring
```

该函数将一个单个的 **bitstring** 值转换为一个单个的 **octetstring** 值。结果 **octetstring** 表示与 **bitstring** 相同的值。

转换遵循以下规则：bit2oct(value)=hex2oct(bit2hex(value))。

例如：

```
bit2oct ('111010111'B)= '01D7'O
```

C.25 hexstring到bitstring

```
hex2bit (hexstring value) return bitstring
```

该函数将一个单个的 **hexstring** 值转换为一个单个的 **bitstring** 值。结果 **bitstring** 表示与 **hexstring** 相同的值。

在该转换中，**hexstring** 被转换为 **bitstring**，转换时，**hexstring** 的十六进制数字以比特组的形式进行转换，如下所示：

'0'H → '0000'B, '1'H → '0001'B, '2'H → '0010'B, '3'H → '0011'B, '4'H → '0100'B, '5'H → '0101'B,
'6'H → '0110'B, '7'H → '0111'B, '8'H → '1000'B, '9'H → '1001'B, 'A'H → '1010'B, 'B'H → '1011'B,
'C'H → '1100'B, 'D'H → '1101'B,'E'H → '1110'B, and 'F'H → '1111'B.

结果 **bitstring** 中，4 比特组的连续次序与 **hexstring** 中的十六进制数字次序相同。

例如：

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 octetstring到hexstring

```
oct2hex (octetstring value) return hexstring
```

该函数将一个单个的 **octetstring** 值转换为一个单个的 **hexstring** 值。结果 **hexstring** 表示与 **octetstring** 相同的值。

在该转换中，一个 **octetstring** 被转换为一个 **hexstring**，它包含与 **octetstring** 相同的十六进制数字序列。

例如：

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 octetstring到bitstring

```
oct2bit (octetstring value) return bitstring
```

该函数将一个 **octetstring** 值转换为一个 **bitstring** 值。结果 **bitstring** 表示与 **octetstring** 相同的值。

转换遵循以下规则：oct2bit(value)=hex2bit(oct2hex(value))。

例如：

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.28 integer到float

```
int2float (integer value) return float
```

该函数将一个 **integer** 值转换为一个 **float** 值。

例如：

```
int2float(4) = 4.0
```

C.29 float到integer

```
float2int (float value) return integer
```

该函数将一个 **float** 值转换为一个 **integer** 值，它删除变元的小数部分并返回结果 **integer**。

例如：

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 随机数生成函数

```
rnd ([float seed]) return float
```

rnd 函数返回一个小于 1 但大于等于 0 的（伪）随机数。通过一个可选种子值的方式来初始化随机数产生器。此后，如果没有提供任何新的种子，最后产生的数将用作下一个随机数的种子。第一次使用 **rnd** 时，如果之前未进行初始化，那么将使用由系统时间计算而来的一个值作为种子。

注 — 如果**rnd**函数每次使用相同的种子值来初始化，那么它会重复产生相同的随机数序列。

可以使用下面公式来产生一个给定范围内的随机整数：

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
// 此处，upperbound和lowerbound表示范围的最大值和最小值。
```

C.31 子串函数

```
substr (any_string_type value, in integer index, in integer returncount) return input_string_type
```

该函数从一个 **bitstring**、**hexstring**、**octetstring** 或任何字符串类型的值返回一个子串。该子串的类型为输入值的源类型，通过第二个 in 参数（下标）定义返回子串的起始点，下标从 0 开始。第三个输入参数定义待返回子串的长度，长度单位在表 4 中定义。

例如：

```
substr ('00100110'B, 3, 4)      // 返回'0011'B。
substr ('ABCDEF'H, 2, 3)        // 返回'CDE'H。
substr ('01AB23CD'O, 1, 2)      // 返回'AB23'O。
substr ("My name is JJ", 11, 2) // 返回"JJ"。
```

C.32 结构化类型中的元素个数

```
sizeoftype(any_type value) return integer
```

该函数返回 **record of** 类型、**set of** 类型或数组类型的模块参数、常量、变量或 **template** 的声明元素数（见注）。该函数适用于带长度限制的类型值。返回的实际个数是最后一个元素的后续编号，而并不考虑其值定义与否（即该函数参数基于的类型定义的最大下标加 1）。

注 — 只对作为函数参数的TTCN-3对象的元素才做计算，也就是说，对嵌套类型/值的任何元素，在确定返回值时都不予考虑。

例如：

```
// 假定
type record of integer MyPDU1;
type set length(1..8) of integer MyPDU2;
type record length(10) of integer MyPDU3;

var MyPDU1 MyRecordOfVar1;
var MyPDU2 MyRecordOfVar2;
var MyPDU3 MyRecordOfVar3;

var integer numElements;

// 则
numElements := sizeoftype(MyRecordOfVar1); // 由于未约束MyPDU1，返回错误。
numElements := sizeoftype(MyRecordOfVar2); // 返回8。
numElements := sizeoftype(MyRecordOfVar3); // 返回10。
```

C.33 charstring到float

```
str2float (charstring value) return float
```

该函数将一个由浮点数组成的 **charstring** 值转换为 **float** 值。除以下几种例外情况，**charstring** 中数字的格式将遵循第 6.1.0 节中的规则：

- 允许有引导0；
- 允许在正数前加引导符号“+”；
- “-0.0”是允许。

例如：

```
str2float('12345.6') //  等同于str2float('123.456E+02')
```

C.34 replace函数

```
replace (in any_string_type str, in integer ind, in integer len, in any_string_type repl)
return any_string_type
```

该函数用串值 **repl** 来替换 **str** 值中第 **ind** 位开始、长度为 **len** 的子串，并返回结果串。**str** 不能被修改。如果 **len** 等于 **0**，那么插入串 **repl**。如果 **ind** 等于 **0**，那么在 **str** 的起始位置插入 **repl**。如果 **ind** 等于 **lengthof(str)**，那么在 **str** 尾部插入 **repl**。**str** 和 **repl** 应具有相同的串类型，并且是基本类型 **bitstring**、**hexstring**、**octetstring** 或任何字符串。返回串的类型与 **str** 和 **repl** 相同。注意，串中的下标从 **0** 开始。

以下错误情况将在编译或运行时导致一个错误：

- **str**或**repl**不是串类型；
- **str**或**repl**是不同类型；
- **ind**小于**0**或大于**lengthof(str)**；
- **len**小于**0**或大于**lengthof(str)**；
- **ind+len**大于**lengthof(str)**。

例如：

```
replace ('00000110'B, 1, 3, '111'B)     // 返回 '01110110'B。
replace ('ABCDEF'H, 0, 2, '123'H)       // 返回 '123CDEF'H。
replace ('01AB23CD'O, 2, 1, 'FF96'O)    // 返回 '01ABFF96CD'O。
replace ("My name is JJ", 11, 1, "xx")  // 返回 "My name is xxJ"。
replace ("My name is JJ", 11, 0, "xx")  // 返回 "My name is xxJJ"。
replace ("My name is JJ", 2, 2, "x")    // 返回 "Myxame is JJ"。
replace ("My name is JJ", 12, 2, "xx")  // 返回测试用例错误。
replace ("My name is JJ", 13, 2, "xx")  // 返回测试用例错误。
replace ("My name is JJ", 13, 0, "xx")  // 返回 "My name is JJxx"。
```

C.35 octetstring到charstring

```
oct2char (octetstring invalue) return charstring
```

该函数将一个 **octetstring** 类型值 invalue 转换为 **charstring** 类型。输入参数 invalue 中不能包含大于 7F 的八位字节。结果 **charstring** 与输入 **octetstring** 的长度相同。八位字节解释为 ITU-T T.50 建议书[9]编码（根据 IRV），结果字符添加至返回值中。

例如：

```
oct2char ('4469707379'O) = "Dipsy"
```

注 — 返回的字符串可以包含非图形字符，它们不能出现在双引号之间。

C.36 charstring到octetstring

```
char2oct (charstring invalue) return octetstring
```

该函数将一个 **charstring** 类型的 invalue 值转换为 **octetstring** 类型。结果 **octetstring** 类型值中的每个八位字节将包含 invalue 对应字符的 ITU-T T.50 建议书[9]编码（根据 IRV）。

例如：

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

附件 D（资料性的）

空缺

注 — 本附件的内容已移至ITU-T Z.146建议书中 [6]。

附件 E（资料性的）

可用类型库

E.1 限制

加至本库的类型名称在整个语言和该库中应是唯一的（即不得是附件 C 中定义的名称之一）。TTCN-3 用户不得将本库中定义的名称用作本附件中给出的定义之外的其它定义的标识符。

注 — 因此，本附件中给出的类型定义可以在TTCN-3模块中重复，但未在本附件中定义的类型不可使用本附件中用到的某个标识符来定义它。

E.2 可用的TTCN-3类型

E.2.1 可用的简单基本类型

E.2.1.0 带符号的和不带符号的单字节整数

这些类型支持范围在–128～127 的带符号整数值和 0～255 的不带符号整数值，用于这些类型的值记法与用于整数类型的值记法相同。这些类型的值以其在系统内表示的单字节形式进行编码和解码，独立于实际使用的表示形式。

注 — 根据实际使用的编码规则，这些类型值的编码可以相同，也可以互不相同，并可以与整数类型（这些有用类型的源类型）的编码不同。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type integer   byte     (-128 .. 127)     with { variant "8 bit" };

type integer   unsignedbyte (0 .. 255)    with { variant "unsigned 8 bit" };
```

E.2.1.1 带符号的和不带符号的短整数

这些类型支持范围为–32768～32767 的带符号整数值和 0～65535 的不带符号整数值，用于这些类型的值记法与用于整数类型的值记法相同。这些类型的值以其在系统内表示的双字节形式进行编码和解码，独立于实际使用的表示形式。

注 — 根据实际使用的编码规则，这些类型值的编码可以相同，也可以互不相同，并可以与整数类型（这些有用类型的源类型）的编码不同。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type integer   short        (-32768 .. 32767)   with { variant "16 bit" };

type integer   unsignedshort  (0 .. 65535)      with { variant "unsigned 16 bit" };
```

E.2.1.2 带符号的和不带符号的长整数

这些类型支持范围为–2147483648～2147483647 的带符号整数值和 0～4294967295 的不带符号整数值，用于这些类型的值记法与用于整数类型的值记法相同。这些类型的值以其在系统内表示的四字节形式进行编码和解码，独立于实际使用的表示形式。

注 — 根据实际使用的编码规则，这些类型值的编码可以相同，也可以互不相同，并可以与整数类型（这些有用类型的源类型）的编码不同。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type integer   long(-2147483648 .. 2147483647)
               with { variant "32 bit" };

type integer   unsignedlong (0 .. 4294967295)
               with { variant "unsigned 32 bit" };
```

E.2.1.3 带符号的和不带符号的特长整数

这些类型支持范围为–9223372036854775808 ～ 9223372036854775807 的带符号整数值和 0 ～ 18446744073709551615 的不带符号整数值，用于这些类型的值记法与用于整数类型的值记法相同。这些类型的值以其在系统内表示的八字节形式进行编码和解码，独立于实际使用的表示形式。

注 — 根据实际使用的编码规则，这些类型值的编码可以相同，也可以互不相同，并可以与整数类型（这些有用类型的源类型）的编码不同。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type integer   longlong(-9223372036854775808 .. 9223372036854775807)
               with { variant "64 bit" };

type integer   unsignedlonglong(0 .. 18446744073709551615)
               with { variant "unsigned 64 bit" };
```

E.2.1.4 IEEE 754浮点数

对二进制浮点运算，这些类型支持 ANSI/IEEE 标准 754（见参考文献）。IEEE 754 的浮点数类型支持基数为 10、8 位指数、23 位尾数和 1 个符号位的浮点数，IEEE 754 的双精度类型支持基数为 10、11 位指数、52 位尾数和 1 个符号位的浮点数，IEEE 754 的 `extfloat` 类型支持基数为 10、最小 11 位指数、最小 32 位尾数和 1 个符号位的浮点数，IEEE 754 的 `extdouble` 类型支持基数为 10、最小 15 位指数、最小 64 位尾数和 1 个符号位的浮点数。

这些类型的值应根据 IEEE 754 的定义进行编码和解码，用于这些类型的值记法与浮点数类型（基数为 10）的值记法相同。

注 — 该类型值的精确编码依赖于所用的实际编码规则。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type  float    IEEE754float     with { variant "IEEE754 float" };

type  float    IEEE754double    with { variant "IEEE754 double" };

type  float    IEEE754extfloat  with { variant "IEEE754 extended float" };

type  float    IEEE754extdouble     with { variant "IEEE754 extended double" };
```

E.2.2 可用的字符串类型

E.2.2.0 UTF-8字符串“utf8string”

该类型支持 TTCN-3 **universal charstring** 类型（见第 6.1.1 节的段落 d）的所有字符集。其不同值为来自该字符集的 0 个、1 个或多个字符。应根据 ISO/IEC 10646 [10]附件 R 中定义的 UCS 转换格式 8（UTF-8）来对该类型的值进行整体编码和解码(如对值中的每一个字符逐个地进行编码和解码)。该类型的值记法与 **universal charstring** 类型的值记法相同。

有关该类型的类型定义为：

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 BMP字符串“bmpstring”

该类型支持 ISO/IEC 10646 [10]的基本多语言平面（BMP）字符集。BMP 代表通用多八位字节编码字符集中组 00 面 00 的所有字符。应根据 UCS-2 编码的转换格式（见 ISO/IEC 10646[10]的第 14.1 节）来对该类型的值进行整体编码和解码（如对值中的每一个字符逐个地进行编码和解码）。该类型的值记法与 **universal charstring** 类型的值记法相同。

注 — 类型“bmpstring”支持TTCN-3 **universal charstring**类型的子集。

有关该类型的类型定义为：

```
type universal charstring bmpstring( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
                         with { variant "UCS-2" };
```

E.2.2.2 UTF-16字符串“utf16string”

该类型支持通用多八位字节编码字符集（见 ISO/IEC 10646 [10]）组 00 面 00 到面 16 的所有字符。其不同值为来自该字符集的 0 个、1 个或多个字符。应根据 ISO/IEC 10646 [10]附件 Q 中定义的 UCS 转换格式 16（UTF-16）来对该类型的值进行整体编码和解码（如对值中的每一个字符逐个地进行编码和解码）。该类型的值记法与 **universal charstring** 类型的值记法相同。

注 — 类型“ utf16string”支持TTCN-3 **universal charstring**类型的子集。

有关该类型的类型定义为：

```
type universal charstring utf16string   ( char ( 0,0,0,0 ) .. char ( 0,16,255,255) )
                        with { variant "UTF-16" };
```

E.2.2.3 ISO/IEC 8859字符串“iso8859string”

该类型支持多方标准ISO/IEC 8859 中定义的所有字母表中的所有字符(见参考文献)。其不同值为来自 ISO/IEC 8859 字符集的 0 个、1 个或多个字符。应根据 ISO/IEC 8859 中规定的编码表示形式（8 比特编码）来对该类型的值进行整体编码和解码（如对值中的每一个字符逐个地进行编码和解码）。该类型的值记法与 **universal charstring** 类型的值记法相同。

注 1 — 类型“iso8859string”支持TTCN-3 **universal charstring**类型的子集。

注 2 — 在每个ISO/IEC 8859字母表中，字符集表中的小写部分（位置从02/00到07/14）兼容于ITU-T T.50建议书[9]字符集。因此，所有附加语言特殊字符仅为该字符表的大写部分（位置从10/00到15/15）定义。由于“iso8859string”类型是定义为TTCN-3 universal charstring类型的一个子集，因此，任何ISO/IEC 8859字母表的任何编码字符表示形式都可以映射到一个来自ISO/IEC 10646 [10]基本拉丁文或Latin-1补充字符表的等价字符（当按8比特编码时，带有相同编码表示形式的一个字符）。

有关该类型的类型定义为：

```
type universal charstring iso8859string ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                        with { variant "8 bit" };
```

E.2.3 可用的结构化类型

E.2.3.0 定点十进制文字

该类型支持 IDL 语法和语义版本 2.6（见参考文献）中定义的定点十进制数字文字的使用。它由整数部分、小数点和小数部分来描述。整数和小数部分都是由十进制（基数为 10）的数字串组成。数字的位数保存在“digits”中，小数部分的长度由“scale”给出，数字本身保存在“value_”中。该类型的值记法与记录类型的值记法相同，该类型的值应编码和解码为 IDL 定点十进制值。

注 — 该类型值的精确编码依赖于实际使用的编码规则。编码规则的详细内容超出了本建议书的讨论范围。

有关该类型的类型定义为：

```
type record IDLfixed {
               unsignedshort  digits,
               short    scale,
               charstring   value_
                  }
            with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 可用的原子串类型

E.2.4.1 单个IRV字符类型

该类型的不同取值是 ITU-T T.50 建议书 [9]版本的单个字符，符合国际参考版本（IRV）的要求，如第 8.2 节/T.50 [9]中所规定的那样（也可参见第 6.1.1 节的注 2）。

有关该类型的类型定义为：

```
type charstring char length (1);
```

注 1 — 该可用类型的名称与用于表示四比特形式的**universal charstring**值的TTCN-3关键字相同。通常，TTCN-3关键字不允许用作标识符。“char”有用类型是唯一的例外，只是为了与之前版本的TTCN-3标准向后兼容。

注 2 — 第28.2.3节中定义的特殊串“8 bit”可与该类型一起使用，为其值指定一个给定的编码。此外，基本类型的其它特性也可通过使用属性机制来改变。

E.2.4.2 单个通用字符类型

该类型的不同取值是来自 ISO/IEC 10646 [10]的单个字符。

有关该类型的类型定义为：

```
type universal charstring uchar length (1);
```

注 — 除“8 bit”外，第28.2.3节中定义的特殊串都可以与该类型一起使用，为其值指定一个给定的编码。此外，基本类型的其它特性也可通过使用属性机制来改变。

E.2.4.3 单个比特类型

该类型的不同取值是单个的二进制数字。

有关该类型的类型定义为：

```
type bitstring bit length (1);
```

E.2.4.4 单个十六进制类型

该类型的不同取值是单个的十六进制数字。

有关该类型的类型定义为：

```
type hexstring hex length (1);
```

E.2.4.5 单个八位字节类型

该类型的不同取值是一对十六进制数字。

有关该类型的类型定义为：

```
type octetstring octet length (1);
```

附件 F（资料性的）

TTCN-3活动对象上的操作

F.1 概述

本附件简单叙述 TTCN-3 中活动对象上的操作语义，活动对象包括测试部件、定时器和端口。这里用状态机的形式来描写动态行为，状态机中：

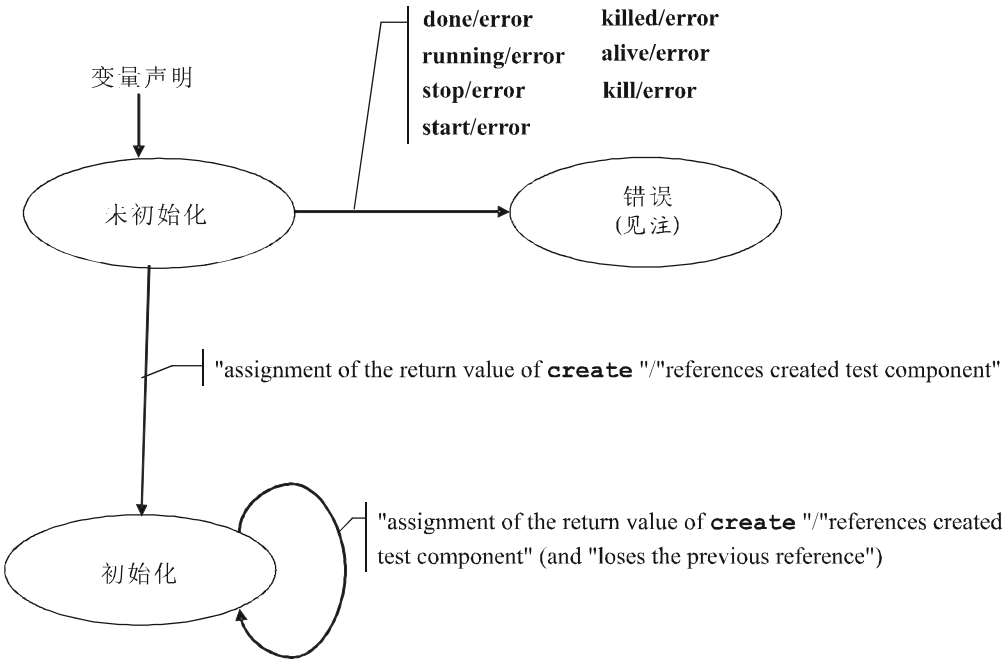
- 状态都经命名，并标识为节点；
- 初始状态用输入箭头来表示；
- 两个状态之间的转换（不必是两个不同的状态），并标识为箭头；
- 转换以该转换的使能条件（即操作或语句调用）和结果状态（如测试用例错误）来标记，二者之间用“/”分隔：
  - — 操作和语句调用指的是适用于对象的TTCN-3操作和语句（以黑体表示 ）；
  - — 作为结果条件的错误指的是测试用例错误（以黑体表示）；
  - — 作为结果条件的空指的是除了一种可能的状态变化之外，没有任何其它适用的结果（以黑体表示）；
  - — 匹配/不匹配指的是转换的匹配结果（以黑体表示）；
  - — 具体值为布尔结果或浮点数结果（以黑体表示）；
  - — 所有其它结果条件用纯文本描述（以标准字体表示）；
- 注释用来进一步详细地解释状态机。

更详细的描述请参考 TTCN-3 [3]的操作语义。当本附件与 TTCN-3 [3]的语义之间存在任何矛盾时，以后者为准。

F.2 测试部件

F.2.1 测试部件引用

测试部件类型的变量、**self** 和 **mtc** 操作用来引用测试部件。**start**、**stop**、**done** 和 **running** 操作并不直接用在测试部件上，但用在部件引用上。测试系统将决定请求的操作是否会影响部件对象本身，或其它动作是否合适(例如，在一个部件 **start** 操作中使用一个已停止 PTC 引用时会发生一个错误)。用于创建 PTC 的 **create** 操作会返回一个对已创建 PTC 的唯一引用，典型地，它绑定于一个测试部件变量。与测试部件变量自身相关的行为如图 F.1 所示：

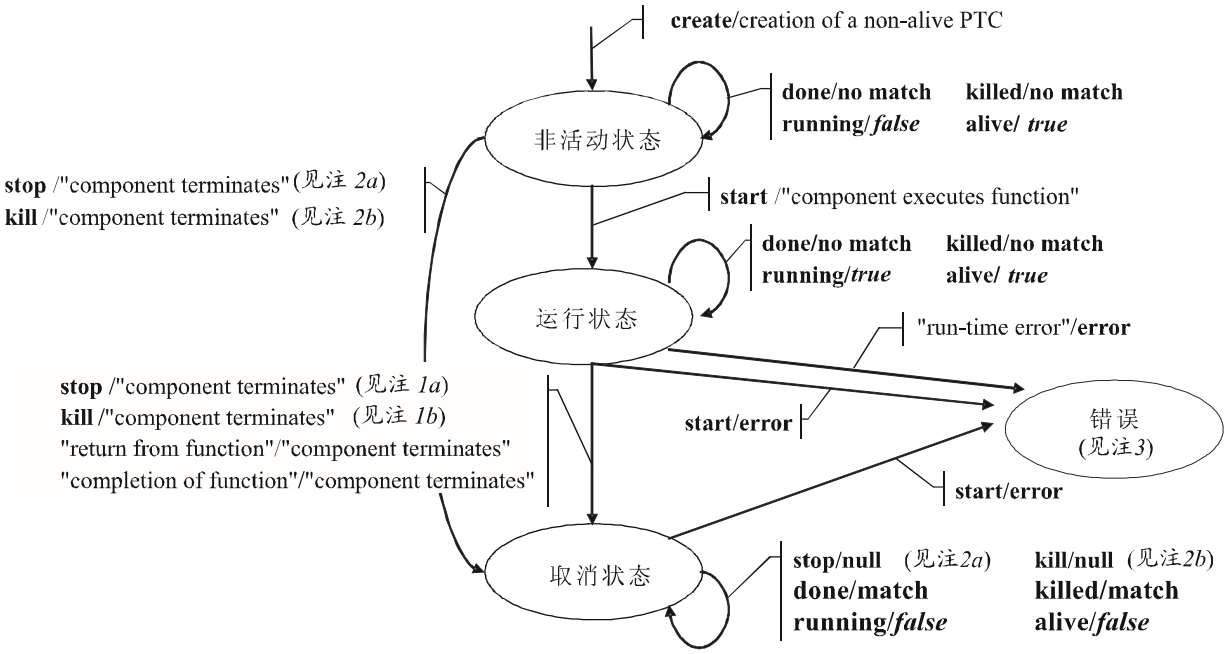


注－无论何时测试部件进入错误状态，为其局部判定指派局部错误判定，测试用例终止，总的测试用例结果将是一个错误。

图 F.1/Z.140－测试部件引用的处理

F.2.2 PTC的动态行为

PTC 可以是非活动类型，也可以是活动类型。非活动类型 PTC 可以是非活动状态、运行状态和取消状态。其动态行为如图 F.2 所示：



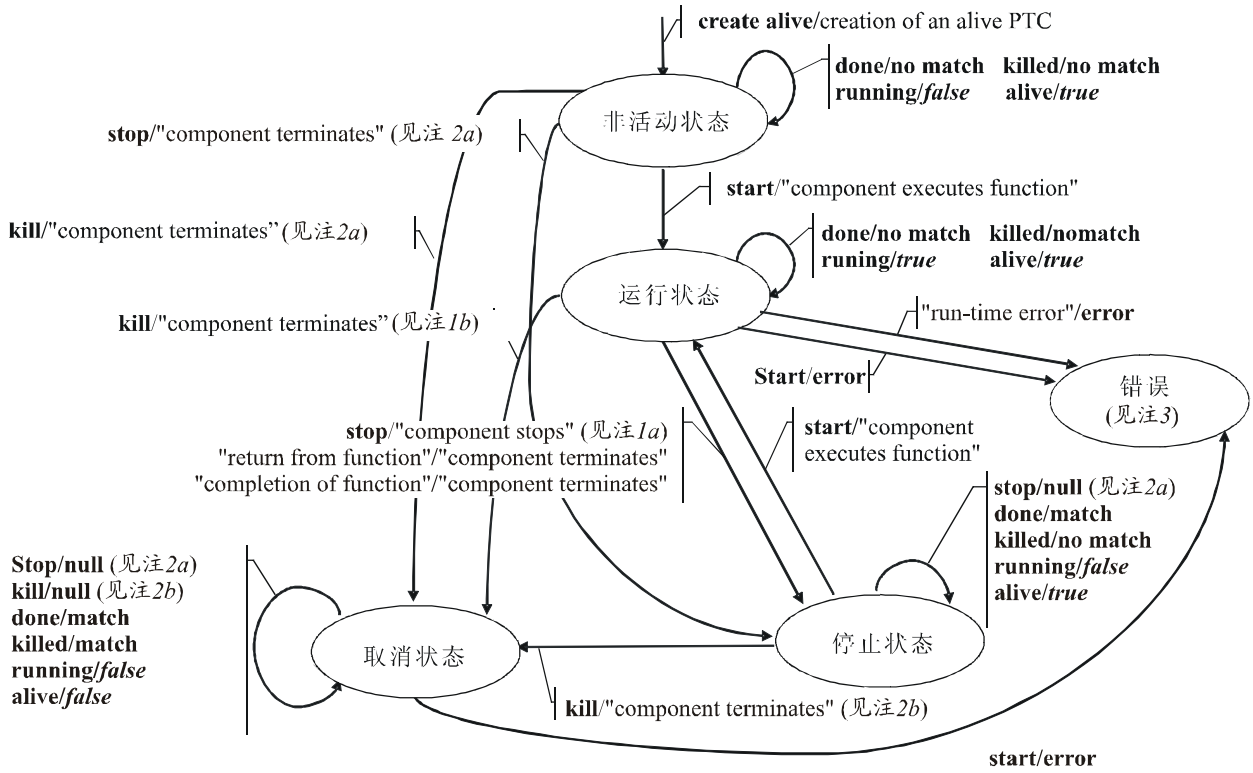
注1 — a）stop可以是一个stop、self.stop或是一个来自其它测试部件的stop；
b）kill可以是一个kill、self.kill、一个来自其它测试部件的kill，或是一个来测试系统的kill（在发生错误的情况下）。

注2 — a）stop只能来自另一个测试部件；
b）kill只能来自另一个测试部件或来自测试系统（在发生错误的情况下）。

注3 — 无论何时测试部件进入错误状态，为其局部判定指派局部错误判定，测试用例终止，总的测试用例结果将是一个错误。

图 F.2/Z.140－非活动类型PTC的动态行为

活动类型 PTC 可以是非活动状态、运行状态、停止状态和取消状态。其动态行为如图 F.3 所示：



注1 — a）stop可以是一个stop、self.stop或是一个来自其它测试部件的stop；
b）kill可以是一个kill、self.kill、一个来自其它测试部件的kill，或是一个来自测试系统的kill（在发生错误的情况下）。

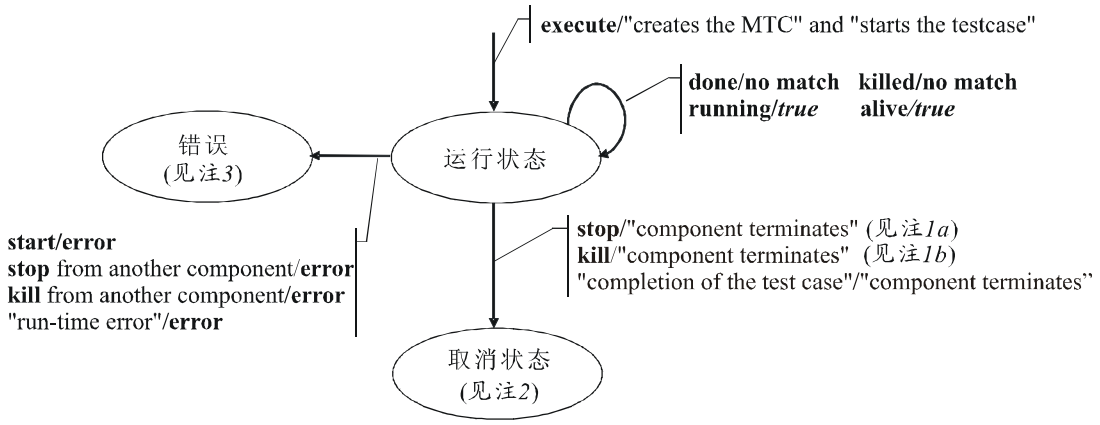
注2 — a）stop只能来自另一个测试部件；
b）kill只能来自另一个测试部件或来自测试系统（在发生错误的情况下）。

注3 — 无论何时测试部件进入错误状态，为其局部判定指派局部错误判定，测试用例终止，总的测试用例结果将是一个错误

图 F.3/Z.140－活动类型PTC的动态行为

F.2.3 MTC的动态行为

MTC 可处于运行或被取消状态。MTC 的动态行为如图 F.4 所示：

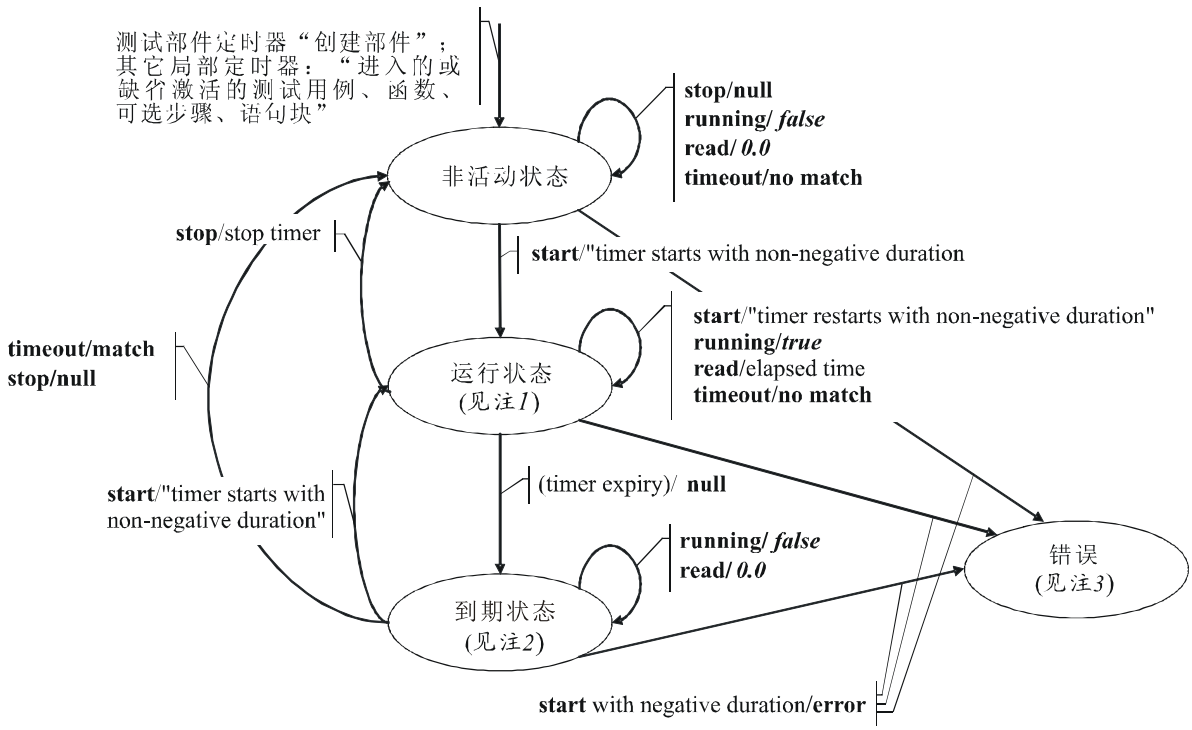


注1 — a）stop可以是一个stop、self.stop或是一个来自其它测试部件的stop；
b）kill可以是一个kill、self.kill、一个来自其它测试部件的kill，或是一个来自测试系统的kill（在发生错误的情况下）。
注2 — 也将取消所有剩余的PTC，测试用例终止。
注3 — 无论何时MTC进入错误状态，为其局部判定指派局部错误判定，测试用例终止，总的测试用例结果将是一个错误。

图 F.4/Z.140－MTC的动态行为

F.3 定时器

定时器可处于非活动、运行或过期状态。定时器的动态行为如图 F.5 所示：



注1— 对任何作用范围单元，其范围内所有处于运行状态的定时器组成运行定时器列表。
注2— 对任何作用范围单元，其范围内所有处于过期状态的定时器组成过期定时器列表。
注3— 无论何时定时器进入错误状态，它所属的测试部件也将进入其错误状态，指派一个局部错误判定，测试用例终止，总的测试用例结果将是一个错误。

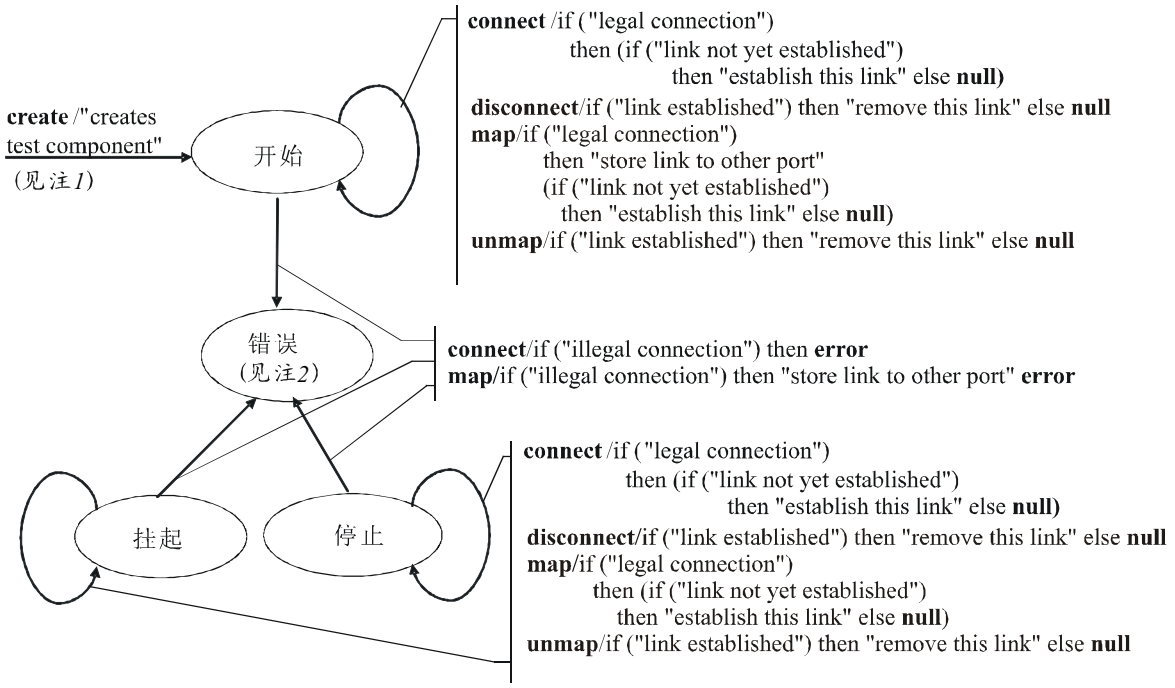
图 F.5/Z.140－定时器的动态行为

F.4 端口

端口可以处于起始或终止状态。由于其行为相当复杂，因此状态机被分为给出配置操作（即连接、断开连接、映射和取消映射）动态行为的状态机、给出端口控制操作（即启动、停止和清除）动态行为的状态机、给出通信操作（即发送、接收、调用、getcall、引起、获取、应答、getreply 和检查）动态行为的状态机。由于 trigger 是 alt 和 receive 的简写形式，在此不予考虑。

F.4.1 配置操作

端口配置操作（如 connect、disconnect、map 和 unmap）对端口状态无关紧要。其行为表现如图 F.6 所示：



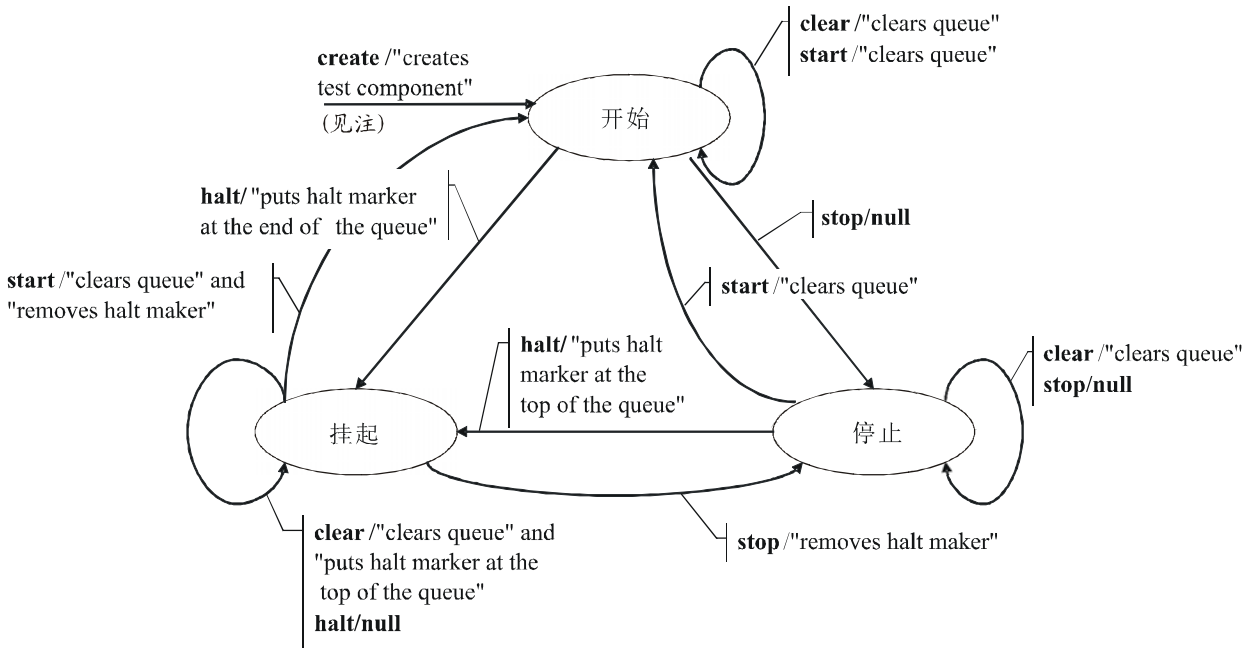
注1 — 当创建一个PTC时，创建和启动该PTC的各端口；当创建MTC时，创建和启动MTC和TSI的各端口。
注2 — 无论何时端口进入错误状态，它所属的测试部件也将进入其错误状态，指派一个局部错误判定，测试用例终止，总的测试用例结果将是一个错误。

图 F.6/Z.140－端口的动态行为：端口配置操作

这些转变不改变端口的主要状态；即端口仍然处于启动或被终止状态。

F.4.2 端口控制操作

端口控制操作的结果如图 F.7 所示：

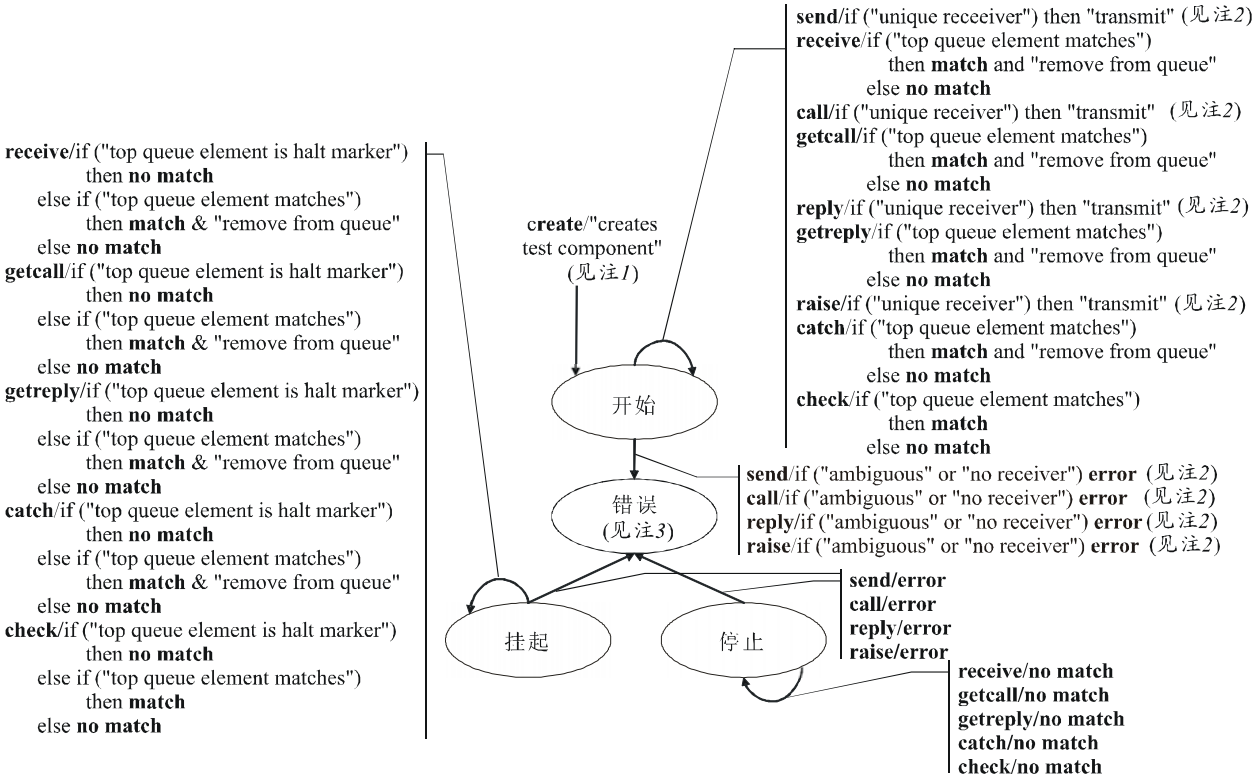


注 — 当创建一个PTC时，创建和启动该PTC的各端口；当创建MTC时，创建和启动MTC和TSI的各端口。

图 F.7/Z.140－端口动态行为：端口控制操作

F.4.3 通信操作

Send、receive、call、getcall、raise、catch、reply、getreply、check 这些通信操作的结果如图 F.8 所示：



注1 — 当创建一个PTC时，创建和启动该PTC的各端口；当创建一个MTC时，创建和启动MTC和TSI的各端口。
注2 — 如果该端口只有一条链路，或者如果目的地址表达式引用了端口连至该端口的一个测试部件，那么存在一个唯一的接收方（一个被终止的测试部件不是一个合法的接收方）。
注3 — 无论何时端口进入错误状态，它所属的测试部件也将进入其错误状态，指派一个局部错误判定，测试用例终止，总的测试用例结果将是一个错误。
注4 — trigger是alt和receive的简写形式，此处不予考虑。

图 F.8/Z.140－端口的动态行为：通信操作

附件 G（资料性的）

不赞成使用的语言特征

G.1 模块参数的组方式定义

本建议书的前一个版本要求声明模块参数时使用组方式的语法，如下例所示。在目前版本中，模块参数的语法与常量和变量的声明已经一致起来，但组方式语法还没有完全废除，还将为工具提供商和用户保留一段时间，以便从旧的语法变为新的语法。模块参数声明的组方式语法计划在本建议书的下一个版本中完全废除。

例如（多余的语法）：

```
module MyModuleWithParameters
{
  modulepar { integer TS_Par0, TS_Par1 := 0;
              boolean TS_Par2 := true
            };
  modulepar { hexstring TS_Par3 };
}
```

G.2 递归导入

本建议书的前一个版本允许以递归方式、通过导入使用命名定义的同一模块的其它定义，隐性地导入命名定义。在本版本中已不赞成使用该特性，并计划在下一个版本中完全废除。

G.3 在端口类型定义中使用`all`

本建议书的前一个版本允许在端口类型定义中使用关键字 **`all`**，而不是显性地列出允许经由给定端口的类型和特征列表。本版本已不赞成使用该特性，并计划在下一个版本中完全废除。

参考资料

- ETSI ES 201 873-1 V1.1.2 (2001-06), *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language.*
- ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.*
- ISO/IEC 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1.*
- Object Management Group (OMG): *The Common Object Request Broker: Architecture and Specification*, Chapter 3 – IDL Syntax and Semantics. Version 2.6, FORMAL/01-12-01, December 2001.
- IEEE 754 (1985), *Binary Floating-Point Arithmetic.*

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