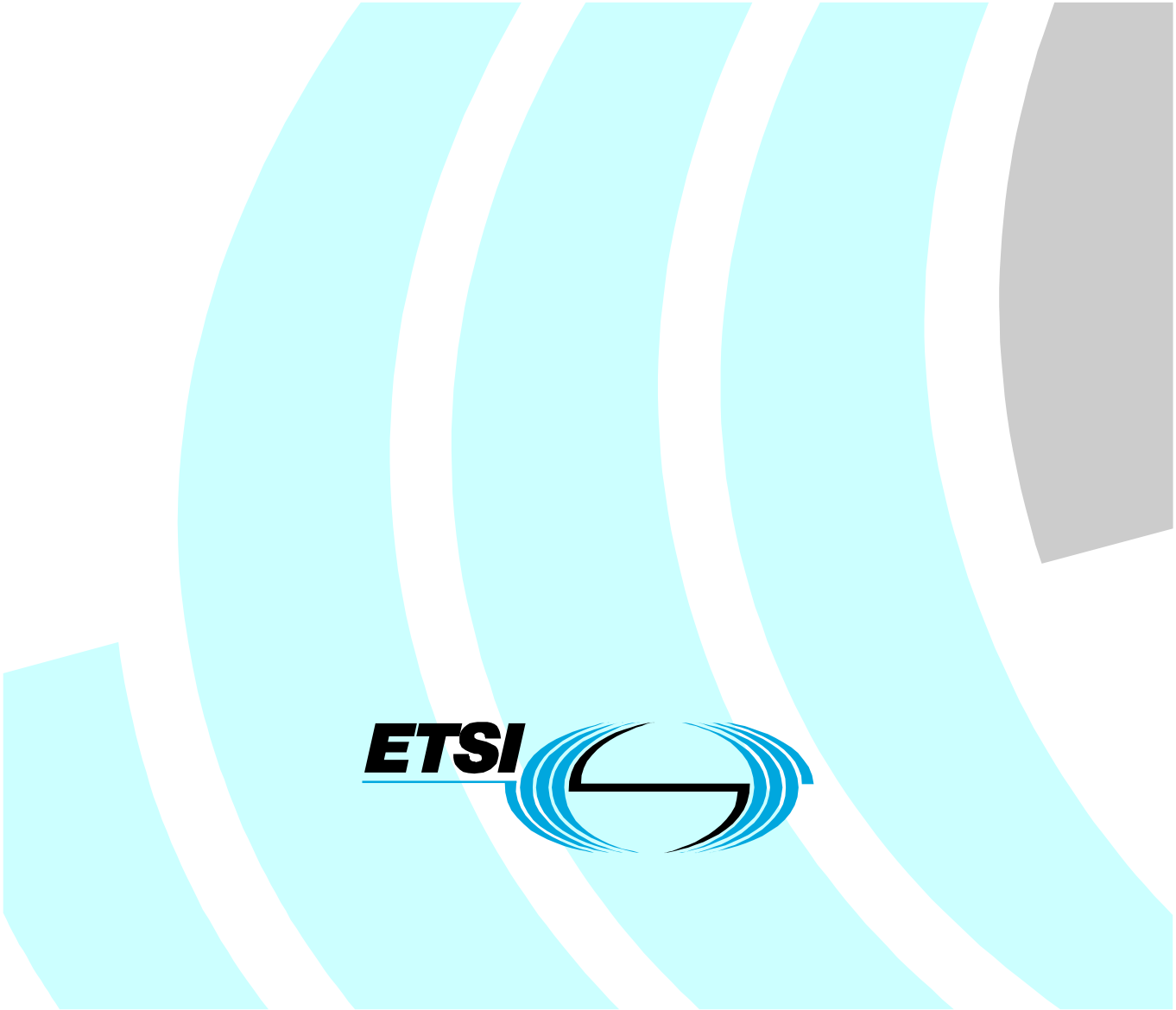
ETSI ES 201 873-1 V2.2.1 (2003-02)

ETSI 标准

测试描述方法 (MTS);
测试和测试控制表示法 第三版;
第一部分：TTCN-3核心语言

翻译：郝丹丹 付晓宇



参考
RES/MTS-00063-1 [2]

关键字
ASN.1，方法，MTS，测试，TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

重要注意事项

当前文档的各拷贝可以从此网站下载
http://www.etsi.org

当前文档可能有多个有效的电子或打印版本，在其内容间存在或可察觉出区别的情况下，以PDF格式版本为参考基准。有争议的情况，应该以ETSI打印机打印的ETSI秘书处保存的PDF版本为参考基准。

当前版本的用户应该知道本文档可能会继续修订或改变状态，其当前状态信息或其他ETSI文档的状态信息见
http://portal.etsi.org/tb/status/status.asp

如果你在当前文档中发现错误，发送您的注释到
editor@etsi.org

译者序

TTCN-3是一个功能强大的基于响应系统的黑箱测试标准，为了使TTCN在中国能更好的推广，译者经历了一年多的时间才正式完成第一个中文翻译本。

本文的翻译工作受到了ETSI的Anthony Wiles先生的大力支持，特此感谢。

译者在协议测试方面的研究和工作中虽然积累了一些TTCN的使用经验，但是由于水平有限，其中不免有错误或翻译不当之处，欢迎指正，也欢迎对TTCN-3感兴趣的朋友和我们联系、交流。

我们的 EMAIL：

ddhao@etang.com

yu_daniel@etang.com

目录

知识产权

本文所必需的或者可能必需的知识产权（IPRs）已经声明给了ETSI。如果需要有关于这些必须的知识产权信息的话，那么这些信息对ETSI成员和非成员均有效，详见ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*，该文档可以从ETSI秘书处获得，其最新更新见ETSI的Web服务器（http://webapp.etsi.org/IPR/home.asp）。

所有出版的ETSI可交付使用的文档应该包括指示读者以上信息源的信息。

前言

此ETSI标准（ES）由ETSI技术委员会测试和描述方法小组（ETSI Technical Committee Methods for Testing and Specification （MTS））制定。

本文是覆盖测试和测试控制表示法第三版（TTCN-3）的多部分可交付使用文档的第一部分，TTCN-3的各部分标识如下：

第一部分： “TTCN-3核心语言”；

第二部分： “TTCN-3表格表示格式（TFT）”；

第三部分： “TTCN-3图形表示格式（GFT）”；

第四部分： “TTCN-3操作语义”；

第五部分： “TTCN-3运行接口（TRI）”；

第六部分： “TTCN-3控制接口（TCI）”；

1 本文研究范围

本文定义TTCN第三版（或TTCN-3）核心语言。TTCN-3可以用做多种通信端口上的各种响应系统测试的描述语言。典型的应用领域是协议测试（包括移动协议和互连网协议）、服务测试（包括增补服务）、模块测试、基于平台、APIs等的CORBA测试。TTCN-3并不仅仅局限于一致性测试，它可用于多种类型的测试，如互操作性测试（Interoperability Testing）、健壮性测试（Robustness Testing）、回归测试（Regression Testing）、系统和集成测试（System and Integration Testing）。对物理层协议的测试套描述不在本文涉及范围之内。

TTCN-3意在用做独立于测试方法、层和协议的测试套的描述。TTCN-3定义了多种表示方法，如表格表示格式 [1]和图形表示格式 [2]，这些表示格式的描述不在本文研究范围之内。

本文定义了ASN.1使用的一个标准化的方式，正如ITU-T Recommendation X.680 系列[7]、[8]、[9]和[10]中定义的一样。其他语言与TTCN-3的协调使用不在本文研究范围之内。

TTCN-3的设计已经考虑到TTCN-3翻译器和编译器的最终实现，但从抽象测试套（ATS）到可执行测试套（ETS）的实现方法不在本文研究范围之内。

2 参考

下列文档通过文本引用的方式，包含了组成本文的条款的条款。

- 引用有特指的（由出版日期和/或版本号标识）或非特指的。.
- 对一个特指的引用，其后续版本并不适用。
- 对一个非特指的引用，其最新版本适用。

可能在网址http://docbox.etsi.org/Reference中找到尚未出版的被引用文档。

[1] ETSI ES 201 873-2 (V2.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT)".

[2] ETSI TR 101 873-3 (V1.1.2): "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT)".

[3] ISO/IEC 9646-1 (1994): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts".

[4] ISO/IEC 9646-3 (1998): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

[5] ISO/IEC 646 (1991): "Information technology - ISO 7-bit coded character set for information interchange".

[6] ISO/IEC 10646: "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".

[7] ITU-T Recommendation X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

[8] ITU-T Recommendation X.681: "Information technology - Abstract Syntax Notation One (ASN.1): Information object specification".

[9] ITU-T Recommendation X.682: "Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification".

[10] ITU-T Recommendation X.683: "Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications".

[11] ITU-T Recommendation X.690: "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)".

[12] ITU-T Recommendation X.691: "Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)".

[13] ISO/IEC 6429 (1992): "Information technology - Control functions for coded character sets".

[14] ITU-T Recommendation T.100 (1988): "International information exchange for interactive videotex".

[15] ITU-T Recommendation T.101 (1994): "International interworking for videotex services".

[16] ITU-T Recommendation X.660 (1992): "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: General procedures".

3 定义和缩写

3.1 定义

出于本文编写目的，ISO/IEC 9646-1 [3]、ISO/IEC 9646-3 [4]以及下面的术语和定义适用范围：

实际参数（actual parameter）：（文中简称实参）在调用位置定义的作为参数传递给被调用实体（函数、测试例、可选步等等）的值、模板或名称引用（标识符）

注意： 传递给一个调用的所有实参的数目、顺序和类型应该与被调用实体中定义的形参列表一致。

基本类型（basic types）：ES 201 873-1的章节6.1.0和6.1.1中预定义的TTCN-3类型。

注意： 通过对基本类型的名字对它们进行引用。

兼容类型（compatible type）：TTCN-3不是强类型分类，但是该语言需要类型的兼容性

注意： 如果满足章节6.7中的条件，那么变量、常量、模板等有兼容类型

通信端口（communication port）：抽象机制，使测试成分之间的通信便利

注意： 一个通信端口在接收方向上被建模如一个先进先出（FIFO）队列，端口可以是基于消息的、基于过程的或二者的混合类型。

数据类型（data types）：简单基本类型、基本串类型、结构类型、特殊数据类型以及所有用户基于以上类型自定义的类型的共用名称（见ES 201 873-1中的表3）

定义的类型或定义的TTCN-3类型（defined types or defined TTCN-3 types）：所有TTCN-3预定义类型（基本类型、所有结构类型、anytype类型、地址类型、端口类型、成分类型和默认类型）和所有在模块中或从其他TTCN-3模块中传入的用户定义类型

动态参数化（dynamic parameterization）：参数化的一种，在动态参数化过程中，实参依赖于运行期事件，如实参的值是在运行期间被接收或依赖于通过一种逻辑关系接收到的值

例外（exception）：在基于过程的通信的情况下，如果一个应答实体不能以正常期望的响应来响应对一个远程过程调用，则它将产生例外（如果定义了例外的话）

形式参数（formal parameter）：（文中简称形参）值或模板的名称引用（标识符），不是在一个实体（函数、测试例、可选步等）定义时被解析，而是在调用它的时候被解析

注意： 从调用实体的地方传递在形参的位置使用的实际值或模板（或它们的名称）。（又见实参定义）

全局可见性（global visibility）：实体（模块参数、常量、模板等）的属性，可以在定义实体的模块中任意位置引用该实体的标识符，包括所有函数、测试例以及定义在相同模块的可选步和该模块的控制部分

实现一致性声明（Implementation Conformance Statement, ICS）：见ISO/IEC 9646-1 [3]

测试实现附加信息（Implementation eXtra Information for Testing, IXIT）：见ISO/IEC 9646-1 [3]

被测实现（Implementation Under Test, IUT)：见ISO/IEC 9646-1 [3]

已知类型（known types）：定义类型、引入的ASN.1和其它引入的外部类型集合

局部可见性（local visibility）：实体（常量和变量等）的属性，仅可在定义实体的函数、测试例或可选步中引用实体的标识符

主测试成分（Main Test Component, MTC)：见ISO/IEC 9646-3 [4]

通过值传参（passing parameter by value）：在进入一个可参数化的实体之前计算变量（arguments）值的传递参数的方式

注意： 只传递了变量的值，在被调用实体中该变量的改变对调用者看到的实际参数没有影响。

通过引用传参（passing parameter by reference）：在进入函数、可选步等之前不计算变量（arguments）值的传递参数的方式，通过调用过程（函数、可选步等）调用被调用过程来传递对参数的引用

注意： 被调用过程中该变量的变化会影响到调用者看到的实际参数。

并行测试成分（Parallel Test Component, PTC）：见ISO/IEC 9646-3 [4]

源类型（root type）：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的默认类型

注意： 对于基于引入的ASN.1类型，源类型由相关联的TTCN-3类型决定（见章节D.1.2）。

静态参数化（static parameterization）：参数化的一种，在静态参数化过程中，实际参数独立于运行期事件，也就是说，在编译时知道或在测试套执行启动时知道模块参数的情况（如从测试套说明中获知、计算引入的定义或者在执行前测试系统得知参数的值）

强分类（strong typing）：根据类型名称相同来严格要求类型的兼容性，不允许有例外

被测试系统（System Under Test, SUT）：见ISO/IEC 9646-1 [3]

注意： 编译时获知所有类型，也就是说静态绑定。

模板（template）：TTCN-3模板是特殊的数据结构，用于传输一组不同的值（distinct values）或检查收到的一组值是否与模板说明匹配

测试例（test case）：见ISO/IEC 9646-1 [3]

测试例错误（test case error）：见ISO/IEC 9646-1 [3]

测试套（test suite）：TTCN-3模块，它通过引入语句（import statements）显式或隐式地完整描述所有定义和完整定义一组测试例所需的行为描述

测试系统（test system）：见ISO/IEC 9646-1 [3]

测试系统接口（test system interface）：提供从（抽象）TTCN-3测试系统的可用端口到实际测试系统的可用端口的映射

类型兼容性（type compatibility）：语言特性，它允许使用一个给定类型的值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）

注意： 类型和值活模板的当前值应该互相兼容。

值的参数化（value parameterization）：把一个值或模板作为一个实参传给一个参数化对象的能力

注意： 这个实际值参数将完成该参数化对象的描述。

用户定义类型（user-defined type）：使用点“.”表示通过定义一个基本类型的子类型、声明一个结构类型或把**anytype**类型限制到一个单一类型中去来定义的类型

注意： 通过使用用户定义类型的标识符（名称）来引用它们。

值符号（value notation）：符号，通过它一个标识符和一个给定的值或一个特定类型的范围相关联

注意： 值可以是常量或变量。

3.2 缩写

由于本文的编写目的，下面的缩写适用于：

API 应用程序接口（Application Programming Interface）

ASN.1 抽象语法符号1（Abstract Syntax Notation One）

ASP 抽象服务原语（Abstract Service Primitive）

ATS 抽象测试套（Abstract Test Suite）

BNF 巴科斯范式（Backus-Nauer Form）

CORBA 公用对象请求调度体系结构（Common Object Request Broker Architecture）

ETS 可执行的测试套（Executable Test Suite）

FIFO先进先出（First In First Out）

IDL 接口描述语言（Interface Description Language）

IUT 被测实现（Implementation Under Test）

MTC主测试成分（Master Test Component）

PDU 协议数据单元（Protocol Data Unit）

PTC 并行测试成分（Parallel Test Component）

(P)ICS （协议）实现一致性声明（（Protocol） Implementation Conformance Statement）

(P)IXIT （协议）测试实现附加信息（（Protocol）Implementation eXtra Information for Testing）

SUT 被测系统（System Under Test）

TTCN 测试和测试控制符号表示法（Testing and Test Control Notation）

4 介绍

4.0 概述

TTCN-3是一种灵活和强有力的语言，它用于描述在多种通信端口上的各种响应系统测试。它应用的典型领域是协议测试（包括移动和互联网协议）、服务测试、基于平台的CORBA测试、API测试等等。TTCN-3并

不仅限于一致性测试，它可以用于许多其他种类的测试，如互操作性测试、健壮性测试、回归测试、系统和集成测试。

从语法的角度看，TTCN-3与在ISO/IEC 9646-3 [4]中定义的该语言的早期版本有很大区别。然而，它保留了大量TTCN的经证实的基本功能性，并在某些方面做了改进。TTCN-3包括以下重要特性：

- 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；
- 描述编码信息和其他属性（包括用户扩展性）的能力；
- 描述数据和带有强有力的匹配机制的属性模板的能力；
- 类型和值的参数化；
- 赋值和测试判定的处理；
- 测试套参数化和测试例选择机制；
- TTCN-3使用和ASN.1的结合（以及与其他语言结合使用的潜力，如与SDL的结合）；
- 良好定义的语法，格式的互换以及静态语义；
- 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

从TTCN的产生和发展看，TTCN总是和一致性测试联系在一起。为了在标准和工业两个领域把该语言的应用范围扩展到更为广泛的测试应用范围， TTCN-3被分为多个部分进行说明：第一部分由本文定义，是TTCN-3的核心语言；第二部分在ES 201 873-2 [1]中定义，是TTCN-3的表格表示格式，在外部特征和功能性方面与TTCN的早期版本相似；第三部分在TR 101 873-3 [2]中定义，是TTCN-3的图形表示格式；第四部分包含TTCN-3的操作语义。

核心语言有三个目的：

a) 作为广义的基于文本的TTCN-3测试语言；

b) 作为TTCN工具之间TTCN测试套的标准化互换格式；

c) 作为各种表示格式的语义基础（如相关，也是语法基础）。

核心语言使用时可以独立于表示格式。但是，表格格式和图形格式却不能独立于核心语言使用。这些表示格式的使用和实现应该基于核心语言。

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，其他格式可以是标准化的表示格式或是TTCN-3用户自己定义的私有表示格式。本文中不定义这些附加格式。

TTCN-3与ASN.1完全协调，可以在TTCN-3模块中选择性地使用ASN.1作为一个替换数据类型和值语法。TTCN-3中ASN.1的使用在本文的附录D中定义。TTCN-3与ASN.1结合的方法可以用于支持其他类型和值的系统与TTCN-3的结合使用。然而，本文并不定义后者的细节。

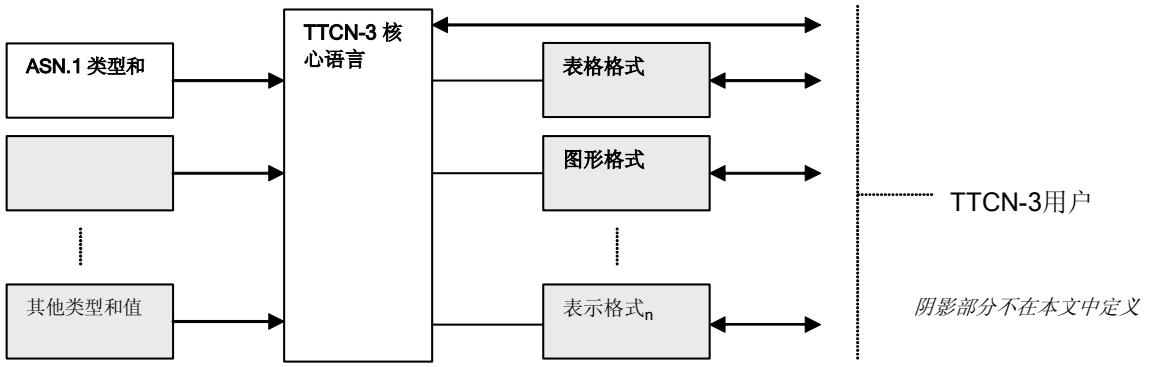


图 1：核心语言和各种表示格式的用户视图

核心语言由一个完整语法（见附录A）和操作语义（见文档的第四部分）定义，出于一些基础的应用领域或方法上的考虑，它包含不限制语言使用的最小静态语义（本文主体部分及附录A中给出），使用私有工具实现的测试套索引等TTCN以前版本的功能性不属于TTCN-3。

4.2 描述的一致性

根据当前文档主体中的文本描述（章节5至28）和附录A中的形式化方法，从语法和语义的角度详细说明TTCN-3语言，在每种情况下，当文本描述不能穷举时，用形式化描述完成。如果文本和形式化的描述有矛盾的时候，应该以形式化描述优先。

4.3 一致性

本文不描述语言的实现层次，但是，对一个声称与TTCN-3语言一致的实现来说，本文所有实现了的特性应该与文本给出的要求一致。

注意： 这不妨碍实现与任何不在当前文档中描述的额外特性相一致的实现。

5 基本语言元素

5.0 概述

TTCN-3的顶层单元是模块。一个模块中不能包含子模块，但是它可以从其他模块中引入定义。模块可以带有参数列表去提供测试套参数化的一个形式，这与TTCN-2的PICS和PIXIT参数化机制相似。

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测试成分、通信端口、数据类型、常数、测试数据模板、函数、测试端口上调用的过程特征（signatures）、测试例等等。

模块的调用部分调用测试例并控制它们的执行。控制部分也可以声明（局部）变量等，程序语句（如**`if-else`**和**`do-while`**）可以用于各个测试例的选择和执行顺序 。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（records）、集合（sets）、联合（unions）、枚举（enumerated）类型和数组。引入的ASN.1类型和值可以与TTCN-3一起使用。

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

测试例表达动态测试行为，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，可以给TTCN-3语言元素赋予属性，如编码信息和显示属性，同样也可以描述（非标准化的）用户定义的属性。

表 1： TTCN-3语言元素一览表

语言元素	相关联的关键字	是否在模块定义中被描述	是否在模块控制中被描述	是否在函数/可选步/测试例中被描述	是否在测试成分中被描述
TTCN-3模块定义	**module**				
其他模块的定义引入	**import**	是			
组定义	**group**	是			
数据类型定义	**type**	是			
通信端口定义	**port**	是			
测试成分定义	**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			
函数定义	**function**	是			
可选步定义	**altstep**	是			
测试例定义	**testcase**	是			
变量声明	**var**		是	是	是
定时器声明	**timer**		是	是	是

5.1 语言元素的顺序

通常来说，声明的顺序是任意的。在一个语句和声明块中，如函数体或一个**if-else**语句分支中，所有的声明（如果有的话）应该仅在该块的开始处进行声明。

例：

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

:
```

5.1.1 向前引用

模块定义部分中的定义可以按任何顺序进行定义，但考虑到可读性因素，应该避免向前引用，不过这并不是必须的。例如，调用其他函数和模块的参数化的递归元素就可能导致不可避免的向前引用。

仅对模块定义中声明允许的向前引用，应该不在模块的控制部分、测试例的定义部分、函数和选择步中使用向前引用。这就意味着对局部变量、局部定时器和局部常量的向前引用绝不会发生。而这个规则的唯一例外是标签，可以在**goto**语句中使用用于标签的向前引用去跳转到前面（见章节19.5）。

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（value）的参数化：

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

b) 语言元素模块（**module**）允许静态的值参数化去支持测试套参数，也就是说，在编译时这个参数化既可以是可解析的也可以是不可解析的，但是它应该在运行开始时被解析（即在运行时是静态的）。这就意味着在运行时模块的参数值是全局可见的，但是不能改变；

c) 所有用户定义的类型定义（包括结构化的类型定义，如record、set等）和特殊的配置类型地址－－类型（address）支持静态值的参数化，即这个参数化应该在编译时进行解析；

d) 语言元素template、signature、testcase、altstep和function支持动态的值参数化（即这个参数化过程应该在开始运行时进行）。

表2给出了可以被参数化的语言元素以及什么可以作为参数传入的语言元素一览表：

表2：TTCN-3可参数化的语言元素一览表

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 Static at start of run-time	所有基本类型、所有用户自定义类型和地址类型（**address**）的值。
type (note 1)	在编译时，静态 Static at compile-time	所有基本类型、所有用户自定义类型和地址类型（**address**）的值。
template	在运行时，动态 Dynamic at run-time	所有基本类型、所有用户自定义类型、地址类型（**address**）和模板类型（**template**）的值。
function	在运行时，动态 Dynamic at run-time	所有基本类型、所有用户自定义类型、地址（**address**）、成分（**component**）端口（**port**）、默认（**default**）、模板（**template**）和定时器（**timer**）类型的值。
altstep	在运行时，动态 Dynamic at run-time	所有基本类型、所有用户自定义类型、地址（**address**）、成分（**component**）端口（**port**）、默认（**default**）、模板（**template**）和定时器（**timer**）类型的值。
testcase	在运行时，动态 Dynamic at run-time	所有基本类型、所有用户自定义类型、地址类型（**address**）和模板类型（**template**）的值。
signature	在运行时，动态 Dynamic at run-time	所有基本类型、所有用户自定义类型、地址类型（**address**）和成分类型（**component**）的值。
注意1： **record of**、**set of**、**enumerated**、**port**、**component**和**subtype**类型定义不允许参数化。 注意2： 不同语言元素中参数化的例子和特殊用法在本文的相关章节中给出。		

5.2.1 参数传递——传参和传值

5.2.1.0 概要

默认来讲，基本类型、基本串类型、用户定义的结构类型、地址类型和成分类型是通过传值来传递所有的实际参数的。可以选择性地使用关键字**in**来表示传值，而如果使用传参的方法传递以上提到类型的参数，则应该使用关键字**out**或**inout**。

定时器和端口类型参数总是通过传参的方法进行参数传递，并通过关键字**timer**和**port**来标识。可以选择性地使用关键字**inout**来表示使用传参的方法进行参数传递。

5.2.1.1 使用传参的方法传递参数

使用传参的方法来传递参数有以下限制：

a) 只对**altsteps**的形参列表进行显式调用，**functions**、**signatures**和**testcase**可以包含传参的参数（pass-by-reference parameters）；

注意： 如何在过程特征中使用传参参数有进一步的限制（见章节23）。

b) 实际参数应该仅是变量（如：不是常量或模板）。

例：

```
function MyFunction(inout boolean MyReferenceParameter){ … };
// 通过传参来传递MyReferenceParameter，且可以在该函数中读出和设置该实参。

function MyFunction(out boolean MyReferenceParameter){ … };
// 通过传参来传递MyReferenceParameter，且仅可以在该函数中设置该实参。
```

5.2.1.2 使用传值的方法传递参数

传值参数可以是变量，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元素的数目和它们顺序应该与其相应的形参列表中的元素数目和出现顺序相同。而且，每个实参的类型应该与相应的形参的类型是兼容的。

例：

```
// 带有形参列表的一个函数定义
function MyFunction(integer FormalPar1, boolean FormalPar2, bitstring FormalPar3) { … }

// 带有实参列表的一个函数调用
MyFunction(123, true,'1100'B);
```

5.2.3 空形参列表

如果TTCN-3语言元素**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时都应该包含这个空的括号。在所有其他的情况下，这个空的括号是可以省略的。

例：

```
// 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 嵌入式参数列表

通常，被描述为一个实参的所有参数化实体应该在实参列表中解析它们自己的参数。

例：

```
// 给定的消息定义
type record MyMessageType
{
    integer     field1,
```

```
    charstring  field2,
    boolean     field3
}

// 一个消息模板可以是
template MyMessageType MyTemplate(integer MyValue) :=
{
    field1 := MyValue,
    field2 := pattern "abc*xyz",
    field3 := true
}

// 带有一个模板参数的测试例可以是
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{   :
    MyPCO.receive(RxMsg);
}

// 当测试例在控制部分中被调用且该参数化模板用作一个实参时，必须提供该模板的实参
control
{   :
    TC001(MyTemplate(7));
    :
}
```

5.3 范围规则

5.3.0 概要

TTCN-3提供六个基本的范围单位：

a) 模块定义部分；

b) 模块的控制部分；

c) 成分类型；

d) 函数；

e) 可选步（altsteps）；

f) 测试例；

g) 复合语句中的“声明和语句块”。

注意1： 用于组（groups）的附加范围规则在章节7.3.1中给出。

注意2： 用于**for**循环中计数的附加范围规则在章节19.7中给出。

范围的每个单位由声明（可选的）组成。可以通过使用TTCN-3语言语句和操作，用范围单位－－模块的控制部分、函数、测试例、可选步和复合语句中的“声明和语句块”来额外描述行为的某种形式（见章节18）。

在模块定义部分中但在其他范围单位之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，包括该模块定义的所有函数、测试例和可选步以及控制部分。从其它模块中引入的标识符对于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的是局部可见性，即只能用在该控制部分中。

通过使用一个**runs on**字句，在测试成分类型中做的定义仅可以在调用该成分类型或一个相一致的成分类型的函数、测试例和可选步中使用它们（见章节16.3）。

函数、测试例和可选步是独立的范围单位，它们之间没有层次关系，即它们主体开始处做的声明具有局部可见性，且仅可以用在给定的函数、测试例和可选步中（如：在一个测试例中做的声明对于被这个测试例调用的函数或被该测试例使用的可选步来说是不可见的）。

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、测试例、可选步、函数中，或嵌套在其它复合语句中，如在一个**while**-循环中使用**if-else**-语句。

复合语句和嵌套式复合语句中的“语句和声明块”对于包括给定“语句和声明块”的范围单位和任意嵌套的“语句和声明块”来说具有层次关系，在一个“语句和声明块”中所做的定义具有局部可见性。

范围单位的层次关系如图2所示。高层的范围单位声明对其所在层次关系中的同一分支中下面层次的所有单位来说是可见的，而层次关系中低层的范围单位声明对于其上层的那些单位来说是不可见的。

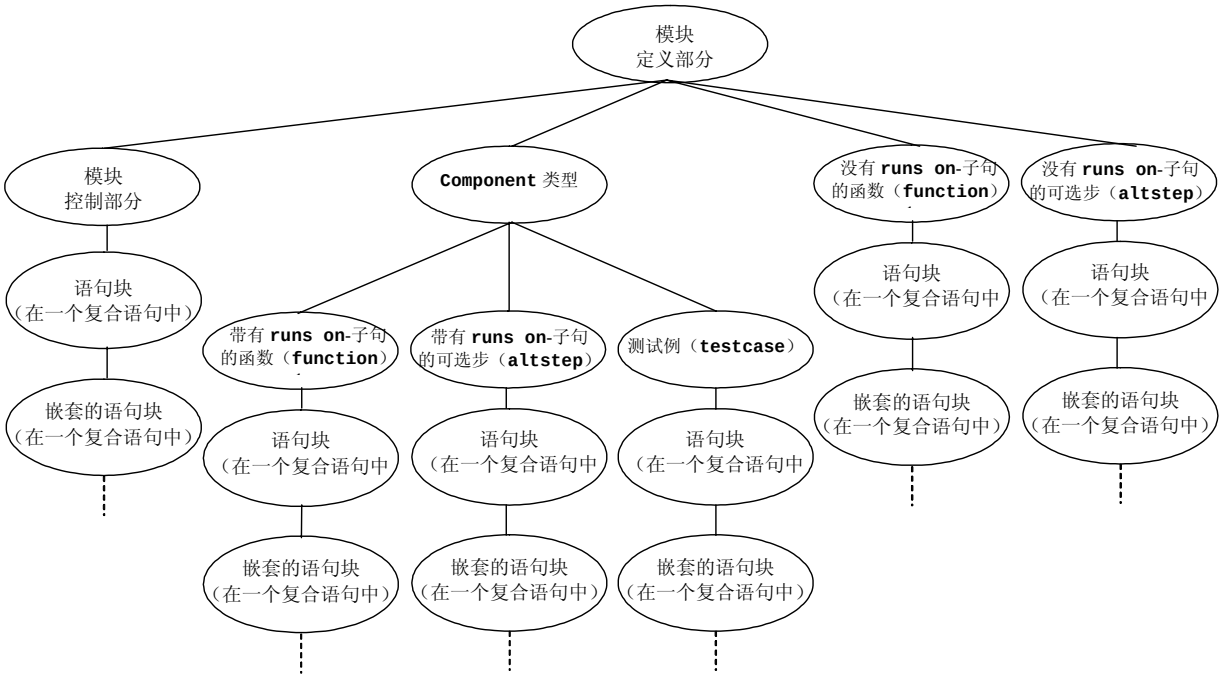


图 2：范围单元的层次关系

例：

```
module MyModule
{   :
    const integer MyConst := 0; // 对于MyBehaviourA和MyBehaviourB来说，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

在一个参数化的语言元素中（如在一个函数调用中）形参的范围应该应该限定到这些参数出现的定义中和相同层次关系中的较低的范围层次，这就是说它们遵循正常的范围规则（见章节5.4）。

5.3.2 标识符的唯一性

TTCN-3要求标识符具有唯一性，即在相同范围层次中的所有标识符互不相同。这就意味着在同一个范围层次的分支中，低层范围中的声明不应该重复使用与高层范围声明中相同的标识符。结构类型字段、枚举值和组的标识符不必全局唯一，然而在枚举值的情况下，标识符应该仅被其它枚举类型中的枚举值重复使用。标识符唯一性规则应该也用于形参标识符。

例：

```
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标识符是大小写敏感的，而关键字应该小写（见附录A）。TTCN-3的关键字应该即不用作是TTCN-3对象的标识符，也不用作是从其他语言的模块中引入对象的标识符。

6 类型和值

6.0 概要

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，如整形（**integer**）、布尔类型（**boolean**）和串类型，也包括一些TTCN-3特殊的类型，如对象标识类型（**objid**）和判定类型（**verdicttype**）。可以从这些基本类型中构造结构类型，如记录类型（**record**）、集合类型（**set**）和枚举类型（**enumerated**）。

特殊的数据类型－**anytype**类型定义作是一个模块中所有已知类型的联合（union）。

与测试配置相关的特殊类型，如地址类型（**address**）、端口类型（**port**）和成分类型（**component**）可以用来定义测试系统的体系结构（见章节22）。

特殊类型**default**类型可以用于默认处理（见章节21）。

TTCN-3类型汇总见表3。

表 3：TTCN-3类型一览表

类型分类	关键字	子类型
简单基本类型	**integer**	range, list
	char	range, list
	universal char	range, list
	float	range, list
	boolean	list
	objid	list
	verdicttype	list
基本串类型	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	range, list, length
	universal charstring	range, list, length
结构类型	**record**	list
	record of	list, length
	set	list
	set of	list, length
	enumerated	list
	union	list
特殊的数据类型	**anytype**	list
特殊的配置类型	**address**	
	port	
	component	
特殊的默认类型	**default**	

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`**：整型，其值为所有的正、负整数和零。

整型值应该用一个或多个数字表示，且除0值外其第一位不应该是0，而0应该由一位数字表示（即单个0）。

b) **`char`**：字符型，其值为与ISO/IEC 646 [5]的章节8.2中描述的国际参考版本（International Reference Version, IRV）相符的ISO/IEC 646 [5] 版本中的字符。

注意1： ISO/IEC 646 [5]中的IRV版本与ITU-T Recommendation T.50（见参考书目）中描述的国际参考字母（International Reference Alphabet，以前是International Alphabet No.5 - IA5）的IRV版本等价。

字符类型的值可以用双引号（"）括起来给出，或者使用预定义的带有编码参数的转换函数计算获得。

相关的操作符号相等（==）和不相等（!=）可以用来比较**`char`**类型的值。

c) **`universal char`**：通用字符类型，其值为来自ISO/IEC 10646 [6]的单个字符。

`universal char`类型的值可以用双引号（"）括起来给出，或者使用预定义的带有编码参数的转换函数计算获得或由一个四元组（quadruple）给出。 这个四元组仅能表示一个单个字符，且它表示一个字符是根据ISO/IEC 10646 [6]使用该字符的组（group）、容器（plane）、行（row）和单元格（cell）的十进制值来表示，由关键字**`char`**来引导，带有一对括号，并用逗号分隔（例如，**`char`** (0, 0, 1, 113) 表示匈牙利字符“ű”）。

注意2： 控制字符仅可以使用四元组的格式表示。

默认地，**universal char**应该与ISO/IEC 10646 [6]的章节14.2中描述的UCS-4编码表示格式相一致。这个默认的编码可以通过使用定义的编码属性来对其进行重写（overridde）（见章节28.2.1）。

注意3： 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。

相关的操作符号相等（==）和不相等（!=）可以用来比较**universal char**类型的值。

d) **float**：浮点类型，描述浮点数的一个类型。

浮点数表示作： <*尾数*> × <*基数*>^{<*指数*>}

其中，<*尾数*>是一个正或负整数， <*基数*>是一个正整数（多数情况为2、10或16），<*指数*>为一个正或负整数。

浮点数的表示限定为以值10为基数，浮点值可以使用下列任一方式表示：

- 在一个数字序列中使用小数点的正常表示，如1.23（表示123*10^{-2}），2.783（即2783 × 10^{-3}），或-123.456789（表示-123456789 × 10^{-6}）；或者
- 使用E来分开两个数字来表示，前一个数字描述尾数，第二个数字描述指数，例如12.3E4（表示12.3 × 10^{4}），-12.3E-4（表示-12.3 × 10^{-4}）。

e) **boolean**：布尔类型，该类型有两个不同值。

布尔类型的值应使用**true**和**false**来表示。

f) **objid**：对象标识类型，其值为与ITU-T Recommendation X.660 [16]的章节6.2一致的所有对象标识符的集合，标识符中的连字符被替换为下划线。

例：

{itu_t(0) identified_organization(4) etsi(0)}

or alternatively {itu_t identified_organization etsi}

or alternatively { 0 4 0}

g) **verdicttype**：判定类型，该类型有五个不同的值。

Verdicttype类型的值应该使用**pass**、**fail**、**inconc**、**none**和**error**表示。

6.1.1 基本串类型和值

TTCN-3支持下列基本串类型：

注意1： TTCN-3中的通用术语“串”或“串类型”指的是比特串（**bitstring**）、十六进制串（**hexstring**）、八位组串（**octetstring**）、字符串（**charstring**）和通用字符串（**universal charstring**）。

a) **bitstring**：比特串类型，其值为不同的0位、1位或多位的0、1序列。

bitstring类型值应该用任意数目的比特数0、1来表示（可能为0位），以字符“'”开始，后接字符“'B”。

例1：

'01101'B

b) **hexstring**：十六进制串类型，其值为0位、1位或多位十六进制数的有序序列，每个十六进制数与一个有序的四比特序列相符。

Hexstring类型值应该用任意数目的十六进制数字来表示（可能为0位）：

0 1 2 3 4 5 6 7 8 9 A B C D E F

以字符“'”开始，后接字符“'H”；使用十六进制表示法每个十六进制数字用于表示半个八位组的值。

例2：

'AB01D'H

c) **octetstring**：八位组串类型，0个或正偶数个十六进制数字的有序序列（每对数字有一个有序的八比特序列相应）。

octetstring类型值应该用任意但必须是偶数数目的十六进制数字来表示（可能为0位）：

0 1 2 3 4 5 6 7 8 9 A B C D E F

以字符“'”开始，后接字符“'O”；使用十六进制表示法每个十六进制数字用于表示半个八位组的值。

例3：

'FF96'O

d) **charstring**：字符串类型，其值为0个、1个或多个与ISO/IEC 646 [5]的章节8.2中描述的国际参考版本International Reference Version，IRV）相符的ISO/IEC 646 [5]版本的字符（characters of the version of ISO/IEC 646 [5]）（见章节中的注意1）。

由关键字**universal**引导的字符串类型表示类型值为来自ISO/IEC 10646 [6]的0个、1个或多个字符的类型。

Charstring类型值应该由来自相关字符集合的任意数目个字符表示，并使用双引号（"）把它括起来。

在串中需要包含字符双引号（"）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，且其间没有空格字符。

例4： ""abcd""表示文字串"abcd"。

universal charstring类型值也可以用来自相关字符集合的任意数目个字符表示，并使用双引号（"）把它括起来或使用一个四元组（quadruple）。这个四元组仅能表示一个单个字符，且它使用ISO/IEC 10646 [6]中该字符的组（group）、容器（plane）、行（row）和单元格（cell）的十进制值来表示这个字符，由关键字**char**来引导，带有一对括号，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（使用一对双引号）时，在串中需要包含字符双引号（"）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，且其间没有空格字符。在使用连接操作符（concatenation operator）的一个串值的表示法中，可以混合地使用两种方法。

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

注意2： 仅可以使用四元组来表示控制字符。

默认地，**universal charstring**应该与ISO/IEC 10646 [6]章节14.2中描述的UCS-4编码表示格式一致。

注意3： 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。

可以使用定义的编码属性（见章节28.2.1）来重写这个默认编码。在附录E中定义了使用这些属性的有用的字符串类型utf8string、bmpstring、utf16string和iso8859string。

6.1.2 存取单个串元素

可使用一个类似数组的文法来访问一个串类型中的各元素，串中仅一个元素可以被存取。

表表4指出了不同串类型元素的长度单位。

索引应该以0值开始。

例：

```
// 给定
MyBitString := '11110111'B;
// 然后做
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 概要

用户定义类型用关键字**type**表示，可以根据表3使用用户定义类型在简单基本类型和简单串类型上创建子类型（如列表（lists）、范围（ranges）和长度（length）限制）。

6.2.1 值列表

TTCN-3允许对表3中给出的任一类型的不同值列表的描述。该列表的值应该是源类型的值，且应该是该源类型定义的值集合的真子集，被这个列表定义的子类型限定了该子类型的允许值为列表中的那些值。

例：

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
type float pi (3.1415926);
type universal char SpecialLetter (char(0, 0, 1, 111), char(0, 0, 1, 112), char(0, 0, 1, 113));
```

6.2.2 值域

6.2.2.0 概要

TTCN-3允许对类型**integer**、**char**、**universal char**和**float**类型（或这些类型的派生类型）值域的描述。这个值域定义的子类型限定了该子类型的值可为值域中的值，且包括该值域的上下界。在**char**和**universal char**类型的情况中，值域的边界值应该根据该类型被编码的字符集合表去计算有效的字符位置（例如给定的位置不应该为空）。并认为上下边界中间的空位置不是所描述值域中的有效值。

例1：

```
type integer MyIntegerRange (0 .. 255);
type char MyCharRange ("a" .. "z");
type float piRange (3.14 .. 3142E-3);
```

char类型的值域描述也可以用在一个**charstring**子类型定义中，而**universal char**类型的值域描述可以用在**universal charstring**子类型定义中。在这样的情况下，值域为串中每个单独字符每个限定了允许的取值范围。

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之内
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使用双引号所描述的值域之内
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之内
```

6.2.2.1 无限值域

为了描述一个无限的整型或浮点型的值域，可以使用关键字**infinity**来代替一个值来表示没有上下边界。上边界应该大于等于下边界。

例：

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

注意： 无限的“值”是依赖于实现的，这个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

对于类型**integer**、**char**、**universal char**和**float**（或这些类型的派生类型）的值来说，也可以混合使用列表和值域。

例：

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
type char MyCharRange ("a", "b", "c", "0" .. "9");
```

在**charstring**和**universal charstring**子类型定义中，不应该在相同的子类型定义中混合使用列表和值域。

6.2.3 串长度限定

TTCN-3允许在串类型上对长度限定进行描述。根据使用长度边界的串类型的不同，该长度边界具有不同的复杂度。在所有的情况中，这些边界都应该为非负整型值（或是派生的整型值）。

例：

```
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，对大长度为8
```

表表4描述了不同串类型的长度单位。

表4：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	八位组
character strings	字符

关键字**infinity**用于上界的时候表示长度没有上限。上限应该大于等于下限。

6.3 结构化的类型和值

6.3.0 概要

关键字**type**也用于描述结构化的类型，如记录类型（**record**）、**record of**类型、集合类型（**set**）、**set of**类型、枚举类型（**enumerated**）和联合类型（**union**）。

可以使用一个明确的赋值表示或一个简写的值列表给出这些类型的值。

例1：

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

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

当使用赋值表示方法 描述部分值的时候（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被描述。在结构中使用值列表表示时，应该使用一个值来描述所有的字段，用符号“-”或关键字**omit**表示不使用的字段。

例2：

```
var MyRecordType MyVariable:=               // 赋值
{
    field1 := '11001'B,
    field3 := "A string"
}

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

在同一（紧接着的）上下文中，不允许混合使用这两种值表示方法。

例3：

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

无论是在赋值表示方法还是值列表表示方法中，都应该对可选字段使用明确的值**omit**来省略相关字段。省略一个字段会引起相关字段值变成未定义字段，而不管该字段以前具有什么样的值。对强制字段（mandatory fields）不应该使用关键字**omit**。

6.3.1 记录类型和值

6.3.1.0 概要

TTCN-3支持有序的结构化类型，即记录类型（**record**）。一个**record**类型元素可以是基本类型或用户定义数据类型（如其它记录、集合或数组类型）的任一种，一个**record**值应该与该**record**字段的类型兼容。对于**record**，其元素标识符是该**record**的本地标识符，且在该**record**中是唯一的（但不必全局唯一）。**record**类型的常量应该既不直接也不间接包含变量或模块参数作为字段值。

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

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

可以定义记录没有字段（即作为一个空记录）。

例1：

```
type record MyEmptyRecord { }
```

把一个记录值赋值给一个单个的元素基（element basis）。

例2：

```
var integer MyIntegerValue:= 1;

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

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

或使用一个值列表。

例3：

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

应该使用省略符号省略可选字段。

例4：

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

// 注意这与下面的写法不同
// MyRecordValue:= {MyIntegerValue, -, "A string"}
// 后面的写法意味着field2的值不变
```

6.3.1.1 引用一个record类型的字段

应使用点号来对**record**类型进行引用：*类型或值标识符*.*元素标识符*，*类型或值标识符*用来解析一个结构化类型或变量的名字，*元素标识符*用来解析结构化类型中一个字段的名字。

例：

```
MyVar1 := MyRecord1.myElement1;
// 如果一个record类型嵌套在另外一个类型中，那么对它的应用可以看起来象下面的格式
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 record类型的可选元素

应使用关键字**optional**来描述一个**record**类型的可选元素。

例：

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

6.3.2 集合（Set）类型和值

6.3.2.0 概要

TTCN-3支持无序的结构化类型，即集合类型（**set**）。**set**类型和值与**record**类型很相似，只是**set**类型字段的顺序 是没有意义的。

例：

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

set类型字段标识符对于**set**类型来说是本地的，且在该**set**类型中应该唯一（但是不必是全局唯一的）。

set类型值不应使用值列表表示方法。

6.3.2.1 对集合类型字段的引用

应使用点号对**set**类型元素进行引用（见章节6.3.1.1）。

例：

```
MyVar3 := MySet1.myElement1;
// 如果一个set类型嵌套在另一个类型中，那么该引用可以看起来象下面的格式
MyVar4 := MyRecord1.myElement1.myElement2;
// 注意，带有被引用标识符myElement2的set类型嵌套在一个record类型中。
```

6.3.2.2 集合中的可选元素

应使用关键字**optional**来描述**set**类型中的可选元素。

6.3.3 单一类型的记录和集合类型

TTCN-3支持对所有元素为同一类型的**record**和**set**类型的描述，并使用关键字**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数组相似。

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

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个整数的记录类型

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**类型的值表示应该是一个值列表表示法或是一个对各元素进行索引的表示方法（与用于数组的值表示法相同，见章节6.5）。

当使用值列表表示法时，列表中的第一个值被赋值给第一个元素，第二个值被赋值给第二个元素，依此类推。赋空值是被允许的（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。

索引值表示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。第一个元素的索引值应为0，且该索引值不应超出子类型设定的长度限制。如果在赋值符号右边索引表示的元素值没有被定义，那么将会导致语义或运行错误。如果在赋值符号左边的一个索引操作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给该元素，并同时创建所有比实际的索引值小的带索引值元素，但并不对这些元素赋值，使这些元素的值为未定义。仅在中间状态（Transient State）允许未定义元素（而值仍就是不可见的）。发送带有未定义元素的一个**record of**类型值将会导致一个动态测试例错误。

例2：

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

MyVar := MyRecordVar[0]; // record of类型值中的第一个元素赋值给了MyVar

// 也允许索引值在赋值符号的左边：
MyRecordVar[1] := MyVar; // MyVar被赋给了第二个元素

// 以及下列两个赋值形式

MyRecordVar := { 0, 1, -, 2, omit };
MyRecordVar[6] := 6;

// 将导致{ 0, 1, <unchanged>, 2, <undefined>, <undefined>, 6 }；
// 注意，如果第三个元素以前没有被赋值，那么它仍是未定义的
// 而且，第六个元素（索引值为5）在这次赋值之前没有被赋值过。
```

注意： 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**类型值成为可能，例如，下面的函数翻转了一个**record of**类型值的元素：

```
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;
}
```

嵌套的**record of**和**set of**类型会导致一个类似多维数组的一个数据结构（见章节6.5）。

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相似的属性：
var MyRecordOfType myRecordOfArray;
// 且对一个特定元素的引用可以看起来如下面的格式（第三个MyBasicRecordOfType构造的第二个元素的值）
myRecordOfArray [2][1] := 1;
```

6.3.4 枚举类和值

TTCN-3支持枚举类型（**enumerated**）。枚举类型用于对只采用值的不同命名集的类型进行建模，每个枚举应该有一个标识符。对枚举类型的操作应该仅使用这些标识符，且仅限于赋值、等价和排序操作符。枚举标识符应该对该枚举类型来说是唯一的（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，且不会在相同范围层次关系分支中同层或低层中被用做局部或全局可见性的标识符（见章节5.4.0中的范围单位）。

例1：

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

type integer Monday;
// 因为类型的名字具有局部或全局可见性，所以这个定义是非法的。

type enumerated MySecondEnumType {
    Saturday, Sunday, Monday
};
// 因为该定义在另一个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
};
```

```
// 因为该定义在另一个结构化类型中使用枚举标识符Monday作为该结构化类型一个字段的标识符，所以它是合法的。

type record MyNewRecordType {
    MyFirstEnumType firstField,
    integer         secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// 通过MyNewRecordType的firstField元素隐式地引用MyFirstEnumType

const integer Monday := 7
// 因为该定义对相同范围单位中的不同TTCN-3对象重复使用枚举标识符firstField，所以它是不合法的。
```

每个枚举可以选择性地在其枚举名字后面的括号中定义一个分配给它的整型值，在一个**enumerated**类型内部，每个元素分配的整型数应该是不同的。对于没有分配整型值的枚举，系统按文本顺序陆续关联一个整型数，从左边、以0为开始，步长为1，同时跳过任一手工分配值占用的数字，这些值仅为系统用来允许关系操作符的使用。

注意1： 整型值也可以被系统用来编码/解码枚举值，不过这超出了本文讨论范围（除了TTCN-3允许的把编码属性关联到TTCN-3条目中去的情况）。

对一个**enumerated**类型的任意实例化或值引用来说，应隐式地或显式地引用给定类型。

注意2： 如果枚举类型是一个用户定义的结构化类型的一个元素，那么在值赋值、实例化等过程中，通过该元素隐式地引用此枚举类型（即通过该元素的标识符或在一个值列表表示法中该值的位置）。

例2：

```
// MyFirstEnumType和MySecondEnumType的有效实例化可以是
var MyFirstEnumType Today := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// 但是，因为两个枚举类型是不兼容的，所以下面的语句是非法的。
Today := Tomorrow
```

6.3.5 联合类型

6.3.5.0 概要

TTCN-3支持联合类型（**union**）类型，**union**类型是字段的汇集，这些字段每个都由一个标识符来标识，**union**类型在对采用有限个已知类型之一的一个结构化类型建模的时候很有用。Union types are useful to model a structure which can take one of a finite number of known types.

例：

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

// MyUnionType的一个有效实例化可以是
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;       // 使用引用字段的值表示。注意这种表示法使得给出的字段就是所选字段。
oneYearOlder := {number := age.number+1};

ageInMonths := age.number × 12;
```

用于设置值的值列表表示法不应该用于**union**类型的值。

6.3.5.1 对union类型字段的引用

应使用点号来引用**union**类型（见章节6.3.1.1）。

例：

```
MyVar5 := MyUnion1.myChoice1;
// 如果一个union类型嵌套在另外一个类型中，那么对它的引用可以看起来象如下格式
MyVar6 := MyRecord1.myElement1.myChoice2;
// 注意，被引用的带有标识符为myChoice2的字段的union类型嵌套在一个record类型中。
```

6.3.5.2 可选性和联合类型

union类型不允许使用可选字段（Optional fields），这就意味着关键字不会与**union**类型一起被使用。

6.4 任意类型

特殊类型**anytype**类型被定义为一个TTCN-3模块中所有*已知类型（known types）*的集合的简写，术语已知类型的定义在章节3.1中给出。

anytype类型的字段名应由相应的类型名字唯一标识。

例：

```
// 类型的一个有效用法可以是
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

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

MyVarThree := MyVarOne × 12;
```

对一个模块来说**anytype**是在本地定义的，（与其他预定义类型一样）不能直接由另外一个模块所引入。然而，**anytype**类型的一个用户定义类型是可以被另一个模块引入的，这就使得该模块的所有类型都被引入。

注意： **anytype**类型的用户定义类型“包含”声明该用户定义类型的模块引入的所有类型，引入这样一个个用户定义类型到一个模块中去可能会引起副作用，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。

例1：

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

数组的维数应该用一个结果为整型值的常数表达式来描述，也可以使用范围来描述，在后一种情况下，该范围上下界的值定义了该数组元素标号的上下界值。

例2：

```
var integer MyArray3[1 .. 5];   // 例示了一个有5个元素的整型数组，其元素标号为1到5
MyArray3[1] := 10;              // 最小标号lowest index
MyArray3[5] := 50;              // 最高标号highest index

var integer MyArray4[1 .. 5][2 .. 3 ];  // 例示了一个有5×2个元素的二维整型数组，标号为(1,2)到(5,3)
```

数组元素的值应该与相应的变量声明兼容，这些值可以使用值列表表示法对它们分别进行赋值，或者使用索引表示法或多次使用值列表表示法进行赋值。当使用值列表表示法的时候，该列表的第一个值被赋给了数组的第一个元素（该元素的索引值为0），第二个值被赋给了第二个元素，依词类推。不予考虑赋值的元素因该在列表中显式地跳过或省略。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。

例3：

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

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

MyArray4:= {{1, 2, 3, 4, 5}, {11, 12, 13, 14, 15}}
```

注意： 使用多维数据结构的一个替换方式就是使用record、record of、set或set of类型。

例4：

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

6.6 递归类型

可应用的TTCN-3类型定义可以是递归的，然而，用户应该确保所有的类型递归是可解析的，且不会发生无限递归。

6.7 类型兼容性

6.7.0 概要

通常，在赋值、实例化和比较时，TTCN-3要求值的类型兼容。

为了本节论述方便，把值“b”称为要赋予、作为参数传递等情况的实际值，类型“B”被称为值“b”的类型，类型“A”被称为要获得的实际值“b”的值类型定义。

6.7.1 非结构化类型的类型兼容性

对于非结构化变量、常量、模板等，如果类型“B”解析为与类型“A”相同的源类型且不违反对类型“A”的子类型定义（例如范围、长度限制）时，值“b”与类型A是兼容的。

例：

```
// 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x的子类型定义

x := 20; // 是一个有效赋值
y := x;
// 是一个无效赋值，因为x的值超出了MyInteger的值域

x := 5; // 是一个有效赋值
```

```
y := x;
// 是一个有效赋值，因为x的值在MyInteger的值域内
```

6.7.2 结构化类型的类型兼容性

6.7.2.0 概要

在结构化类型的情况下（**enumerated**类型除外），如果类型“B”的有效值与类型“A”兼容，类型“B”的一个值“b”与类型“A”兼容，在这种情况下允许赋值、实例化和比较。

6.7.2.1 枚举类型的兼容性

枚举类型与其它基本类型或结构化类型从不兼容（也就是说对于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

对于**record**类型，如果在定义的文本顺序上字段的数目、类型和可选性与值结构是相同的，且值“b”的每个已有字段与类型“A”的相应字段的类型兼容，那么这些有效值结构是相互兼容的。值“b”的每个字段的值被赋值给类型“A”的相应字段。

例1：

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

type record BType {
    integer         a   optional,
    integer (0..10) b   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 {     // 带有一个额外字段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};

// Then

MyVarA := MyVarB;   // 是一个有效的赋值，MyVarA的值是( a := <undefined>, b:= 2, c:= true)
MyVarC := MyVarB;   // 是一个有效赋值，MyVarC的值是( 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;   // 因为字段数目不匹配，所以不是一个有效赋值
```

```
MyVarC := { d:= 20 };// MyVarC的实际值是{ 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'的子类型定义
```

对于**record of**类型和数组类型，如果它们的成分类型是兼容的，且类型“B”的值“b”不违反**record of**类型的长度限定的子类型或类型“A”数组的维数，其有效的值结构是可兼容的。值“b”元素的值应该顺序赋值给类型“A”的实例，其中包括未定义的元素。

record of类型和单维数组类型的有效值结构与**record**类型的兼容，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”的元素数目精确相同，那么**record of**类型和单维数组类型与**record**类型是兼容的。在确定兼容性的时候，**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的时候总是被包括在内）。使用**record of**类型元素值或数组对一个**record**类型实例的赋值应该按照相应**record**类型定义的文本顺序进行，未定义元素也包括在内。如果一个带有未定义值的元素被赋给该**record**类型的一个可选字段，将会引起对该可选元素的省略。把一个带有未定义值的元素赋值给一个**record**类型的必要元素将导致错误。

注意： 如果**record of**类型没有长度限制或已有的长度限制超出了相比较的**record**类型元素数目，且任意该**record of**类型定义的元素的索引小于或等于**record**类型元素数目的最小值，那么总是要履行兼容性要求。

如果一个**record**类型没有违反**record of**类型的长度限制或数组的维数大于等于该**record**类型元素数目，那么这个**record**类型值总是可以被赋值给该**record of**类型实例或单维数组。应该以未定义值给**record**类型值中遗漏的可选元素赋值。

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，且没有违反子类型定义。

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// 是一个无效赋值，因为Htype的必要字段“c”没有接到赋值。
```

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，**set**类型和**set of**类型使用与**record**和**record of**类型相同的兼容性规则。

注意： 这就暗示了尽管不知道元素的发送和接收顺序，在为**set**类型决定类型兼容性的时候，类型定义中字段的文本顺序是决定性的因素。

例：

```
// 给出
type set FType {
    integer a   optional,
    integer b   optional,
    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;   // 是一个无效赋值，因为MyVarA是一个record类型
```

6.7.2.4 子结构的兼容性

为结构化类型兼容性在本节中定义的规则对于这些类型的子类型来说也是有效的，即子结构的等价。

例：

```
// 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 成分类型的类型兼容性

如果成分类型“B”的定义包含成分类型“A”的定义，则“B”的一个成分引用“b”与成分类型“A”兼容。这就意味着“B”包含至少与“A”相同的端口、变量和定时器实例以及常量声明，对于端口、变量和定时器实例来说，它们的类型和标识符都应该是相同的。

如果成分类型“B”的一个成分引用“b”与成分类型“A”兼容，则该成分引用“b”可以被赋值给一个类型为成分类型“A”的变量“a”。

6.7.4 通信操作的类型兼容性

通信操作（见章节23）**send**、**receive**、**trigger**、**call**、**getcall**、**reply**、**getreply**和**raise**是较弱的类型兼容性规则的例外，它们要求强类型机制（Strong typing）。用于这些操作的值的类型或直接用作参数的模板必须也明确地定义在相关联的端口类型定义中。在**receive**或**trigger**操作中，也使用强类型机制来来存储接收到的值、地址或成分引用。

6.7.5 类型变换

如果需要把一个类型的值变换为另一个非相同源类型派生的类型的值时，将使用附录C中定义的预定义转换函数或用户定义的函数。

例：

```
// 使用预定义函数int2hex把一个整型值转换为一个十六进制值
MyHstring := int2hex(123, 4);
```

7 模块（Modules）

7.0 概要

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测试套或仅仅是一个库，一个模块由定义部分（可选的）和一个模块控制部分（可选的）组成。

注意： 术语“测试套”与一个包含测试例和一个控制部分的完整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，后接一个可选的对象标识符。

注意1： 模块标识符是模块非正式的文本名。

注意2： 模块名可以仅在对象标识符部分不同，不过在这种情况下，因为标识符的前缀不能解析这种名称冲突，所以在引入的时候应谨慎以避免名称冲突（见章节7.5.8）。

7.2 模块参数

7.2.0 概要

模块（**module**）参数列表定义了由测试环境在运行时提供的一个值的集合，在测试执行时，将应按照常量来对待这些值。通过一对紧接在关键字**modulepar**之后的花括号中列出模块参数的标识符和类型来声明模块参数，模块应该仅在模块的定义部分中声明，允许模块参数声明的多次出现，但是每个参数只能声明一次（即不允许模块参数的重复定义）。

例：

```
module MyModulewithParameters
{
    modulepar { integer TS_Par0, TS_Par1; boolean TS_Par2 };
    :
    template MyType Mytemplate
        {
        field   TS_Par3
        };
    modulepar { hexstring TS_Par3 };
}
```

注意： 这提供与TTCN-2中给测试套提供PICS和PIXIT值的测试套参数相似的功能性。

7.2.1 模块参数的默认值

描述模块参数的默认值（default values）是允许的，这应该在模块参数列表中使用赋值来完成。一个默认值可以只是一个常量值（literal value），且仅在该参数声明的地方赋值。如果测试系统对给定的参数不提供一个实际的运行值，则应该在测试执行期间使用默认值，否则的话由测试系统提供该实际值。

例：

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

7.3 模块定义部分

7.3.0 概要

模块定义部分描述该模块的顶层（top-level）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范围规则在章节5.3中给出，可以在TTCN-3模块中定义的那些语言元素列在表1中，模块定义可以被其他模块引入。

例：

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

动态语言元素（如**var**或**timer**）的声明应该仅在控制部分、测试例、函数、可选步或成分类型中进行。

注意： 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。但是，在一个测试成分中定义的变量可以被在该成分上运行的所有的测试例、函数等所使用，且在控制部分中定义的变量有能力使它们的值独立于测试例执行。

7.3.1 组定义

在模块定义部分中，定义可以被汇集在所谓的组（Groups）中。可以在允许单个声明的任何地方描述一个声明组。组可以被嵌套，即组可能包含其它组，这就允许测试套描述符在其它的概念中构建测试数据或描述测试行为的函数的类集。

如果需要的话，使用分组来提高可读性和为测试套增加逻辑结构。除了在组标识符的上下文中和通过使用关联的**with**语句给出的属性的情况之外，组和嵌套的组没有范围，这就意味着：

- 整个模块的组标识符不必唯一，但是在相同层次上所有组标识符应该是唯一的，且层次中低层的子组不应该与高层的组具有相同的组名，如果必要的话，应使用点号来唯一标识组层次中的子组，例如对一个特定子组的引入。
- 属性的重写规则（Overriding rules）在章节28.4中给出。

例：

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

// 一个可选步组
group MyStepLibrary {
    group MyGroup1 {    // 具有与上面带有定义的子组相同名字的子组
        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，描述实际测试例的执行顺序（可能是重复的），应该在模块的定义部分定义测试例，在控制部分调用该测试例。

例：

```
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(); // 测试例的顺序执行
        execute MyTestCase2();
        :
    }
}
```

7.5 从模块引入

7.5.0 概要

可以使用**import**语句重复使用不同模块中说明的定义。TTCN-3没有明确的输出构架（export construct），因此，默认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，但是不应在控制部分使用该语句。

如果在**import**语句中对象标识符作为模块名（被引入该定义的模块的名字）被提供，那么应使用这个对象标识符去标识该正确的模块。

从一个模块中引入的所有定义应该仅在一个**import**语句中被引用。

如果一个引入的定义具有（使用**with**语句定义的）属性，那么这些属性也应该被引入，改变引入定义属性的机制在章节28.6进行说明。

注意： 如果模块具有全局属性，那么这些属性与不带有这些属性的定义相关联。

例：

```
module MyModuleA
{   // 这个模块包含定义和引入的定义
    :
    const integer MyConstant := 1;
    import from MyModuleB all;  // 引入的定义的范围对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、特性、常量、外部常量、数据模板、特性模板、函数、外部函数、可选步和测试例。每个定义有一个*名字*（*name*，定义该定义的标识符，如一个函数名）、一个*细节描述*（*specification*，如一个类型描述或一个函数的特性），在函数、可选步和测试例的情况中，还包括一个相关的*行为描述*（*behaviour description*）。

例：

	Name	Specification	Behaviour description

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

	Specification	Name	Specification

type	**record**	MyRecordType	{
			field1 MyType4,
			field2 integer
			}

	Specification	Name	Specification

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、可选步或测试例被引入的时候，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影响，因此也不在后续的说明中考虑它们。

可引入定义的说明部分包含本地定义（*local definitions*，如结构化类型定义的字段名或枚举类型值）和引用的定义（*referenced definitions*，对类型定义、模板、常量或模块参数的引用），对于上面的例子，这意味着：

	名字	本地定义	引用的定义
function	MyFunction	MyPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType3, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

注意1： 本地定义一栏指的是在可引入定义中新定义的标识符，赋给可引入定义的字段的值也可以作为本地定义来加以考虑（如在模板定义中），但是它们对说明引入机制并不重要。

注意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。

被引用的定义也是可引入的定义，这也就是说一个被引用定义的源可能再次被构造成一个名字和一个描述部分，且这个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的可引入定义中递归地构造。

TTCN-3引入机制与在可引入定义的说明部分中使用的本地定义和引入定义相关，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：可引入定义可能的本地定义和引用定义

可引入的定义 Importable Definition	可能的本地定义 Possible Local Definitions	可能的引用定义 Possible Referenced Definitions
模块参数		模块参数类型
用户定义类型（对于所有的）	参数名	参数类型
• 枚举类型	具体值	
• 结构化类型	字段名	字段类型
• 端口类型		消息类型，特性
• 成分类型	常量名、变量名、定时器名和端口名	常量类型，变量类型，端口类型
特征	参数名	参数类型，返回类型，例外（exceptions）类型
常量		常量类型
外部常量		常量类型
数据模板	参数名	模板类型，参数类型，常量，模块参数，函数
特性模板		特性定义，常量，模块参数，函数
函数	参数名	参数类型，返回类型，成分类型（**runs on**-子句）
外部函数	参数名	参数类型，返回类型
可选步	参数名	参数类型，成分类型（**runs on**-子句）
测试例	参数名	参数类型，成分类型（**runs on**-和**system**-子句）

TTCN-3引入机制区分引入定义中*引用定义的标识符*和*引用定义使用的必要信息*，对于引用定义的使用，不要求引用定义的标识符，因此不是自动引入。

7.5.2 使用引入操作的规则

使用引入操作时应该应用如下规则：

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) 定义的名字和所有局部定义一起被引入。

注意1： 一个本地定义，如一个用户定义记录类型的一个字段名，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，而不能出了这个范围。

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。

注意2： **import**语句是传递的，如，如果一个模块A从使用模块C中定义的一个类型引用的模块B中引入一个定义，则该类型使用所需的相应信息自动被引入到模块A中。

e) 默认地，不自动引入引用定义的标识符。如果希望隐式引入该引用定义的标识符，应使用**recursive**命令（见章节7.5.3）。

注意3： 如果在使用默认引入机制时（如变量实例化）希望在引入模块中使用被引用的定义，应该从它的源模块显式地引入。

f) 引入一个函数、可选步或测试例时，相应的行为说明和行为说明内使用的所有定义对于引入模块来说仍是不可见的。

例：

```
module ModuleONE {

    modulepar {
        integer ModPar1, ModPar2 := 7
    }

    type record RecordType_T1 {
        integer Field1_T1,
        boolean Field2_T1
    }

    type record RecordType_T2 {
        MyRecordType_T1 Field1_T2,  // RecordType_T1的使用
        MyRecordType_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中只有名字Template_T2和TempPar_T2是可见的。
    // 注意，标识符TempPar_T2只能被用在Template_T2的上下文中，如提供一个实参值时。
    // 为引用定义RecordType_T2、RecordType_T1、Field1_T2、Field2_T2、Field3_T3、MyConst和ModPar1
    // 引入Template_T2使用所需的所有信息（如用于类型检查目的信息），但是它们的标识符在ModuleTWO中是可见的。
    // 这意味着，不可能在不显式引入ModuleTWO中类型RecordType_T1或RecordType_T2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可以被用做为一个整型的常量

} // 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的一个模块参数的引用
        port MyPortType MyPort
    }

    function MyFunction () return integer {
        return MyConst      // 返回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的一个模块参数的引用
```

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

    } // 结束测试例MyTestCase

} // end ModuleTHREE


module ModuleFOUR {

    import from ModuleTHREE {
        testcase MyTestCase
    }

    // 只有名字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行为部分使用的所有定义对于ModuleFOUR使用者来说仍是隐藏的。


} // 结束ModuleFOUR
```

7.5.3 递归引入Recursive import

TTCN-3默认的引入机制引入引用定义而不需它们的标识符，这就意味着在引入模块中不能使用引用定义，例如声明一个变量或在一个端口上发送。尽管这个默认引入机制避免引入模块名字空间的混乱，但是在一些情况下，和标识符一起引入所有引用定义还是被期望的。在TTCN-3中，关键字**recursive**提供这个特性。

在使用带有**recursive**命令的**import**操作时，应该遵循以下规则：

a) 章节7.5.2中的规则a)、b)、c)和f)仍有效。

b) 递归的引入定义与所有引用定义一起被引入，即所有引用定义的标识符在引入模块中是可见的和可用的。

注意1： 递归的引入语句在源模块中是传递的，例如，一个模块A从使用类型T的模块B（类型T也是定义在模块B中）中递归地引入一个定义，那么类型T自动地被引入到模块A中。

注意2： 递归的引入语句在模块的边界间是不传递的，这就是说，如果模块A从使用类型T（类型T在模块C中定义）的模块B中递归地引入一个定义，那么类型T不能自动地被引入到模块A中去，类型T必须被显式地从模块C中引入，既从它的源模块中引入。

例：

```
// 模块ModuleONE和ModuleTHREE与章节7.5.2中例子定义的一样
module ModuleFIVE {

    import from ModuleONE recursive {
        template Template_T2
    }

    // Template_T2的递归引入将从ModuleONE引入RecordType_T2、RecordType_T1、MyConst和ModPar1的定义。
    // 由于类型RecordType_T2和RecordType_T1的引入，这些类型的字段名Field1_T1、Field2_T1、Field1_T2、
    // Field2_T2和Field3_T3在ModuleFIVE中将变为可见的。

} // 结束模块ModuleFIVE

module ModuleSIX {

    import from ModuleTHREE recursive {
        testcase MyTestCase
    }

    // 如果该模块不包括从ModuleONE中递归引入RecordType_T2的一个引入语句将导致一个错误。
    // 从ModuleTHREE对MyTestCase的递归引入需要从RecordType_T2和MyCompType的源模块中对它们进行递归引入。
    // RecordType_T2的源模块是模块ModuleONE。
    // 尽管MyCompType的源模块是模块ModuleTHREE，但是对它的递归引入仍将导致一个错误，因为这个定义需要
```

```
        // ModuleONE中的定义。

    } // 结束ModuleSIX

    module ModuleSEVEN {

        import from ModuleONE recursive {
            modulepar ModPar2;
            type RecordType_T2
        }

        // 从ModuleONE引入ModPar2、RecordType_T2和RecordType_T1（RecordType_T1被RecordType_T2使用）。
        // 通过递归引入，RecordType_T2和RecordType_T1的字段名Field1_T1、Field2_T1、Field1_T2、Field2_T2
        // Field3_T3将变为可见的。

        import from ModuleTHREE recursive {
            testcase MyTestCase
        }

        // 从ModuleTHREE引入MyTestCase、MyCompType（被MyTestCase使用）和MyPortType（被MyCompType使用）。
        // 通过对MyTestCase和MyComptype的递归引入，标识符MyPar（在MyTestCase中定义）、MyComponentVar和
        // MyPort（都在MyCompType中定义）变为可见的。
        // MyTestCase递归引入所需的来自ModuleOne的定义通过前面的import语句被递归引入。
`

    } // 结束ModuleSEVEN

    module ModuleEIGHT {

        import from ModuleONE {
            modulepar ModPar2;
            type RecordType_T2
        }

        import from ModuleTHREE recursive {
            testcase MyTestCase
        }

        // 将导致一个错误，因为为了对MyTestCase的递归引入，也需要从ModuleONE完全引入类型RecordType_T1，或者，
        // 换句话说，需要递归地引入RecordType_T2。

    } // 结束ModuleEIGHT
```

7.5.4 引入单个定义

可以引入单个定义。

例：

```
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，那么应该没有其它的引入格式用于相同的**import**语句。

例1：

```
import from MyModule all;
```

如果有些声明不希望被引入，那么它们的种类和标识符应该列在关键字**except**后一对花括号中的例外列表中。

例2：

```
import from MyModule all except {
    type MyType3, MyType5
        // 从import语句中除去类型声明MyType3和MyType5
        // 但是引入MyModule的所有其它声明
}
```

也允许在例外列表中使用关键字**all**，这就排除了来自引入语句的相同种类的所有声明。

例3：

```
import from MyModule all except {
    type MyType3, MyType5;  // 从import语句中排除了两个类型
    template all            // 从import语句中排除了MyModule中声明的所有模板
}
```

7.5.6 引入组

可以引入定义组（group）。

例1：

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

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

TTCN-3的组只是用于结构化的目的，而不是范围单元，因此不允许直接引入子组（即定义在另一个组中的组），即不通过子组所嵌套的组就直接引入子组。如果应引入的子组名与相同模块中另一个子组名相同（见章节7.3.1），应使用点号来唯一标识要被引入的子组。

如果不愿引入一个组的一些定义，应在关键字**except**后面花括号中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import语句中除去类型定义MyType3和MyType5，但引入MyGroup的所有其它定义。
    }
}
```

允许在例外列表中使用关键字**all**，这从import语句排除了相同中列的所有定义。

例3：

```
import from MyModule {
    group MyGroup except {
        type MyType3, MyType5;  // 从import语句排除了两个类型
        template all            // 从import语句排除了排除了MyGroup中定义的所有模板
    }
}
```

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。

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import语句中排除一个种类的一些声明，那么这些声明使用的标识符应列在关键字**except**之后。

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Mymodule中定义的所有模板
}
```

7.5.8 处理引入中的名字冲突

所有的TTCN-3模块应该有它们自己的名字空间，在各自的名字空间里所有的定义应该被唯一标识。由于引入可能引起名字冲突，例如来自不同模块的引入、组引入或递归定义引入。应该使用被引入定义的模块的标识符作为（引起名字冲突的）引入定义的前缀来解决名字冲突，前缀和标识符使用点号（.）隔开。

在没有多义性的情况下，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，该模块（当前模块）的标识符也可用做该定义标识符的前缀。

例：

```
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
         :
    }
}
```

注意： 即便在不同模块中带有相同名字的定义的实际定义是相同的，也总是假设在不同模块中带有相同名字的定义是不同的。例如，引入一个已经在本地定义了的类型，甚至使用了相同的名字，会导致在该模块中两个不同的有效类型。

7.5.9 处理相同定义的多个引用

在单个定义、定义组、相同种类定义等上**import**的使用可能导致对相同定义的多次引用，这样的情况应该通过系统去解决，且定义应该仅被引入一次。

注意： 解决这样的多义性的机制（如重写（overwriting）和给用户发送警告）不在本文研究范围之内，应由TTCN-3工具提供。

所有**import**语句和引入语句中的定义应该按照它们出现的顺序一个接一个地分别加以考虑，有必要指出的是，**except**语句通常不从被引入中排除所列出的定义。相同种类定义的所有引入定义语句可以看作是与单个定义标识符的等价列表的简写表示，**except**语句仅从这个单个的列表中排除定义。

例：

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

```
        type MyType3                    // 明确引入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中，应使用语言说明来指示被引入定义来源的语言（如模块（module）、包（package）、库（library）或者甚至是文件（file），可能带有 版本号），由关键字**language**和一个随后的表示语言的文本声明组成。当从一个与引入模块相同版本的TTCN-3模块中引入时，该语言说明的使用是可选的，说明ASN.1模块的语言标识符在附录D.1中给出。

例：

```
import from MyASN1Module language "ASN.1:1997" {
    type MyASN1Type
}
```

注意： 设计引入机制来允许重复使用来自其它TTCN-3或ASN.1模块的TTCN-3和ASN.1定义。引入使用其他语言（如SDL包）书写的说明中的定义的规则可以遵循TTCN-3规则，或可能是必须分别定义的规则。对非TTCN-3和ASN.1语言的引入规则不包括在本文之中。

8 测试配置

8.0 概要

TTCN-3允许对并发测试配置（或简称配置）的（动态）描述，一个配置由一个带有良好定义的通信端口的互连测试成分集合和定义该测试系统边界的明确的测试系统接口组成。

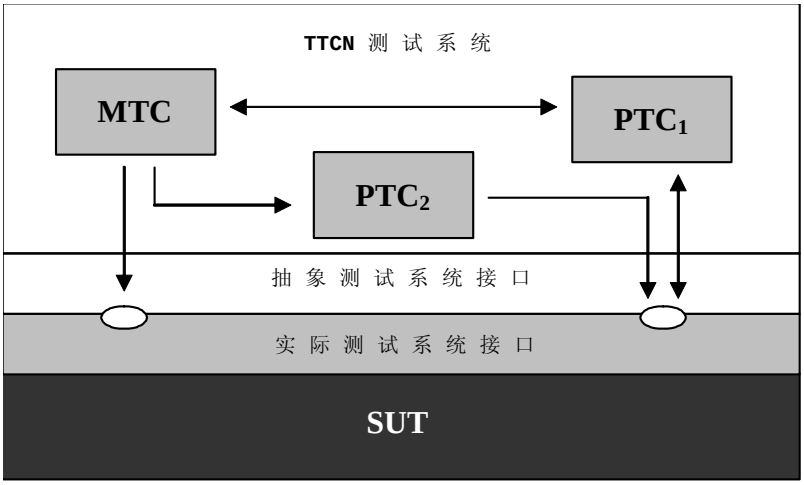


图 3：一个典型的TTCN-3测试配置的概念化视图

在每个配置中应该有一个（且仅有一个）主测试成分（MTC），非主测试成分被称为并行测试成分（PTCs）。MTC应该在每个测试例执行开始时由系统创建，而测试例主体中定义的行为应该在该成分上执行。在一个测试例执行期间，通过显式使用**create**操作来动态创建其它成分。

应在MTC终止时结束测试例执行，平等对待所有其它的PTCs，即在它们之间没有明确的层次关系，一个PTC既不能终止其它PTCs也不能终止MTC。当MTC终止时，测试系统必须停止所有测试例结束时未被终止的PTCs。

测试成分之间以及测试成分和测试系统接口之间的通信通过通信端口实现（见章节8.1）。

用关键字**component**和**port**指示的测试成分类型和端口类型应该在模块定义部分中定义，成分的实际配置以及它们之间的连接通过在其测试例行为中执行**create**和**connect**操作来实现，利用**map**操作将成分端口连接到测试系统接口的端口上（见章节22.2）。

8.1 端口通信模型

测试成分通过它们的端口连接，也就是说测试成分之间以及一个成分和测试系统接口之间的连接是面向端口的。每个端口被建模为一个无限的先进先出（FIFO）队列，这个队列用于存储进来的消息或者是过程调用直至拥有该端口的成分处理它们。

注意： 原则上TTCN-3端口是无限的，然而在实际的测试系统中他们可能会溢出，这时应该以一个测试例错误来对待它（见章节25.2.1）。

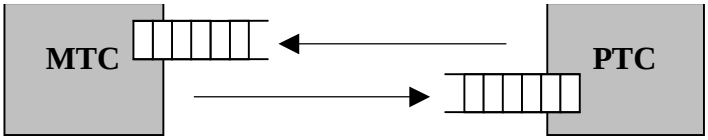


图 4：TTCN-3通信端口模型

8.2 连接上的限制

TTCN-3连接是端口到端口（port-to-port）和端口到测试系统接口（port-to-test system interface）的连接（见表5）。对于一个成分可以维护的连接数目没有限制，也允许一对多的连接（例如表5 (g)或表5 (h)）。

TTCN-3不允许以下连接：

- 一个成分A的一个端口不应该与该成分上的两个或两个以上的端口相连（图6 (a)和图6 (e)）。
- 一个成分A的一个端口不应该与成分B上的两个或两个以上的端口相连（见图6 (c)）。
- 一个成分A的一个端口与测试系统接口可能仅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)）。

因为TTCN-3允许动态配置和动态地址，所以不可能在编译时总是检查连接上的限制，应该在运行时进行检查，且在失败时引起一个测试例错误。

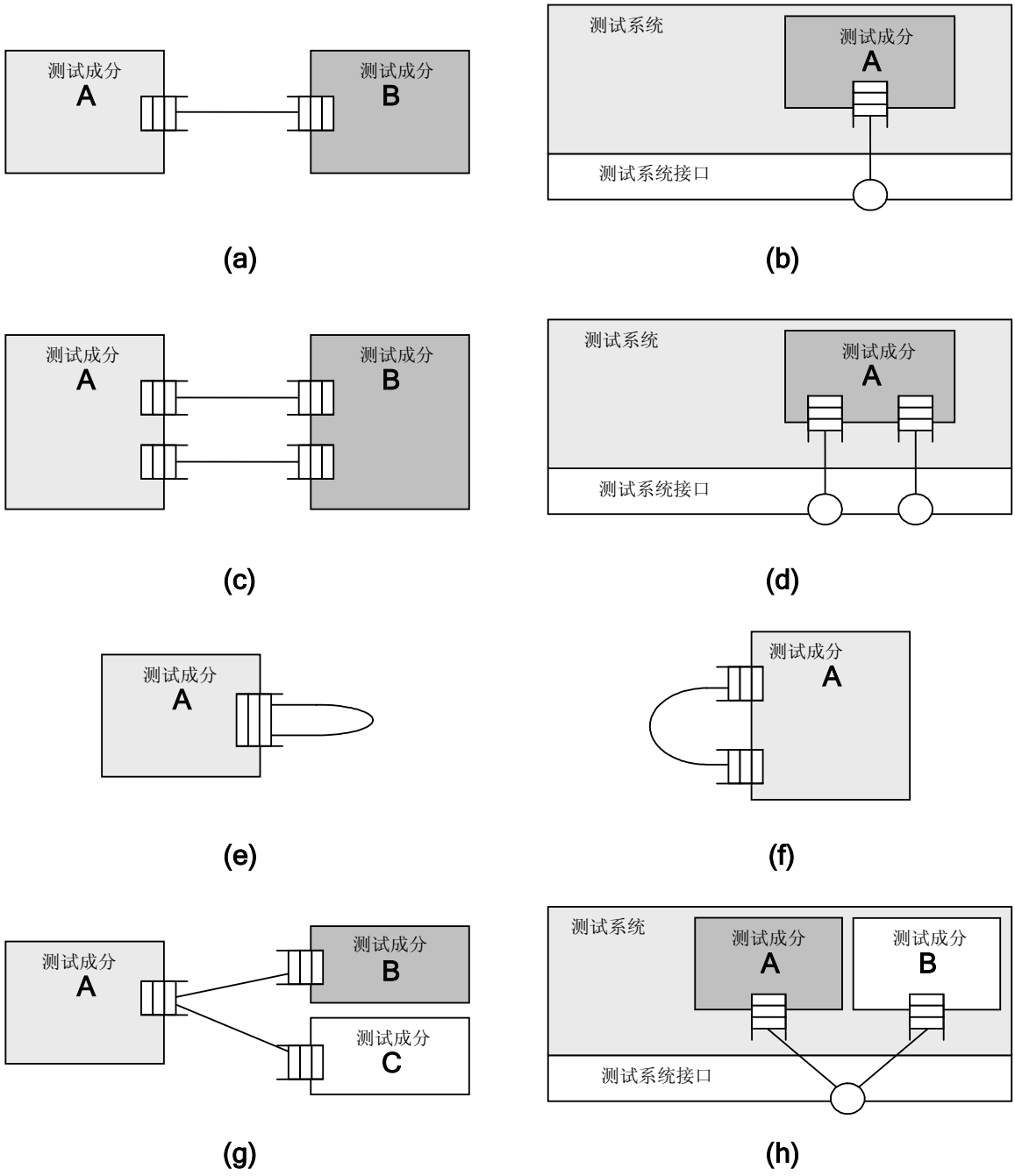

测试成分
A
测试成分
B
(a)
测试系统
测试成分
A
测试系统接口
(b)
测试成分
A
测试成分
B
(c)
测试系统
测试成分
A
测试系统接口
(d)
测试成分
A
(e)
测试成分
A
(f)
测试成分
A
测试成分
B
测试成分
C
(g)
测试系统
测试成分
A
测试成分
B
测试系统接口
(h)

图 5：允许的连接

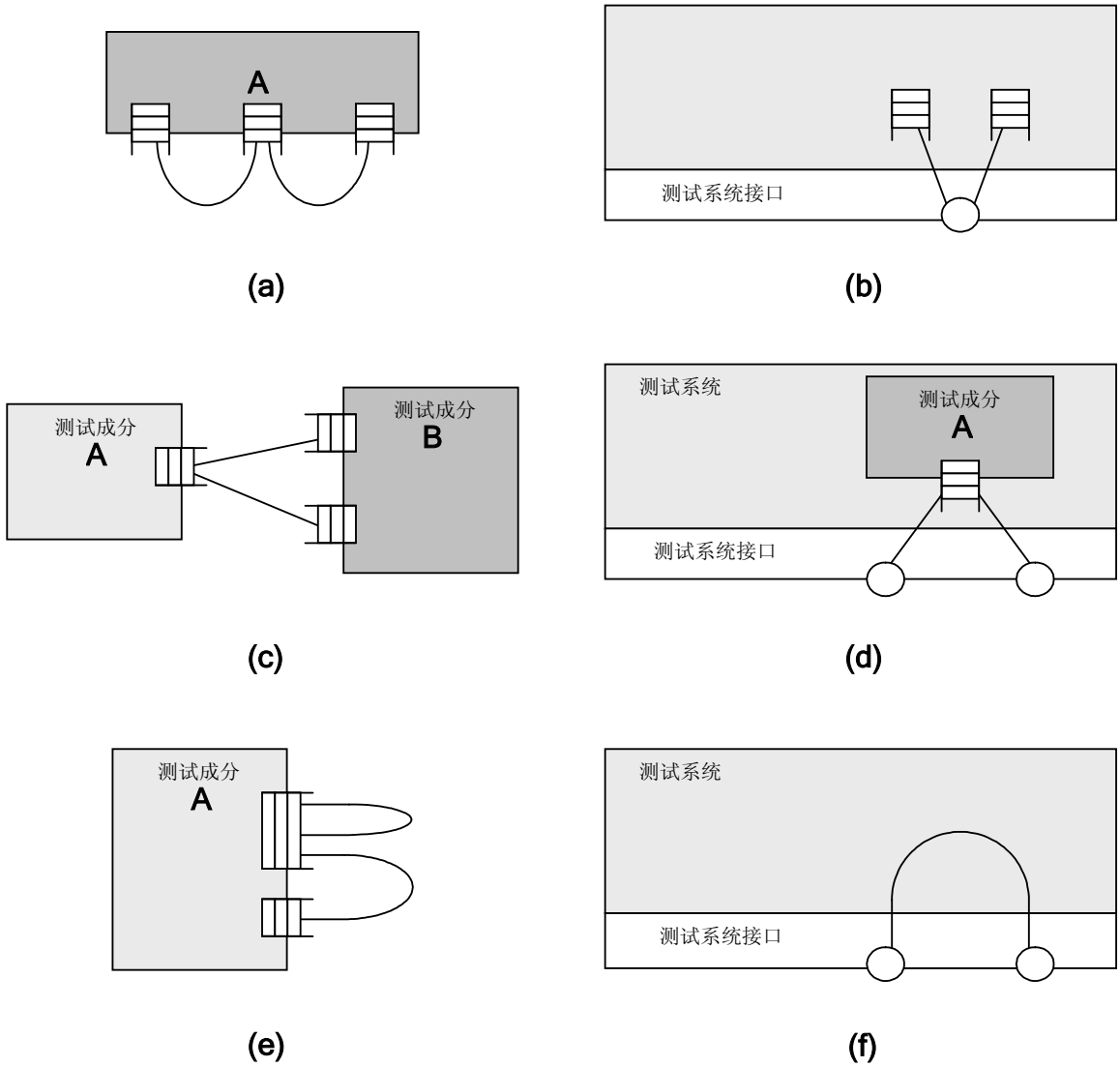


图 6：不允许的连接

8.3 抽象测试系统接口

TTCN-3用于测试实现，被测试的对象被认为是被测实现（Implementation Under Test，IUT）。IUT可以为测试提供直接接口，或者在被测对象是被测系统（System Under Test，SUT）时它可以是该系统的一部分。在本文中，术语SUT被用作是一个通用的概念，意味着SUT或IUT。

在一个实际的测试环境中，测试例需要与SUT进行通信。然而，对实际物理连接的说明超出了TTCN-3的范围，作为替代，一个良好定义的（但是是抽象的）测试系统接口应该与每个测试例相关联。一个测试系统接口定义与一个成分定义相同，即它是一个所有可能通信端口的集合，通过这些端口测试例被连接到SUT。

测试系统接口静态地定义测试运行时与SUT连接的端口连接的数目和类型，但测试系统接口和TTCN-3测试成分之间的连接实际上是动态的，可以在一个测试运行时使用**map**和**unmap**操作对其进行修改（见章节22.2和22.3）。

8.4 定义通信端口类型

8.4.0 概要

端口便于测试成分之间和测试成分与测试系统接口之间的通信。

TTCN-3支持基于消息和基于过程的端口。每个端口应该定义为基于消息的或基于过程的（或者是章节8.4.1中描述的二者混合型），通过关键字**message**来标识基于消息的端口，而相关联端口类型定义中的关键字**procedure**则是用来标识基于过程的端口的。

端口是有方向的，它的方向通过关键字**in**（输入方向）、**out**（输出方向）和**inout**（输入/输出方向）来标识的。每个端口类型定义应该有一个或多个列表来指明所允许的（消息）类型集合和/或带有允许的通信方向的过程。

例1：

```
// 基于消息的端口，其上允许接收MsgType1和MsgType2类型，发送MsgType3类型，并可发送和接收任意整形值
type port MyMessagePortType message
{
    in      MsgType1, MsgType2;
    out     MsgType3;
    inout   integer
}

// 基于过程的端口，它允许过程Proc1、Proc2和Proc3的远程调用。
// 注意Proc1、Proc2和Proc3定义为过程特征（signatures）
type port MyProcedurePortType procedure
{
    out     Proc1, Proc2, Proc3
}
```

注意： 术语消息指的是模板和表达式的实际值定义的消息，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。

对一个与端口类型相关联的列表，使用关键字**all**来允许在该模块中定义的在该端口上传递的所有类型和过程特征。

例2：

```
// 基于消息的端口，它允许该端口上任意方向上传输所有嵌入类型和用户自定义类型的任何值。
type port MyAllMesssagesPortType message
{
    inout   all
}
```

8.4.1 混合型端口

把一个端口定义为基于消息的通信和基于过程的通信混合型的端口也是可能的，这种情况下使用关键字**mixed**表示。这就意味着混合型端口的列表也将是混合的，它包括了过程特征和类型。此时，关键字表示该模块中定义的所有类型和过程特征，在定义中也不分别说明。

```
// 混合型端口，定义了相同名字的一个基于消息的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integer涉及混合型端口中基于消息的部分，
// Proc1、Proc2、Proc3、Proc4和Proc5涉及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;
}

// 混合型端口，模块中定义的所有类型和所有过程特征都可以用在这个端口上与SUT或其他测试成分进行通信。

type port MyAllMixedPortType mixed
{
    inout   all
```

}

TTCN-3中的混合型端口实际是两类端口定义的缩写表示，也就是说，一个基于消息的端口和一个基于过程的端口具有相同名字。在运行时，通过通信操作来区分两类端口。

如果调用一个混合性端口的标识符，用于控制端口的操作（见章节23.5），即**start**、**stop**和**clear**应该在两个队列上（按照任意的顺序）都执行操作。

8.5 定义通信类型

8.5.0 概要

成分类型（**component**）定义了与一个成分相关联的端口。这些定义应该在模块的定义部分中进行定义，一个成分中的端口名字对该成分来说是本地的，也就是说另外的成分可以有相同名字的端口。但是，相同成分的所有端口名字应该是唯一的。一个成分的定义本身并不意味着该成分在这些端口上有任何连接。

注意： 在这个方面，TTCN-3与TTCN-2有区别。测试配置是静态的，而对测试成分、PCOs和ASPs的声明则隐含了在初始化测试例执行时他们的自动连接。

例：

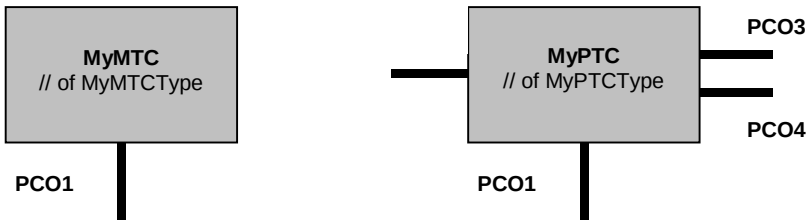


图 7： 典型的测试成分

```
type component MyMTCType
{
    port MyMessagePortType  PCO1
}

type component MyPTCType
{
    port MyMessagePortType      PCO1, PCO4;
    port MyProcedurePortType    PCO2;
    port MyAllMesssagesPortType PCO3
}
```

8.5.1 在一个成分中声明本地变量和定时器

可以声明一个特殊成分的本地常量、变量和定时器。

例：

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

这些声明对于该成分上运行的所有函数和可选步都是可见的，这应该使用关键字**runs on**来显式说明（见章节16）。

成分变量和定时器与该成分实例相关联，并遵循章节5.3定义的范围规则。一个成分的每个新实例也将有它自己的变量和定时器集合，就象成分定义中所描述的那样（如果说明了，则还包括任意初始化的值）。

注意： 当成分被用作测试系统接口时（见章节8.8），该成分不能使用成分中声明的任一常量、变量和定时器。

8.5.2 定义带有端口数组的成分

在成分类型定义中可以定义端口数组（另见章节22.9）。

例：

```
type component My3pcoCompType
{
    port MyMessageInterfaceType PCO[3]
    // 定义一个成分类型，它带有的数组由三个端口组成
}
```

8.6 SUT内部的编址实体

一个SUT可以由数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给SUT实体编址的一种数据类型。当地址数据类型与**to**、**from**和**sender**一起使用的时候，它只能用在映射到测试系统接口的端口的接收和发送操作中。**address**的实际数据表示由该测试套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**类型被作为TTCN-3说明中的一种开放式的类型）。这就允许独立于SUT任意特定的实际地址机制描述抽象测试例。

如果地址类型定义在一个模块内部，那么应该仅在该TTCN-3模块内部生成明确的SUT地址。如果该类型不是定义在一个该模块的内部，那么明确的SUT地址应该仅可以在消息中作为字段参数被传入或接收，或者作为远程过程调用的参数被传入或接收。

此外，可以使用特定类型**null**来指示一个未定义的地址，例如用于地址类型变量的初始化中。

例：

```
// 把整型与开放式类型——地址类型相关联
type integer address;
 :
// 用null初始化新的地址类型变量
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;
 :
// 发送模板MyResult的操作中接收地址的用法
PCO.send(MyResult) to MySUTentity;
 :
// 在接收一个确认模板操作中，接收地址的用法
PCO.receive(MyConfirmation) from MySUTentity;
```

8.7 成分引用

成分引用是对在一个测试例执行时被创建的测试成分的唯一引用。这个唯一引用在一个成分被创建时由测试系统产生，也就是说一个成分引用是**create**操作的结果（见章节22.1）。另外，成分引用由预定义操作**system**（返回标识测试系统接口的端口的成分引用）、**mtc**（返回MTC的成分引用）和**self**（返回调用**self**的成分的成分引用）。

在配置操作**connect**、**map**和**start**操作（见章节22）中使用成分引用来建立测试配置，而出于编址的目的，在连接到测试成分端口（而非测试系统接口）的通信操作的**from**、**to**和**sender**部分中使用成分引用（见章节23和图5）。

此外，特殊类型**null**可用来指示一个未定义的成分引用，例如用于处理成分引用的变量的初始化。

成分引用的实际数据表示方法应该由测试系统来进行外部解析，这也就允许独立于任意实际TTCN-3运行环境描述抽象测试例，换句话说，就是TTCN-3并不对有关测试成分处理和标识的一个测试系统实现加以限制。

注意： 一个成分引用包括成分类型信息。举个例子，这就意味着用于处理成分引用的变量在其声明中必须使用相符的成分类型名。

例：

```
// 一个成分类型定义
type component MyCompType {
    port PortTypeOne PCO1;
    port PortTypeTwo PCO2
}

// 为处理MyCompType类型成分的引用声明两个变量，并创建这个类型的一个成分
var MyCompType MyCompInst := MyCompType.create;

// 在配置操作中成分引用的用法
// 总是引用上面已经创建的成分
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self)); // self被作为参数传递给MyBehavior

// 在from-和to-子句中的成分引用的用法
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
// 的多个成分类型接收M1的值，且重新获得发送者。在这种情况下，可以使用如下方案：
 :
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);   // 一些结果从函数中获得
 :
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，因为成分类型定义和测试系统定义具有相同的形式（都是定义可能的连接点的端口集合），因此成分类型定义被用来定义测试系统接口。

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

通常，定义测试系统接口的一个成分类型引用应该与使用多个测试成分的每个测试例都相关联。当测试例启动执行时，测试系统接口的端口应该与MTC一起由该系统自动初始化，即从模块的控制部分调用该测试例。

返回测试系统接口的成分引用的操作是**system** ，这个操作被用于对测试系统端口进行编址。

例：

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

在测试例执行时初始化的唯一测试成分是MTC的情况下，测试系统接口不需要与测试例相关联。在这种情况下，与MTC相关联的成分类型定义隐式地定义了相应的测试系统接口。

9 常量声明

可以在模块的定义、成分类型定义、模块控制部分、测试例、函数和可选步（altsteps）中声明和使用常量。常量由关键字**const**来标识，应该在常量声明的位置为其赋值。

例1：

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

可以在模块内部给常量赋值，也可以在模块外部进行，后一种情况用关键字**external**来标识外部常量声明。

例2：

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

一个外部常量可以具有模块中已知类型之外的任意类型，即模块中定义的源类型（root type）或用户定义类型以及从其他模块中引入的类型之外的其他类型。从该类型到一个外部常量的外部表示的映射超出了本文讨论范围，如何给一个模块传入外部常量值的机制也超出了本文的讨论范围。

10 变量声明

变量用关键字**var**表示，可以在模块的控制部分、测试例、函数或是可选步中声明、使用变量。此外，还可以在成分类型定义中声明变量，并在测试例、可选步和使用这些给定成分类型的函数中使用它们。但是，不能在模块的定义部分声明或使用变量，也就是说TTCN-3不支持全局变量的概念。变量声明时可以给它赋一个的初始值，这个赋值是可选的。

例：

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

运行时使用未初始化的变量会导致测试例错误。

11 定时器声明

11.0 概要

可以在模块的控制部分、测试例、函数或是可选步中声明、使用定时器。此外，还可以在成分类型定义中声明定时器，并在测试例、可选步（和使用这些给定成分类型的函数中使用它们。声明一个定时器时，可以选

择给它赋一个默认持续时间值，如果没有再指定其他值，该定时器将使用该默认值开始计时。定时器的值是一个非负浮点数（即大于等于0.0），以秒为单位。

例：

```
timer MyTimer1 := 5E-3; // 声明定时器MyTimer1，默认值为5ms

timer MyTimer2; // 声明定时器MyTimer2，但没有赋默认值，将在运行时给该定时器赋值
```

除了声明单个的定时器外，还可以声明定时器数组。定时器数组中每个元素的持续时间值由一个值数组来赋值，值数组中的第一个元素赋值给定时器数组的第一个元素，以此类推。如果对定时器默认的持续时间赋值时希望跳过定时器数组的一些元素，则必须用表不使用的符号（"-"）明确地声明。

注意：这意味着定时器数组元素个数与值数组个数必须保持完全一致。

例：

```
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默认持续时间值

timer t_Mytimer2[5] := { 1.0, 2.0, 3.0, 4.0, - }    // 定时器数组的最后一个元素t_Mytimer2[4]没有被赋值
```

11.1 定时器做参数

定时器只能在函数和可选步中作为参数传递，传入函数和可选步的定时器在这些函数和可选步的行为定义中是已知的。

作为传参参数传递的定时器的使用与其他定时器一样，即它不必被声明。一个已经启动了的定时器也可以传递给函数和可选步，这时该定时器仍继续运行，也就是说它不会在暗中被中断。因此，被传入定时器的函数和可选步能够在其内部处理可能发生的超时事件。

例：

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

12 消息声明

TTCN-3的一个关键元素就是在测试配置中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SUT明确相关的消息，也可以是与特定测试配置相关的内部协调或控制消息。

注意：在TTCN-2中，这些消息是抽象服务元语ASPs（Abstract Service Primitives）、协议数据单元PDUs（Protocol Data Units）和协调消息（co-ordination messages）。而在TTCN-3核心语言中，消息没有任何语法和语义的区别，从这个意义上讲是通用的。

13 过程特征的声明

13.0 概要

在基于过程的通信中，过程特征（Procedure Signatures，简称特征）是必须的。 基于过程的通信可以用于测试系统（Test System）内部的通信，即测试成分之间或测试系统与SUT之间的通信。在后面的章节中，一个

过程可以在被测系统SUT中被调用（invoke，即测试系统执行这个调用）或在测试系统中被调用（invoke，即被测系统SUT执行这个调用）。对于所有使用到的过程，即用于测试成分之间通信的过程、被测系统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特征特性（**Procedure Signature**）。

13.1 阻塞的和非阻塞的通信中的过程特征

TTCN-3支持基于过程的阻塞的（Blocking）和非阻塞的（Non-Blocking）的通信。对于非阻塞的（Non-Blocking）通信中的特性（Signature）使用关键字**noblock**，只有**in**参数（见章节13.2），且没有返回值（见章节13.3），但可能出现例外（exceptions，见章节13.4）。没有**noblock**关键字的特性默认为阻塞的基于过程的通信。

例：

```
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

过程特征定义中可以带有参数。在一个**signature**定义中，参数列表可以包括参数标识符、参数类型和它们的方向，即输入（**in**）、输出（**out**）或输入输出（**inout**）。方向输入/出（**inout**）和输出（**out**）表示这些参数用于远程过程（Remote Procedure）重新获得信息。值得注意的是，参数的方向指的是从被调用方（Called Party）来看的，而不是调用方（Calling Party）。

例：

```
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。这个过程有三个参数：Par1是整型输入参数，
// Par2是一个浮点型输出参数，Par3是一个整型的输入/出参数。
```

13.3 远程过程的返回值

一个远程过程可以在其终止后返回一个值返回值的类型要通过相应的**signature**定义中的**return**子句来指明。

例：

```
signature MyRemoteProcFour (in integer Par1) return integer;
// MyRemoteProcFour将用于阻塞的基于过程的通信，它有一个整型输入参数Par1，在过程终止后返回一个整型值。
```

13.4 例外描述

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例外（Exceptions）。因此，可以使用模板机制和匹配机制来描述和检查远程过程的返回值。

注意：由于从被测系统SUT产生或发送到被测系统SUT的例外向相应的TTCN-3类型或SUT表示方法的转换属于工具和系统特性（system specific）类，因此它超出了本文的讨论范围。

例外是在**signature**定义中以例外列表的形式被定义的。这个列表定义了与可能出现的例外相关的所有可能的各种类型（在例外的涵义中，例外通常只是通过它们自身类型的特定值来区分是否是例外）。

例：

```
signature MyRemoteProcFive (inout float Par1) return integer
        exception (ExceptionType1, ExceptionType2);
```

```
// MyRemoteProcFive将用于阻塞的基于过程的通信。它可以通过输入输出参数Par1返回一个浮点值和一个整型值，
// 或是产生ExceptionType1类型或ExceptionType2类型的例外。

signature MyRemoteProcSix (in integer Par1) noblock
        exception (integer, float);
// MyRemoteProcSix将用于非阻塞的基于过程的通信。在过程非成功终止的情况下，MyRemoteProcSix可能
// 会产生整型或浮点型例外。
```

14 模板声明

14.0 概要

模板（Templates）用于传送一个特定值的集合或是测试接收的值的集合是否与模板说明匹配。

模板具有下列特性：

a) 模板提供了一种组织和重复使用测试数据的方法，其中包括继承的简单形式；

b) 模板能够被参数化；

c) 模板允许匹配机制；

d) 模板既能够用于基于消息的通信，也能用于基于过程的通信。

可以在一个模板的值集合中说明范围（ranges）和匹配属性，然后在基于消息的通信或基于过程的通信中中使用。模板可以用来说明任意TTCN-3类型或是过程特征。基于类型的模板（Type-based Templates）用于基于消息的通信，而过程模板（Signature Templates）用于基于过程的通信。

一个模板声明必须详细说明一个基本值的集合，或是一个与在相应的类型或过程特征（signature）中定义的每一个字段（field）相匹配的符号的集合，即必须完整地描述它。一个模板修改声明（Modified template declaration，见章节14.6）仅说明原模板中要改变的字段即可，也就是说，它是一个对模板部分说明。在用于消息的模板说明中不能使用“未被使用符号”（“-”），但“未被使用符号”（“-”）可以在过程特征模板的不相关参数和所有模板修改声明中用于指明特定字段或元素保持不变。

14.1 消息模板的声明

14.1.0 概要

可以用模板来说明带有实际值的消息实例，一个模板可以被认做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中定义的除特定配置和默认类型（**port, component, address**和**default**类型）外的任意TTCN-3类型说明模板。

例：

```
// 用于接收操作时，该模板将匹配任意整型值
template integer Mytemplate := ?;
// 该模板仅匹配整型值1、2、3
template integer Mytemplate := (1, 2, 3);
```

14.1.1 用于发送消息的模板

在一个发送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。用于发送消息的模板在执行发送操作（**send**）时会被完全定义，也就是说所有字段都会解析到实际值，而且在模板的字段中不会直接或间接地使用匹配机制。

注意：对于用于发送消息的模板来讲，省略一个可选择字段会被看做是一个值符号（value notation），而不是一个匹配机制。

例：

```
// 给定的消息的定义
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

用于接收操作（**receive**）的模板定义了一个与接收消息相匹配的数据模板，用于接收的模板可以使用匹配机制（定义在附录B中），且接收值与该模板不绑定。

例：

```
// 给出的消息的定义
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
}

// 一个相应的接受操作可以是
MyPCO.receive(MyTemplate);
```

14.2 过程特征模板的声明

14.2.0 概要

可以使用模板说明带有实际值的过程参数列表的实例，可以通过引用相关的过程特征定义来为任意过程定义模板。

例：

```
// 一个远程过程的过程特征定义
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 用于过程调用的模板

用于**call**或**reply**操作的模板为所有的输入（**in**）参数和输入/出（**inout**）参数定义了一个完整的字段值集合。在**call**操作中，模板中的所有的**in**和**inout**参数将解析到实际的值，在这些字段中不会直接或间接地使用匹配机制。任意**out**参数的模板说明都被简单地忽略掉了，因此允许为这些字段说明匹配机制或省略匹配机制（见附录B）。

例：

```
// 在14.2介绍中给定的例子

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

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

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

// 模板从不返回值，对于Par2和Par3的情况，必须使用在调用语句的末尾处的赋值子句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**操作的模板定义了一个与接收参数字段相匹配的数据模板。附录B中定义的匹配机制可能被用于任意被**getcall**操作使用的模板，但是不会发生接收值与模板的绑定。在匹配过程中，任意**out**参数都将被忽略。

例：

```
//在14.2介绍中给定的例子

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

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

// 有效的getcall，Par1 == 1且Par3为任意整型值时都将匹配
MyPCO.getcall(RemoteProc:Template3);
```

14.3 模板匹配机制

一般来讲，匹配机制用来替换单个模板字段的值或是整个模板的内容，其中一些匹配机制可以联合使用。

匹配机制和通配符也可以嵌入在接收事件（如**receive**, **getcall**, **getreply**和**catch**操作）语句行中使用，它们可以出现在明确的模板字段值中。

例1：

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

当模板字段值明确地标识了类型时，类型标识符可以省略。

例2：

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

注意：下列类型可以被省略：integer, float, boolean, objid, bitstring, hexstring, octetstring。

然而，嵌入式模板（in-line template）的类型应该在接收该模板所通过的端口的列表中。在列出的类型和模板字段值出现分歧的情况下（例如由子类型造成的），类型名应该包含在接收语句中。

匹配机制分为四类：

a) 特定的字段值：

- 求特定值的表达式；
- **omit**：字段值将被省略；

b) 可以用来代替字段值的特殊符号：

- **(…)**：值列表；
- **complement (…)**：值列表的补集（Complement）；
-
- **?**：表任意值的通配符；
- *****：表任意值或空的通配符（例如一个省略值）；
- **(**下限 **to** 上限**)**：上下限范围内的整型值域，包括上下限；

c) 可以被用做内部值（Inside Value）的特殊符号：

- **?**：表串、数组、**record of**或**set of**中任意单个元素的通配符；
- *****：表串、数组、**record of**或**set of**中任意数目的连续元素或根本为空（例如一个省略的元素）

d) 描述字段值属性的特殊符号：

- **length**：用于串类型的串长度限制以及**record of**, **set of**和数组的元素数目的限制；
- **ifpresent**：用于匹配没有省略的可选择字段值。

支持的匹配机制、相关的符号（如果有）和应用范围见表六。表中左起第一列列出了与适用于使用这些匹配机制的类型相等价的所有ITU-T Recommendation X.680 series [7]、[8]、[9]和[10]中定义的TTCN-3和ASN.1的类型，每个匹配机制的完整描述见附录B。

表六: TTCN-3 匹配机制

Used with values of	值		代替值							内部值		属性	
	特定值	省略值	补集	值列表	任意值 (?)	任意值或空 (*)	值域	超集	子集	任意元素 (?)	任意元素或空 (*)	长度限制	IfPresent
boolean	是	是	是	是	是	是							是
Integer	是	是	是	是	是	是	是						是
Char	是	是	是	是	是	是	是						是
universal char	是	是	是	是	是	是	是						是
Float	是	是	是	是	是	是	是						是
bitstring	是	是	是	是	是	是				是	是	是	是
octetstring	是	是	是	是	是	是				是	是	是	是
hexstring	是	是	是	是	是	是				是	是	是	是
character strings	是	是	是	是	是	是	是			是	是	是	是
Record	是	是	是	是	是	是							是
Record of	是	是	是	是	是	是				是	是	是	是
Array	是	是	是	是	是	是				是	是	是	是
Set	是	是	是	是	是	是							是
set of	是	是	是	是	是	是		是	是	是	是	是	是
enumerated	是	是	是	是	是	是							是
Union	是	是	是	是	是	是							是

14.4 模板参数化

14.4.0 概要

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字段值和模板、函数和特定的匹配机制。形参和实参列表规则遵循章节5.2中的定义。

例：

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

// 可以以如下方式被使用
pco1.send(MyTemplate(123));
```

14.4.1 带有匹配属性的参数化

为了使模板或匹配符号可以作为参数传送，需要在类型字段前加外部关键字**template**。这样做使得一个参数类型为模板类型，并有效地扩展相关类型所容许的参数，使其包括正常值集和适当的匹配属性集合（见附录B）。模板参数字段不能通过传参来调用。

例：

```
// 模板
template MyMessageType MyTemplate (template integer MyFormalParam):=
```

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

// 可以以如下方式使用
pco1.receive(MyTemplate(?));
// 或
pco1.receive(MyTemplate(omit));
```

14.5 作为参数传递模板

只有函数（**function**）、测试例（**testcase**）、可选步（**altstep**）和模板（**template**）定义中可以带有作为形参的模板。

例：

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

14.6 修改模板

14.6.0 概要

通常，一个模板描述了一个基本的或默认的值的集合，或定义在适当的定义中用于每个匹配符的值集。在仅有少量变化而需要去说明一个新的模板的情况下，可以说明一个修改模板（Modified Template）。一个修改模板说明了对原始模板的特定字段的直接或间接的更改。

用关键字**modifies**来表示修改模板所源于的父模板，这个父模板可以是一个原始模板，也可以是一个修改模板。

更改以链接的方式发生，最终要追溯到原始模板。如果一个模板字段和该字段相应的值或匹配符号在修改模板中没有被说明，那么将使用父模板中该字段值或匹配符号。当被修改的字段嵌套在一个结构类型的模板字段中时，除了这个被明确说明的字段，父模板中相应的结构类型字段中的其它字段都保持不变。

无论是直接还是间接方式，修改模板都不能自己调用自己，也就是说不允许递归调用。

例：

```
// 给定
template MyRecordType MyTemplate1 :=
{
    field1 := 123,
    field2 := "A string",
    field3 := true
}
// 再定义
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
    field1 := omit,                 // field1是可选项，但在MyTemplate1中则要
    field2 := "A modified string"
                                    // field3没有变化
}
// 也可以写为
template MyRecordType MyTemplate2 :=
{
    field1 := omit,
    field2 := "A modified string",
    field3 := true
}
```

14.6.1 修改模板的参数化

如果一个基础模板有一个形参列表，则下列规则将应用于所有该基础模板所派生出的修改模板（可经过一个或多个修改步骤产生）：

a) 派生模板（Derived Template）不能省略从基础模板到实际的修改模板的任何一个修改步骤中定义的参数；

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；

d) 在修改模板中，包含有参数化模板的基础模板的字段不能被修改或是明确地被表示为省略字段。

例：

```
// 给定
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 :=
{   // 在Template1中，field1是参数化模板，因而在Template2中，field1仍是
    field2 := "A modified string",
}
```

14.6.2 嵌入式修改模板

与明确地命名修改模板一样，在TTCN-3中还允许定义嵌入式修改模板。

例：

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

// 可以定义Setup的一个嵌入式修改模板
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嵌入式派生模板来改变模板字段。在该模板随后的相关通信事件中，这些改变对模板字段值将不会造成影响。

在通信事件中，不使用带点的表示符号*MyTemplateId.FieldId*设置或重新得到模板的值，而使用“->”符号（见章节23）。

14.8 匹配操作

匹配（**Match**）操作允许比较模板值与变量值，该操作返回一个布尔值。如果模板值与变量值类型不兼容（见章节6.7），则该操作返回假（false）。如果类型值项匹配，则该操作的返回值指出变量值与给定模板值是否一致。

例：

```
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（false）
     :
}
```

14.9 操作的值

valueof操作允许把模板中指定的值赋值给变量的字段，但该变量和模板字段的类型应该是兼容的（见章节6.7），模板的每个字段可以解析为一个单个的值。

例：

```
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 概要

TTCN-3支持许多可用于TTCN-3表达式的术语中的预定义操作符，它们分为以下七类：

a) 算术运算符（arithmetic operators）；

b) 串运算符（string operators）；

c) 关系运算符（relational operators）；

d) 逻辑运算符（logical operators）；

e) 位运算符（bitwise operators）；

f) 移位运算符（shift operators）；

g) 循环移位符（rotate operators）。

.

表 7: TTCN-3 运算符列表

类别（Category）	运算符（Operator）	符号或关键字（Symbol or Keyword）
算术运算符（Arithmetic operators）	Addition	**+**
	subtraction	**-**
	multiplication	*****
	division	**/**
	modulo	**mod**
	remainder	**rem**
串运算符（String operators）	concatenation	**&**
关系运算符（Relational operators）	equal	**==**
	less than	**<**
	greater than	**>**
	not equal	**!=**
	greater than or equal	**>=**
	less than or equal	**<=**
逻辑运算符（Logical operators）	logical not	**not**
	logical and	**and**
	logical or	**or**
	logical xor	**xor**
位运算符（Bitwise operators）	bitwise not	**not4b**
	bitwise and	**and4b**
	bitwise or	**or4b**
	bitwise xor	**xor4b**
移位运算符（Shift operators）	shift left	**<<**
	shift right	**>>**
循环移位运算符（Rotate operators）	rotate left	**<@**
	rotate right	**@>**

表8表示了这些运算符的优先级别，表中每一行中所列的运算符都具有相同的优先级别。如果一个表达式中出现多个具有相同优先级的运算符，则按从左至右的顺序评定优先级别。括号可以用于把操作数集合在一起，在这种情况下，括号具有最高优先级。

表 8：运算符优先级

优先级	操作符类型	操作符
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	**==, !=**
	二元	**not**
	一元	**and**
	二元	**xor**
	二元	**or**
最低	二元	

15.1 算术运算符

算术运算符指加（addition）、减（subtraction）、乘（multiplication）、除（division）、取模（modulo）和取余（remainder）操作。除了取模操作（**mod**），这些操作符的操作数应该为整型（**integer**，包括整型的派生类型）或浮点型（**float**，包括浮点型的派生类型）；取模操作只能使用整型（**integer**，包括整型的派生类型）。

对整型操作数进行算术运算，结果仍为整型，对浮点型操作数进行算术运算，结果为浮点型。

在加或减法操作用做为一元操作符时，操作上述操作数的规则也同样使用。对正数进行减法结果可以是负数，反之对负数进行减法结果可以是正数。

执行除法（/）操作的两个结果：

a) 被除数和除数均为整数时，结果亦为整数（也就是说分数部分被丢弃）；

b) 被除数和除数均为浮点数时，结果为浮点数（也就是说不会丢弃分数部分）。

对整型操作数进行取余（**rem**）和取模（**mod**）操作，结果为整型，x **rem** y和x **mod** y都计算x被y除所得的余数。因此，y仅被定义为非零的操作数。对于正数x和y，x **rem** y和x **mod** y操作的结果一样，但是对于为负数的x和y，结果则不一样。

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|          如果 x >= 0
        = 0                  如果 x < 0 且x rem |y| = 0
        = |y| + x rem |y|        如果 x < 0且x rem |y| < 0
```

表9举例说明了**Mod**和**rem**操作的区别：

表9：Mod和rem操作的结果

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

预定义的串运算符完成可兼容串类型的连接，该操作是从左至右的简单连接，没有暗含算术加法的形式，结果类型为操作数的原始类型。

例：

```
'1111'B & '0000'B & '1111'B 结果为 '111100001111'B
```

15.3 关系运算符

预定义的关系运算符表示的关系有：等于（equality，==），小于（less than，<），大于（greater than，>），不等于（non-equality to，!=）大于等于（greater than or equal to，>=）和小于等于（less than or equal to，<=）。除了枚举类型（**enumerated**）这个例外，相等和不相等的操作数可以是任意的但兼容的类型。而其他关系运算操作的操作数类型只能是整型（**integer**，包括整型的派生类型）、浮点型（**float**，包括浮点型的派生类型）或一些枚举类型（**enumerated**）的实例。该运算的结果为布尔类型。

如果两个字符串（**charstring**）或通用字符串（**universal charstring**）的长度以及所有对应位置的字符都相等，则这两个串相等。**bitstring**、**hexstring**或**octetstring**类型的值使用相同的相等规

则，但有例外——所有位置上应该相等的部分是相应的是比特串（bits）、十六进制串（hexadecimal digits）或十六进制位组串（pairs of hexadecimal digits）。

两个记录类型（**record**）、集合类型（**set**）、**record of**类型或**set of**类型值相等，当且仅当他们的有效值结构兼容（见章节6.7）且所有相应域的值相等。**Record**类型值可以与**record of**类型值比较，而**set**值可以与**set of**值比较，在这些情况下的比较规则与比较两个**record**类型值或**set**类型值的规则一样。

注意： “所有字段”意味着在一个**record**类型没有表示实际值的可选字段将被认为是未定义值。这个字段在和另一个**record**类型比较时，仅和遗漏的可选字段等价（同样被认为是一个未定义字段值）；当它与**record of**类型的一个值比较时，仅与一个未定义值的元素相等。这个规则同样适用于两个**set**类型值或一个**set**类型值和一个**set of**类型值做比较时。

两个**union**类型值相等，当且仅当在这两个**union**类型的值中，被选中的字段的类型兼容且它们的实际值相等。

例：

```
// 给定
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,
            };

// 以及
const   SetA    conSetA1    :=  { a1 := 0, a3 := 2 };
            // 注意字段值的顺序无关紧要
const   SetB    conSetB1    :=  { b1 := 0, b3 := 2 };
const   SetB    conSetB2    :=  { b2 := 0, b3 := 2 };
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 };

// 那么
conSetA1 == conSetB1;
    // 返回true
conSetA1 == conSetB2;
    // 返回false，因为a1和a2的值与它们在conSetB2中的对应成分的值不相等（对应元素没有被省略）
conSetA1 == conSetC1;
```

```
    // 返回false，因为SetA和SetC中的有效的值结构是不兼容的
conSetA1 == conSetOf1;
    // 返回true
conSetA1 == conSetOf2;
    // 返回false，作为与a2对应成分的值是2，但是与a3对应的成分却没有定义
conSetC1 == conSetOf2;
    // 返回true
conUniD1 == conUniE1;
    // 返回true
conUniD1 == conUniE2;
    // 返回false，作为被选中字段的e2与UniD1中的d1字段并不是对应字段
conUniD1 == conUniF1;
    // 返回false, UniD1和UniF的有效的值结构是不兼容的
```

15.4 逻辑运算符

预定义的布尔操作符执行否定（negation）、逻辑与（logical **and**）、逻辑或（logical **or**）和逻辑异或（logical **xor**）操作，他们的操作数应该是布尔类型（**boolean**），逻辑运算的结果也是布尔类型。

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

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

如果逻辑或（**or**）运算的两个操作数中，至少有一个操作数的值为**true**，则返回**true**；否则返回**false**。

如果逻辑异或（**xor**）运算的两个操作数中有且只有一个操作数的值为**true**，则返回**true**；如果其两个操作数的值同为**false**或同为**true**，则返回**false**。

15.5 位运算符

预定义的位运算符执行按位取反（bitwise **not**）、按位与（bitwise **and**）、按位或（bitwise **or**）或按位异或（bitwise **xor**）操作，这些运算符分别为**not4b**、**and4b**、**or4b**和**xor4b**。

注意：读作“按位取反”、“按位与”等。

位运算符的操作数的类型应该是**bitstring**、**hexstring**、**octetstring**。**and4b**、**or4b**和**xor4b**的两个操作数的类型应该是可兼容的，位运算的结果类型应该是该运算操作数的原始类型。

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

```
not4b  '1'B  得到  '0'B
not4b  '0'B  得到  '1'B
```

例1：

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

位运算符**and4b**接受两个等长的操作数，对于两个操作数中相应位的值均为1时，结果中相应位的值亦为1，否则为0，即：

```
'1'B and4b '1'B 得到  '1'B
'1'B and4b '0'B 得到  '0'B
'0'B and4b '1'B 得到  '0'B
'0'B and4b '0'B 得到  '0'B
```

例2：

```
'1001'B and4b '0101'B 得到 '0001'B
```

```
'B'H and4b '5'H       得到  '1'H
'FB'O and4b '15'O     得到  '11'O
```

位运算符**or4b**接受两个等长的操作数，对于两个操作数中相应位的值均为0时，结果中相应位的值亦为0，否则为1，即：

```
'1'B or4b '1'B 得到  '1'B
'1'B or4b '0'B 得到  '1'B
'0'B or4b '1'B 得到  '1'B
'0'B or4b '0'B 得到  '0'B
```

例3：

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

位运算符**xor4b**接受两个等长的操作数，对于两个操作数中相应位的值均为0或均为1时，结果中相应位的值亦为0，否则为1，即：

```
'1'B xor4b '1'B 得到  '0'B
'0'B xor4b '0'B 得到  '0'B
'0'B xor4b '1'B 得到  '1'B
'1'B xor4b '0'B 得到  '1'B
```

例4：

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

15.6 移位运算符

预定义的移位运算符执行左移（shift left，**<<**）和右移（shift right，**>>**）操作。此类运算符左侧的操作数类型应该是**bitstring**、**hexstring**或**octetstring**，右侧的操作数类型应该是整型**integer**，而运算结果的类型则应该与其左侧操作数的类型相同。

移位运算根据其左侧的操作数类型的不同而不同，如果其左侧操作数的类型是

a) **bitstring**，那么应用的移位单位是1bit；

b) **hexstring**，那么应用的移位单位是1十六进制位；

c) **octetstring**，那么应用的移位单位是1个八位组bit。

左移（**<<**）操作符接受两个操作数，它将左侧操作数左移右侧的操作数所描述的位数，移出的过多的位（bits、hexadecimal digits或octets）被丢弃。左侧操作数的每个移动单位在向左移的过程中，低位（右边的位）补0（根据左侧操作数的类型决定补'0'B、'0'H或是'00'O）。

注意：如果对左侧的操作数执行左移操作时，发生了一个与系统相关的溢出，则应该指定一个错误判定。

例1：

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

右移（**>>**）操作符接受两个操作数，它将左侧操作数右移右侧的操作数所描述的位数，移出的过多的位（bits、hexadecimal digits或octets）被丢弃。左侧操作数的每个移动单位在向右移的过程中，高位（左边的位）补0（根据左侧操作数的类型决定补'0'B、'0'H或是'00'O）。

例2：

```
'111001'B  >> 2          得到  '001110'B
'12345'H  >> 2           得到  '00123'H
'1122334455'O  >> (1+1)  得到  '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（rotate left，**<@**）和循环右移（rotate right，**@>**）操作。此类运算符左侧的操作数类型应该是**bitstring**、**hexstring**、**octetstring**、**charstring**或 **universal charstring**，右侧的操作数类型应该是整型**integer**，而运算结果的类型则应该与其左侧操作数的类型相同。

循环移位运算根据其左侧的操作数类型的不同而不同，如果其左侧操作数的类型是

a) **bitstring**，那么应用的循环移位单位是1bit；

b) **hexstring**，那么应用的循环移位单位是1十六进制位；

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

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

循环左移（**<@**）操作符接受两个操作数，它将左侧操作数循环左移右侧的操作数所描述的位数，移出的过多的位（bits、hexadecimal digits、octets或characters）被从左侧操作数的右边重新插入到该操作数中去。

例1：

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

循环右移（**@>**）操作符接受两个操作数，它将左侧操作数循环右移右侧的操作数所描述的位数，移出的过多的位（bits、hexadecimal digits、octets或characters）被从左侧操作数的左边重新插入到该操作数中去。

例2：

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

16 函数和可选步

在TTCN-3中，函数（Functions）和可选步（Altsteps）用于表示和构造测试行为、定义一个模块中的默认行为和组织计算，下面的章节对此做了详细的描述。

16.1 函数

16.1.0 概要

在TTCN-3的一个模块里用函数来表达测试行为，组织测试执行或是计算，如计算一个单个值，对一个变量集合进行初始化，或是检查条件。函数可以返回一个值，用后面跟了 类型标识符的关键字**return**来表示函数要返回一个值。当关键字**return**用在函数体中且返回类型定义在函数头中时，关键字**return**后要接一个值来表示该返回值，即其后接一个常数、变量引用或是一个表示返回值的表达式。返回值的类型和返回类型应该是兼容的。函数体中的返回语句终止函数执行，并在函数调用处返回返回值。

例1：

```
// 不带参数的函数MyFunction的定义
```

```
function MyFunction() return integer
{

    return 7;   // 函数终止时返回整型值7
}
```

函数可以被定义在一个模块中，或被声明为在外部被定义的（**external**）。对于一个外部函数，在TTCN-3的模块中仅仅必须提供函数的接口。外部函数的实现在本文讨论范围之外。外部函数不允许包含端口操作。

```
external function MyFunction4() return integer; // 返回一个整型值的不带参数的外部函数

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

注意1：TTCN-3的函数代替TTCN-2中的测试步和测试套程序的定义，而外部函数代替TTCN-2中的测试套操作。非形式化的函数（Informal functions）可以被声明为带有说明性注释的外部函数，或通过使用带有注释的空的形式化的函数（Formal functions）声明为外部函数。

在一个模块中，可以使用程序语句和第18章描述的操作定义函数的行为。如果一个函数使用了一个成分类型定义中声明的变量、常量、定时器或端口，应该在该函数头中使用关键字**runs on**来引用这个成分类型定义。这个规则的一个例外是该成分的作用范围内的信息是否被作为参数传入。

例2：

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

不带有**runs on**子句的函数，从不会调用一个函数或可选步，或是激活一个作为默认的带有局部**runs on**子句的可选步。

被测试成分操作**start**启动的函数总是有一个**runs on**子句（见章节22.5），并且被认为是在该成分中被调用时启动，也就是说它不具有局部意义的。不过，在不带有**runs on**子句的函数中调用这个测试成分操作**start**是可以的。

注意2：关于**runs on**子句的这个限制仅与函数和可选步有关，而与测试例无关。

在TTCN-3模块控制部分中使用的函数不会有**runs on**子句，不过，允许它们执行测试例。

16.1.1 函数的参数化

函数可以被参数化，形参列表的规则遵从章节5.2中的定义。（Functions may be parameterized. The rules for formal parameter lists shall be followed as defined in clause 5.2.）

例：

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

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函数。不返回值的函数应该直接调用，有返回值的函数既可直接调用，也可以在表达式中调用。用于实参列表的规则遵从章节5.2中的定义。

例：

```
MyVar := MyFunction4(); // MyFunction4返回的值被赋给MyVar，返回值的类型MyVar的类型必须相同

MyFunction2(MyVar2);    // MyFunction2不返回值，带有实参MyVar2被调用，可以通过传参来传入MyVar2
```

```
MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3);  // 在表达式中使用的函数
```

对于使用测试成分操作**start**来绑定到测试成分的函数，使用在章节22.5中描述的那些特定的限制条件。

16.1.3 预定义的函数

TTCN-3包含了许多使用前不需声明的预定义（内置，built-in）函数。

表10： TTCN-3 预定义函数列表

类别	函数	关键字
转换函数（Conversion functions ）	转换**integer**值为**char**值	**int2char**
	转换**integer**值为**universal char**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**char**值为**integer**值	**char2int**
	转换**universal char**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**charstring**值为**integer**值	**str2int**
	转换**charstring**值为**octetstring**值	**str2oct**
长度/大小函数（Length/size functions）	返回任意串类型的长度	**lengthof**
	返回一个**record, record of, template**, **set**, **set of** or **array**类型的元素个数	**sizeof**
出现/选择函数（Presence/choice functions）	确定一个**record, record of, template, set** or **set of**类型的可选字段是否出现（present）	**ispresent**
	确定在一个**union**类型中做了哪些选择	**ischosen**
串函数（String functions）	返回输入串中与指定的模式描述匹配的部分	**regexp**
	返回输入串的指定部分	**substr**
其他函数	产生一个随机浮点数	**rnd**

调用一个预定义函数时：

1) 实参的数目应该和形参的数目相同；且

2) 每个实参应该确定其相应的形参类型元素的值；且

3) 所有出现在实参列表中的变量都应该被绑定。

附件C给出了预定义函数的完整描述。

16.2 可选步

16.2.0 概要

TTCN-3使用可选步（Altsteps）来描述默认行为，或构造一个**alt**语句的选择对象（Alternatives）。可选步是与函数相似的范围单位。可选步主体定义了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*（*top alternatives*）构成了可选步的主体。顶层选择对象使用与**alt**语句的选择对象的语法规则相同的语法规则。

可以使用程序语句和第18章汇总的操作来定义可选步的行为。如果一个可选步包含端口操作或使用成分变量、常数或定时器，则应该在该可选步头部使用关键字**runs on**来引用相关的成分类型。这个规则的一个例外是该可选步中使用的所有端口、变量、常量和定时器是否作为参数传入的情况。

例：

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

// 使用PCO1、PCO2、MyIntVar和MyTimer of MyComponentType的可选步的定义

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

可选步可以调用函数和可选步，或作为默认来激活可选步。一个不带有**runs on**子句的可选步从不会调用带有**runs on**子句的函数、可选步或作为默认来激活可选步。

16.2.1 可选步的参数化

可选步可以被参数化。被作为默认来激活的一个可选步应该仅带有值参数，也就是说输入（**in**）参数。在**alt**语句中仅作为一个选择对象或在TTCN-3行为描述中仅作为独立（stand-alone）语句被调用的可选步可以有输入（**in**）、输出（**out**）和输入/出（**in/out**）参数。用于形参列表的规则遵从章节5.2中的定义。

16.2.2 可选步中的局部定义

16.2.2.0 概要

可选步可以定义常量、变量和定时器的局部定义。应该在选择对象前定义该局部定义。

例：

```
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 可选步中局部定义初始化的限制

通过调用值返回函数来进行局部定义的初始化可能会有副作用。为避免成分实际的快照（Snapshot）和其状态之间不一致的副作用，在一个局部定义的初始化过程中，不应调用下列操作：

- **done**操作
- 所有的端口操作，也就是start (port)、stop (port)、clear、send、receive、trigger、call、getcall、reply、getreply、raise、catch、check。

注意1： **done**、**start** (port) 、**stop** (port) 、**clear**、**receive**、**trigger**、**getcall**、**getreply**、**catch**和**check**操作的执行可能导致与实际快照的冲突。它们的执行可能会从端口队列中删除信息，限制对端口队列的访问和/或在实际快照确定值的过程中导致新的快照。

注意2： 应该避免为了可读性目的使用**send**、**call**、**reply**和**raise**操作，也就是说所有通信都应该明确，而不是作为通信中的副作用。

- 定时器操作**start** (timer)、**timeout**和**stop** (timer)。

注意3： 允许使用读定时器（**readtimer**）和定时器运行（**running**）操作。

注意4： 这些用于可选步中局部定义初始化的限制，与避免**alt**语句中或可选步中用来选择和取消选择的布尔表达式的副作用的限制相同。

16.2.3 可选步的调用

可选步的调用总是与**alt**语句相关。该调用可以通过默认机制隐式地完成（见章节21），或是通过在**alt**语句中的一个直接调用显式地完成（见章节20.1.6）。可选步的调用不导致新的快照，通过使用调用该可选步的**alt**语句的实际快照，来完对成可选步的顶层选择对象求值。

注意： 如果在一个选定的顶层选择对象中，指定并进入了一个新**alt**语句，当然会采纳可选步中的一个新快照。

对于借助于默认机制的可选步的一个隐式调用，在到达调用位置之前，必须通过**activate**语句将该可选步激活为一个默认。

例1：

```
 :
var default MyDefVarTwo := activate(MySecondAltStep()); // 一个可选步激活为默认
 :
```

在**alt**语句中的可选步的显式调用看起来象作为选择对象的函数调用。

例2：

```
 :
alt {
    [] PCO3.receive {
         …
        }
    [] AnotherAltStep();        //作为一个alt语句的选择对象显式调用可选步
    [] MyTimer.timeout {}
}
```

在**alt**语句中显式调用可选步时，下一个要检查的选择对象是该可选步中的第一个选择对象。与带有例外（进入该可选步时没有照新快照）的**alt**语句的选择对象（见章节20.1）一样的方式来检查和执行该可选步选择对象。可选步的非成功终止（即所有该可选步的顶层选择对象都被检查了，但没有发现匹配分支）引起对下一个选择对象求值，或调用默认机制（如果显式调用是该可选步的最后的选择对象）。一个成功终止会引起测试成分的终止，即可选步以一个**stop**语句结束；或引起该**alt**语句的一个新快照和重新求值，即带有可选步以**repeat**（见章节20.2）语句结束，或是该**alt**语句的立即继续，即该测试步的被选定的顶层的选择对象不以显式的**repeat**语句结束。

在TTCN-3行为描述中，可选步可以作为独立语句被调用。在这种情况下，对可选步的调用可以解释为用于仅带有一个描述显式调用该可选步的选择对象的**alt**语句的简写（shorthand）。

例3：

```
// 语句
AnotherAltStep(); // 假设AnotherAltStep是一个正确定义的altstep

//是下面的alt语句一个简写：

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

16.3 用于不同成分类型的函数和可选步

如果根据章节6.7.3，类型“A”和“B”兼容的话，可以在成分类型“A”的一个实例上启动在自己的**runs on**子句中引用了成分类型“B”的函数或可选步。

17 测试例

17.0 概要

测试例（Test cases）是函数的一个特殊种类。在一个模块控制部分，使用**execute**语句来启动测试例（见章节27.1），一个测试例的执行结果总是一个**verdicttype**类型值。每个测试例应该包含一个且仅一个MTC，并在该测试例定义的头部引用MTC类型。 定义在测试例主体中的行为是MTC的行为。

调用一个测试例时创建MTC，并实例化MTC端口和测试系统接口，测试例定义中指定的行为在MTC上开始。所有这些行为将会被隐式地执行，也就是说，不带有明确的**create**和**start**操作。

在测试例头部，为允许这些隐式操作提供信息有两部分内容：

a) 接口部分（必须的）：用关键字**runs on**表示，关键字**runs on**为MTC引用必要的成分类型，并使相关联的端口名在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，关键字**system**引用为测试系统接口定义所需端口的成分类型。在测试执行期间，仅当MTC被实例化了，测试系统部分才应该被省略。在这种情况下，MTC隐式地定义了测试系统接口端口。

例：

```
testcase MyTestCaseOne()
runs on MyMtcType1          // 定义MTC类型
system MyTestSystemType     // 使TSI的端口名对于MTC可见
{
    :   // 当测试例被调用时，这里定义的行为在MTC上执行
}

// 或者，一个仅MTC被实例化的测试例
```

```
testcase MyTestCaseTwo() runs on MyMtcType2
{
    :   //当测试例被调用时，这里定义的行为在MTC上执行
}
```

17.1 测试例的参数化

测试例可以被参数化，且应遵循章节5.2中定义的形参列表规则。

18 程序语句和操作的纵览

测试例、函数、可选步和TTCN-3模块的控制部分的基本程序元素是基本的程序语句（如表达式、赋值、循环构造（Loop Constructs）等）、行为语句（如顺序行为、选择对象行为、交叉、默认等）和操作（如**send**、**receive**、**create**等）。

语句既可以是单一语句（不包括其他程序语句），也可以是复合语句（可以包含其他语句或语句和声明块）。

语句将按照它们出现的先后顺序执行，也就是说象图8所示顺序执行。

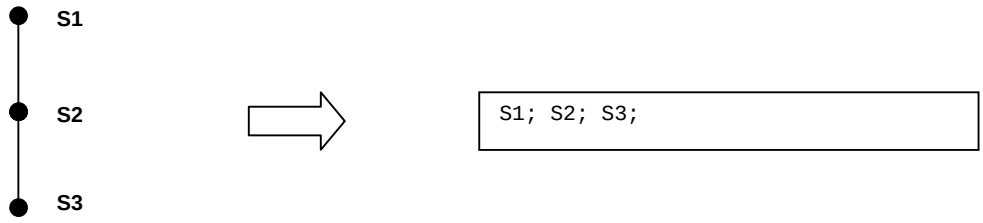


图 8：顺序执行图示

在语句序列中，用分隔符“;”来分隔各个单独的语句。

例：

```
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，语句中的一个分支意味着采取空动作，即什么都不做。

表 11：TTCN-3 语 句 和 操 作 纵 览

语句（Statement）	相关的关键字或符号（Associated keyword or symbol）	能用在模块控制中（Can be used in module control）	能用在函数、测试例和可选步中（Can be used in functions, test cases and altsteps）
基本程序语句（Basic program statements）			
表达式（Expressions）	**(…)**	是	是
赋值（Assignments）	**:=**	是	是
日志（Logging）	**log**	是	是
标记和跳转（Label and Goto）	**label / goto**	是	是
条件（If-else）	**if (…) {…} else {…}**	是	是
For循环（For loop）	**for (…) {…}**	是	是
While循环（While loop）	**while (…) {…}**	是	是
Do while循环（Do while loop）	**do {…} while (…)**	是	是
停止执行（Stop execution）	**stop**	是	是
行为程序语句（Behavioural program statements）			
选择对象行为（Alternative behaviour）	**alt {…}**	是 （见注意1）	是
重新确定选择对象行为（Re-evaluation of alternative behaviour）	**repeat**	是 （见注意1）	是
交叉行为（Interleaved behaviour）	**interleave {…}**	是 （见注意1）	是
返回控制（Returning control）	**return**		是
用于默认处理的语句（Statements for default hanling）			
激活一个默认（Activate a default）	**activate**	是 （见注意1）	是
停用一个默认（Deactivate a default）	**deactivate**	是 （见注意1）	是
配置操作（Configuration operations）			
创建并行测试成分（Create parallel test component）	**create**		是
连接两个成分	**connect**		是
断开两个成分	**disconnect**		是
映射端口到测试接口	**map**		是
从测试系统接口取消端口的映射	**unmap**		是
获得MTC地址	**mtc**		是
获得测试系统接口地址	**system**		是
获得自身地址	**self**		是
开始测试成分的执行	**start**		是
停止测试成分的执行Stop execution of test component	**stop**		是
检查一个PTC的终止	**running**		是
等待一个PTC的终止	**done**		是
通信操作（Communication operations）			
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
提出例外（对一个已被接受的调用）Raise exception (to an accepted call)	**raise**		是
接受消息	**receive**		是
触发消息（Trigger on message）	**trigger**		是
从远程实体接受过程调用	**getcall**		是
处理来自以前调用的响应	**getreply**		是
抓住例外（从被调用实体）Catch exception (from called entity)	**catch**		是
检查（当前）消息/接收到的调用 Check (current) message/call received	**check**		是
清除端口（Clear port）	**clear**		是
Clear and give access to port	**start**		是
Stop access (receiving & sending) at port	**stop**		是
定时器操作（Timer operations）			
启动定时器	**start**	是	是

停止定时器	**stop**	是	是
读取经过的时间Read elapsed time	**read**	是	是
检查定时器是否运行Check if timer running	**running**	是	是
超时事件Timeout event	**timeout**	是	是
判定操作（Verdict operations）			
设置本地判定Set local verdict	**setverdict**		是
获得本地判定Get local verdict	**getverdict**		是
外部事件（External actions）			
外部模拟一个（SUT）活动Stimulate an (SUT) action externally	**action**	是	是
测试例执行（Execution of test cases）			
执行测试例	**execute**	是	是 （见注意2）
注意1：仅能用于控制定时器操作。 注意2：仅能用于模块控制中的函数和可选步。			

19 基本的程序语句

19.0 概要

基本的程序语句是指表达式、操作、循环构造（loop constructs）等。所有基本程序语句都可以用在模块的控制部分和TTCN-3的函数、可选步和测试例中。

表 12： TTCN-3 基 本 的 程 序 语 句 纵 览

基 本 程 序 语 句	
语 句	**相关的关键自或符号**
Expressions	**(…)**
Assignments	**:=**
Logging	**log**
Label and Goto	**label / goto**
If-else	**if (…) { … } else { … }**
For loop	**for (…) { … }**
While loop	**while (…) { … }**
Do while loop	**do { … } while (…)**
Stop execution	**stop**

19.1 表达式

19.1.0 概要

TTCN-3允许表达式（Expressions）说明中使用第15章定义的操作符。表达式是由其他（简单）表达式构建的，可以只使用返回值的函数。一个表达式的结果应该是一个特定类型值，且操作符应该和操作数类型兼容。

例：

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

19.1.1 布尔表达式

一个布尔表达式应该仅包含布尔值和/或布尔操作符以及/或关系运算符，并计算一个布尔值，即**`true`**或**`false`**。

例：

```
((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。在执行赋值的过程中，“:=”的右边用于计算出左边相同类型的元素值。赋值的结果就是把一个变量和一个表达式的值绑定，这个表达式应该不包含没有绑定的变量。所有赋值的发生顺序按照它们的出现顺序进行，也就是说，按从左至右处理。

例：

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

19.3 日志语句

日志语句（**log**）提供了向有关测试控制设备或使用该语句的测试成分写字符串的方法。

例：

```
log("Line 248 in PTC_A");
// 把串“Line 248 in PTC_A”写到测试系统的一些日志设备
```

注意： 定义复杂的日志和可能依赖工具的回溯能力，这超出了本文的范围。

19.4 标签语句

标签语句（**label**）允许测试例、函数、可选步和模块控制部分中对标签的说明，可以根据附录A中定义的语法规则，象其它TTCN-3行为的程序语句一样自由地使用标签语句。标签语句可以在一个TTCN-3语句前或后使用，但是不能作为一个选择对象的第一个语句，或是一个**alt**语句、**interleave**语句或可选步中顶层的选择对象。用在关键字之后的标签在同一个测试例、函数、可选步或控制部分中的所有标签中应该是唯一的。

例：

```
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);
        goto L1;                        // 跳到标签L1
    }
}
 :
```

19.5 Goto语句

goto语句可以用在函数、测试例、可选步和TTCN-3模块的控制部分中，它执行一个跳转，跳转到一个标签（**label**）。

goto语句提供自由跳转（即在一个语句序列里向前和向后）、跳出一个单个的复合语句（如一个**while**循环）和跳过嵌套的复合语句数层的可能性。然而，**goto**语句的使用将受到下面规则的限制：

a) 不允许使用**goto**语句跳出或跳入函数、测试例、可选步和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：

```
// 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;               // … 向前跳出两层嵌套alt语句
             }
          [] PCO2.receive(MyError) {
                goto L3;               // … 跳回到当前的alt语句，
             }
          [] any port.receive {
                goto L2;               // … 向后跳出两层嵌套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，也是通常所说的条件语句，用于表示布尔表达式导致的控制流中的分支，其用法如下：

```
if (表达式1)
    语句块1
else
    语句块2
```

语句块$_x$处引用一个语句块。

例：

```
if (date == "1.1.2000") 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，但格式不会具有语法或语义的意义。

19.7 for语句

for语句定义了记数循环，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语句包含两个赋值和一个布尔表达式。第一个赋值对初始化循环下标（index，或计数器）变量，布尔表达式终止这个循环，而第二个赋值用于控制该下标变量。

例1：

```
for (j:=1; j<=10; j:= j+1) { … }
```

循环的终止条件应该由布尔表达式表示，在每次新的循环反复的开始时检查该布尔表达式。如果表达式值为**true**，则继续执行**for**循环语句后面的语句。

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，也可以在**for**语句头部中声明并初始化该下标变量。如果该下标变量是在**for**语句头部中声明并初始化的，那么它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;                          // 整型变量j的声明
for (j:=1; j<=10; j:= j+1) { … }        // 作为循环的下标变量的j的用法


for (var float i:=1.0; i<7.9; i:= i*1.35) { … } // 在循环的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的开始时检查循环条件。如果不满足循环条件了，那么退出该循环，继续执行紧跟在**while**循环后的语句。

例：

```
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时检查循环条件，这就意味着如果使用**do-while**循环，在第一次对循环条件求值之前，循环体的行为至少被执行一次。

例：

```
do { … } while (j<10);
```

19.10 停止执行语句

根据使用**stop**语句的环境不同，**stop**语句以不同的方式终止执行操作。如果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，则它终止该测试执行。如果在一个测试成分上执行的测试例、可选步或在函数中使用**stop**语句，则他终止相关的测试成分。

例：

```
module MyModule {
     :  // 模块定义
    testcase MyTestCase() runs on MyMTCType system MySystemType{
         :
        stop         // 停止一个测试成分
    }
    control {
         :  // 测试执行
        stop    // 停止测试活动
    } // 结束控制
} // 结束模块
```

注意：停止一个测试成分的**stop**语句的语义与停止成分操作**self.stop**的语义相同（见章节 22.6）。

20 行为的程序语句

20.0 概要

行为的程序语句可以用在测试例、函数和可选步中，除了：

(a) 仅用在函数中的**return**语句；和

(b) 也可用在模块控制中的**alt**语句、**interleave**语句和**repeat**语句。

行为的程序语句明确描述了通过通信端口上的测试成分的动态行为，可以作为一个选择对象集或它们的组合来顺序地表达测试行为。一个交叉操作符允许对交叉序列或选择对象的说明。

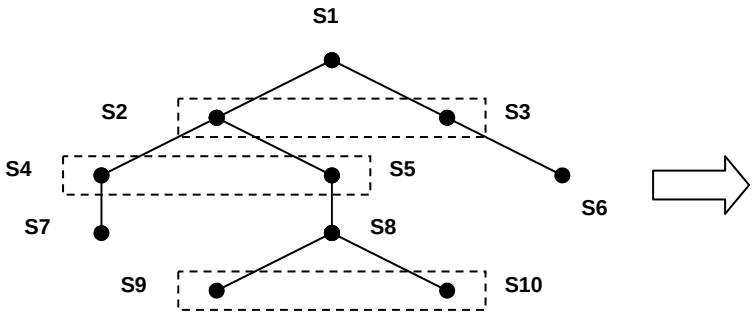
表 13：TTCN-3 行为的程序语句纵览

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语句的重新求值	**repeat**
交叉行为	**interleave { … }**
返回控制	**return**

20.1 选择性行为

20.1.0 概要

行为的一个更为复杂的形式是，语句序列被表示成形成执行路径树的可能的选择对象的集合，如图9所示：




```
S1;
alt {
  [] S2 {
       alt {
                 }
       }
  [] S3 { S6 }
}
```

图 9：选择性行为的例示

alt语句表示，由通信的接收和处理和/或定时器事件以及/或并行测试成分终止引起的测试行为的分支，，也就是说，**alt** 语句与TTCN-3的**receive**、**trigger**、**getcall**、**getreply**、**catch**、**check**、**timeout**和**done**操作有关。**Alt**语句表示将与一个特定快照相匹配的可能的事件集合（见章节20.1.1）。

注意： **alt**语句与TTCN-2中缩进排列的选择对象相似，但有明显的区别，例如：

a) 不能在通过使用布尔表达式来检查端口队列之后使一个选择对象无效；

b) 在**alt**语句中，函数不能作为一个选择对象被调用，除非在一个else防卫表达式（else guard，即[**else**]）是选择对象中最后的那个选择对象的情况下（见章节20.1.3）。

例：

```
// 嵌套的选择对象语句的使用
 :
alt {
  [] L1.receive(DL_REL_CO:*) {
       setverdict(pass);
       TAC.stop;
       TNOAC.start;
       alt {
         [] L1.receive(DL_EST_IN) {
              TNOAC.stop;
              setverdict(pass);
            }
         [] TNOAC.timeout {
              L1.send(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 选择对象行为的执行

进入一个**alt**语句时，照一张快照。快照被认为是一个测试成分的部分状态，这个测试成分包括：计算防卫选择对象分支的布尔条件的所有必需信息，所有有关的被终止的测试成分，所有相关的超时事件和相关的输入端口队列的队头消息、调用、应答以及例外。认为在**alt**语句中至少一个选择对象中被引用的任意测试成分、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测试步或作为默认被激活的测试步的顶层选择对象是相关的。快照语义的详细描述在TTCN-3的操作语义中给出（本文的第四部分（见参考书目））。

注意1： 快照仅仅是描述**alt**语句行为的概念上的方法，用于快照处理的具体算法见本文的第四部分（见参考书目）。

注意2： TTCN-3语义假设照快照是瞬时的，即没有延迟。在实际的实现中，照快照可能占用一些时间，可能出现竞态条件（race conditions）。这些竞态条件的处理超出了本标准的范围。

alt语句中的选择对象分支和被调用可选步中的顶层选择对象以及被激活为默认的可选步的最上层选择对象是按照它们的出现顺序被处理的。如果激活了数个默认，激活顺序决定了确定默认中的最上层选择对象的值的顺序。通过第21章中描述的默认机制到达活动的默认的选择对象分支。

各选择对象分支是可以被一个布尔表达式或一个else分支防卫的分支，即选择对象分支用**[else]**开始

在到达else分支时，总是选中和执行它们（见章节20.1.3）。

可被一个布尔表达式防卫的分支或激活一个可选步（可选步分支，*altstep-branch*），或以**done**操作（完成分支，*done-branch*）、**timeout**操作（超时分支，*timeout-branch*）或接收操作（接收分支，*receiving-branch*）开始，即**receive**、**trigger**、**getcall**、**getrepy**或**catch**。布尔防卫表达式的取值基于快照。如果没有定义布尔防卫或是其值为真，则认为*通过了*布尔防卫。按照以下的方式执行和处理分支：

如果通过了布尔防卫，则选中一个可选步分支（*altstep-branch*）。一个可选步分支的选择引起对被引用的可选步的调用，即调用该可选步且在这个可选步中继续对快照求值。

如果通过了布尔防卫，且如果指定的测试成分在快照的被停止成分的列表中，则选中一个完成分支（*done-branch*）。该选择引起**done**操作后的语句块的执行，而对**done**操作本身没有更进一步的影响。

如果通过了布尔防卫，且如果指定的超时事件在快照的超时事件列表中，则选中一个超时分支（*timeout-branch*）。该选择引起指定的超时（**timeout**）操作的执行，也就是说从超时队列（timeout-list）中移去该超时事件，并执行**timeout**操作后的语句块。

如果通过了布尔防卫，且如果快照中的一个消息（messages）、调用（calls）、应答（replies）或例外（exceptions）与接收操作的匹配标准相匹配，则选中一个接收分支（*receiving-branch*）。该选择导致执行接收操作，即从端口队列中移去相匹配的消息、调用、应答或例外，或许是把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，如果通过了布尔防卫但不符合匹配标准，也从端口队列中移去队头消息，在这种情况下不执行语句块中给定的选择对象。

注意3： TTCN-3语义描述了对作为测试成分的一系列不可以分割活动的快照的求值。该语义并不假设对快照求值没有延时。在对快照求值过程中，测试成分可以停止，定时器可以超时，新消息、调用、应答或例外可以进入该测试成分的端口队列。然而，这些事件并不改变实际的快照，因此快照的求值不考虑这些事件。

如果在**alt**语句中、被调用可选步的顶层选择对象或被激活的默认中没有选择分支被激活和执行的话，应该再次执行**alt**语句，也就是说，照一个新的快照，并以新快照重复对选择分支的求值。这个重复求值过程应该继续直到选中和执行一个选择分支，或是该测试例被另一个测试成分或测试系统停止。

如果测试成分被完全阻塞，测试例应该停止并指示一个动态错误。这就意味着没有选择对象被选中，没有相关的测试成分、定时器在执行，所有的相关端口包含至少一条消息、调用、应答或例外。

注意4: 照完整快照和为所有选择对象求值的反复过程对于描述**alt**语句的语义只是一个概念上的方法。实现这个语义的具体算法超出了本标准的范围。

20.1.2 选择对象的选择和去除选择

如果必要的话，可以利用置于选择对象方括号“[ ]”之间的一个布尔表达式去启动/禁止（enable/disable）一个选择对象。

对防卫一个选择对象的布尔表达式的求值可能有副作用。为了避免导致实际快照和成分状态之间不一致的这种副作用，应使用于初始化可选步内部定义的限制相同的限制（见章节16.2.2.1）。

方括号的开和关 “[” “]”应在每个选择对象的开始处出现，即便他们之间为空。这不仅对可读性有帮助，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：

```
// 带有布尔表达式的选择对象的使用（或防卫）
 :
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处开和关括号之间包含关键字**else**，定义**alt**语句的最后一个分支为一个else分支。这个else分支不应包含布尔表达式防卫的分支中允许的任何动作，即可选步调用、完成、超时或接收操作。如果这个else分支前（文本上顺序）没有处理过其它的选择对象，总是执行这个else分支的语句块。

例：

```
//  带有布尔表达式的选择对象的使用（或防卫）
 :
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，默认机制（见章节21）总是在所有选择对象的结束时才被调用。如果定义了一个**else**分支，将步调用默认机制，也就是说决不进入活动的默认部分。

注意1： 可以在可选步内使用**else**。

注意2： 允许使用一个**repeat**语句作为**else**分支的最后一个语句。

20.1.4 空

20.1.5 alt语句的再次求值

可以使用一个**repeat**语句来描述对**alt**语句的重新求值（见章节20.2）。

例：

```
alt {
  [] PCO3.receive {
        count := count + 1;
        repeat                  // repeat的用法
     }
  [] T1.timeout { }
  [] any port.receive {
        setverdict(fail);
        stop;
     }
}
```

20.1.6 作为选择对象的可选步的调用

TTCN-3允许在**alt**语句中作为选择对象调用可选步（见章节16.2.3）。

例：

```
  :
alt {
  [] PCO3.receive { }
  [] AnotherAltStep();  // 对作为alt语句的选择对象的可选步AnotherAltStep的显式调用
  [] MyTimer.timeout { }
}
  :
```

20.2 repeat语句

repeat语句引起对**alt**语句的重新求值（Re-evaluation），即照一个新的快照，并根据**alt**语句的选择对象的说明顺序来对这些选择对象求值。**repeat**语句应仅用作**alt**语句中选择对象的最后一个语句，或者是可选步定义中顶层的选择对象的最后一个语句。

如果一个**repeat**语句被用作是**alt**语句中选择对象的最后一个语句，则该**repeat**语句引起一个新的快照和对该**alt**语句的重新求值。

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;
     }
}
```

如果把**repeat**语句用作可选步定义中最上层的选择对象的最后一个语句，则该**repeat**语句引起一个新快照和对调用可选步的**alt**语句的重新求值。可以通过默认机制隐式调用该可选步（见章节21），也可以在**alt**语句中显式调用该可选步（见章节20.1.6）。

例2：

```
// 可选步中repeat的用法
```

```
altstep AnotherAltStep() runs on MyComponentType {
    [] PCO1.receive{
        setverdict(inconc);
        repeat                  // repeat的用法
       }
    [] PCO2.receive {}
}
```

20.3 交叉的行为

interleave语句允许描述**done**、**timeout**、**receive**、**trigger**、**getcall**、**catch**和**check**语句的交叉发生（interleaved occurrence）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的直接调用和用户定义函数的（直接和间接）调用，不应用在**interleave**语句中。此外，不允许用布尔表达式去防卫**interleave**语句的分支（即“[ ]”应该总是为空），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合代替。这个代替过程以及交叉的操作语义在本文的第四部分描述（见参考书目）。

interleave语句求值的规则如下：

a) 不论何时执行一个接收语句，直到到达下一个接收语句或交叉语句结束时才执行后面的非接收语句；

注意: 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**和**timeout**。非接收语句指所有可用于**interleave**语句内的非控制转移语句。

b) 求值后继续照下一个快照。

交叉的操作语义完整地定义在本问的第四部分（见参考书目）。

例：

```
// 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 返回语句

return语句终止函数的执行，并把控制返回到函数的调用点。**return**语句可以选择是否与一个返回值关联。**return**语句应该仅用在函数中。

例：

```
function MyFunction() return boolean {
     :
    if (date == "1.1.2000") {
        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; // 判定的显式返回
}
```

21 默认处理

21.0 概要

TTCN-3允许激活一个可选步为默认（Default，见章节16.2）。对每个测试成分，以列表的形式存储默认（即激活的可选步），并根据它们的激活顺序进行列表。TTCN-3使用**activate**（见章节21.3）和

deactivate（见章节21.4）操作对默认列表进行操作。**activate**语句添加一个新的默认到默认列表的末端，而**deactivate**语句从默认列表中移去一个默认。默认列表的一个默认可以通过作为相应**activate**操作结果产生的默认引用来识别。

表 14： 用于默认处理的TTCN-3语句一览表

用于默认处理的语句	
语句	**相关的关键字或符号**
激活一个默认	**activate**
停用（Deactivate）一个默认	**deactivate**

21.1 默认机制

如果由于实际的快照而导致的没有指明的选择对象能被执行，则在各**alt**语句结束处唤醒默认机制。被唤醒的默认机制调用默认类表中第一个可选步，并等待它的终止结果，可以是成功终止，也可以是非成功终止。非成功终止意味着该可选步中没有定义默认行为的（见章节16.2）的顶层选择对象可以被选中，而成功终止意味着选中并执行类一个顶层的选择对象。

在非成功终止的情况下，默认机制调用默认列表中的下一个默认。如果默认列表中的最后一个默认都非成功终止了，那么默认机制将返回到**alt**语句中它的调用点，也就是说，返回到**alt**语句的结尾，并指示一个非成功的默认执行。如果默认列表为空时，也将指示一个非成功的默认执行。

如果测试成分被阻塞，那么一个非成功默认执行可以导致一个新的快照，或一个动态错误。（见章节20.1）

在成功终止的情况下，默认可以利用一个**stop**语句停止测试成分，或在调用默认机制的**alt**语句之后立即继续测试成分的主控制流，或给测试成分照一个新的快照并对**alt**语句重新求值。后者用**repeat**语句只是它（见章节20.2）。如果选中的默认的顶层选择对象不以**repeat**语句结束，那么在**alt**语句之后立即继续该测试成分的控制流。

注意: TTCN-3不限制默认机制的实现。默认机制可以在每个**alt**语句的结尾处以被隐式调用的进程的形式实现， 也可以以只对默认处理负责的单独线程的形式实现，唯一的要求是根据其激活顺序调用默认机制。

21.2 缺省引用

默认引用（Default references）是激活默认的唯一的一种引用，这样的唯一的默认引用是在一个可选步被激活为一个默认时产生的。也就是说，一个默认引用是一个**activate**操作（见章节21.3）的结果。

默认引用具有特殊的预定义类型**default**。**default**类型变量可以用于处理激活测试成分中的默认。对于未定义的默认引用来说，特定的值空（**null**）是有效的，例如用于处理默认引用的变量初始化。

用在**deactivate**操作中使用默认引用来标识要被去激活的默认。

应该测试系统外部解析**default**类型的实际数据表示。这允许对抽象测试例的说明独立于任意实际TTCN-3运行环境之外，换句话说，TTCN-3不限制带有有关默认处理和识别机制的测试系统的实现。

例：

```
// 用于默认处理的变量的声明，并初始化该变量为为空
var default MyDefaultVar := null;
 :
// 为保存一个激活的默认MyDefaultVar的用法
MyDefaultVar := activate(MyDefAltStep()); // MyDefAltStep被激活为一个默认
 :
// 为去激活默认MyDefAltStep的MyDefaultVar的用法
deactivate(MyDefaultVar);
 :
```

21.3 激活操作

21.3.0 概要

activate操作用于把可选步激活为默认。**activate**操作将把被引用的可选步添加到默认列表中，并返回一个默认引用。默认引用是用于默认的一个唯一标识符，可以用在**deactivate**操作中去激活默认。

activate操作的影响局限于调用它的测试成分本地，这就意味着一个测试成分不能激活另外一个测试成分中的默认。

例：

```
 :
// 用于默认处理的变量的声明
var default MyDefaultVar := null;
 :
// 一个默认引用变量的声明和作为默认的一个可选步的激活
var default MyDefVarTwo := activate(MySecondAltStep());
 :
// 默认的一个可选步的MyAltStep的激活
MyDefaultVar := activate(MyAltStep()); // MyAltStep被激活为一个默认
 :
```

21.3.1 参数化可选步的激活

应该在相应的**activate**语句中提供要被激活为默认的参数化可选步（见章节16.2.1）的实际参数，这就是说在默认被激活时，实际参数与该默认绑定（不是在它被默认机制调用时）。

例：

```
 :
var default MyDefaultVar := null;
 :
MyDefaultVar := activate(MyAltStep2(5, MyVar);
    // MyAltStep2为激活为默认，带有实际参数5和MyVar的值。
    // 使用默认机制的一个MyAltStep2调用之前，MyVar的变化将不改变该调用的实际参数。
 :
```

21.4 去激活操作

去激活（**deactivate**）操作用来去激活默认，即以前被激活的可选步。**deactivate**操作将从默认列表中移去被引用的默认。

deactivate操作的影响局限于调用它的测试成分本地，这就意味着一个测试成分不能去激活另外一个测试成分中的默认。

不带参数的**deactivate**操作去激活一个测试成分的所有默认。

调用一个带有特定值**null**的**deactivate**操作没有任何作用。调用一个带有未定义的默认引用的**deactivate**操作，如已经被去激活的一个默认的旧引用或没有被初始化的默认引用变量，将导致一个运行错误。

例：

```
 :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
```

```
deactivate(MyDefaultVar); // 去激活MyAltStep
 :
deactivate; // 去激活所有其它默认，即我们这个情况中的MySecondAltStep和MyThirdAltStep
 :
```

22 配置操作

22.0 概要

配置操作用于建立和控制测试成分。这些操作应仅用于TTCN-3测试例、函数和可选步（也就是说不能用于模块控制部分）。

表 15: TTCN-3 配置操作一览表

配 置 操 作	
语句	**操作名**
创建并行测试成分	**create**
连接一个测试成分到另一个	**connect**
断开两个测试成分的连接	**disconnect**
映射成分端口到测试接口端口	**Map**
去除端口到测试系统接口的映射	**unmap**
获得MTC地址	**Mtc**
获得测试系统接口地址	**system**
获得自身地址	**Self**
启动测试成分的执行	**Start**
停止测试成分的执行	**Stop**
检查一个PTC的终止	**running**
等待一个PTC的终止	**Done**

22.1 创建操作

MTC是测试例启动时自动创建的唯一一个测试成分，所有其它的测试成分（并行测试成分， PTCs）都应该在测试执行期间用**create**操作显式地创建。一个测试成分被创建时，带有它所有的端口集合，且这些端口的输入队列为空。而且，如果定义一个端口类型为**in**或**inout**，该端口将处于监听状态（listening state），准备接收连接上的流量。

当显式或隐式创建测试成分时，所有成分变量和定时器重新设置它们的初值（如果有的话），且重新指派所有成分常量的值。

因为每个测试例终止时隐式地销毁所有的测试成分，在调用每个测试例时，应该完全地创建它所需要的测试成分和连接的配置

```
// 这个例子声明了一个地址类型变量，用来存储一个类型为MyComponentType的新创建的测试成分的引用，
// 它是create操作的结果。
 :
var MyComponenttype MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
```

create操作会返回一个新创建实例的唯一的成分引用（component reference），典型地，该成分的唯一引用将保存在一个变量中（见章节8.7），且可用于连接实例和通信目的（如发送和接收）。

可以在行为定义的任何地方创建测试成分，提供有关动态配置全部的灵活特性（即任意测试成分可以创建任意其它PTC）。测试成分引用的可见性（visibility）应遵循与变量相同的范围规则，为了可以在测试成分的创建范围之外引用它们，测试成分引用应作为一条消息中的一个参数或字段来传递。

22.2 连接和映射操作

22.2.0 概要

一个测试成分的端口可以连接到另一个测试成分或测试系统接口的端口。在两个测试成分连接的情况下，应使用连接（**connect**）操作。当连接一个测试成分到一个测试系统接口时，应使用映射（**map**）操作。**Connect**操作直接把一个端口连接到另一个端口，即一个端口的输出口连到另一个端口的输入口，而输入口则连到另一个端口的输出口。另一方面，**map**操作可以完全看作是定义如何引用通信流的名字翻译操作。

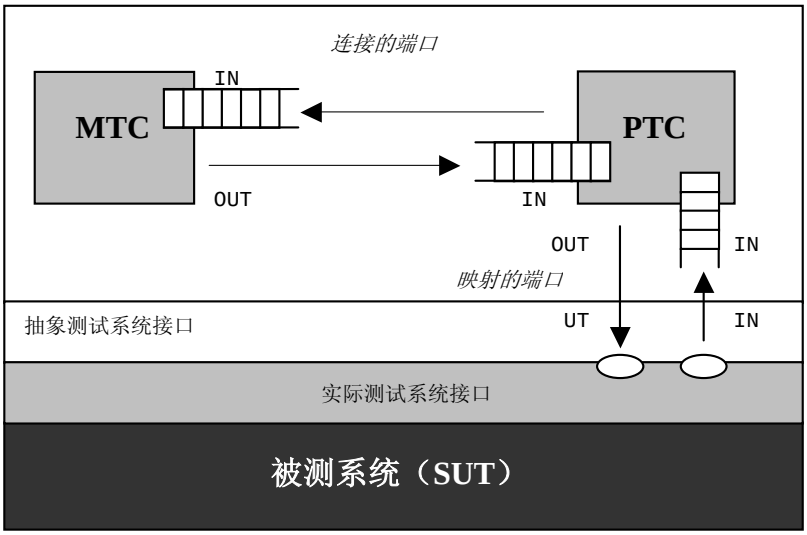


图 10： 连接和映射操作的图示

伴随着**connect**操作和**map**操作，通过对被连接的测试成分的成分引用和连接到的端口名字来标识被连接的端口。

Mtc操作标识主测试成分MTC，**system**操作标识测试系统接口（见章节22.4），且这两个操作都可用来标识和连接端口。

Connect和**map**操作都可以被除了模块控制部分之外的任意行为定义调用。然而，在它们被调用之前，要被连接的测试成分应该已经创建了，且它们的成分引用和相关的端口名字都应该是已知的。

Connect和**map**操作都允许把一个端口连接到一个以上的其它端口，但不允许连接到一个映射端口或映射到一个已连接的端口。

例：

```
// 假设在相应的端口类型和成分类型定义中正确定义和声明了端口Port1、Port2、Port3和PCO1。
 :
var MyComponentType  MyNewPTC;
 :
MyNewPTC := MyComponentType.create;
 :
 :
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
 :
 :
// 在这个例子中，创建了类型MyComponentType的一个新的成分，并把对它的引用存储在变量MyNewPTC中。
// 然后在connect操作中，这个新成分的Port1端口与MTC的Port3端口连接。
// 然后，利用map操作，新成分的Port2端口连接到测试系统接口的PCO1端口。
```

22.2.1 一致性连接和映射

对于**connect**和**map**操作，只允许一致性的连接。

有如下假设：

a) 要连接的端口为PORT1和PORT2；

b) inlist-PORT1定义端口PORT1输入方向的消息和过程；

c) outlist-PORT1定义端口PORT1输出方向的消息和过程；

d) inlist-PORT2定义端口PORT2输入方向的消息和过程；以及

e) outlist-PORT2定义端口PORT2输出方向的消息和过程。

connect操作被允许，当且仅当：

- outlist-PORT1 ⊆ inlist-PORT2且outlist-PORT2 ⊆ inlist-PORT1。

map操作（假设PORT2是测试系统接口端口）被允许，当且仅当：

- outlist-PORT1 ⊆ outlist-PORT2且inlist-PORT2 ⊆ inlist-PORT1。

在所有其他的情况下，都不允许**connect**和**map**操作。

因为TTCN-3允许动态配置和动态地址，所以在编译时，不是所有的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，应该在运行时执行，且发生错误时将导致一个测试例错误。

22.3 断开连接和取消映射操作

断开连接（**disconnect**）和取消映射（**unmap**）操作是**connect**和**map**操作的反向操作。它们执行操作来断开（以前已连接）测试成分端口的连接和取消（以前映射的）测试成分端口和测试系统接口中端口之间的映射。

如果有关的的成分引用和有关的端口名是已知的，那么任意成分都可以调用**disconnect**和**unmap**操作。如果要被移去的连接和映射已经被事先创建了，那么，**disconnect**和**unmap**操作只有一个作用。

例：

```
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
disconnect(MyNewComponent:Port1, mtc:Port3);     // 断开以前建立的连接
unmap(MyNewComponent:Port2, system:PCO1);        // 取消以前做的映射
```

22.4 MTC、System和Self 操作

成分引用（见章节8.7）有两个操作——**mtc**和**system**，它们分别返回对主测试成分和测试系统接口的引用。此外，操作可以用于返回调用它的测试成分引用。

例：

```
var MyComponentType MyAddress;
MyAddress := self; // 存储当前的成分引用
```

在成分引用上仅允许赋值和等值操作。

22.5 启动测试成分操作

一旦创建了一个PTC，且把连接行为和这个PTC绑定，就必须启动这个PTC的执行，这是通过**start**操作来完成的（创建PTC并不启动这个测试成分行为的执行）。区别**create**和**start**操作的原因是允许在实际运行测试成分之前完成连接操作。

Start操作会把必需的行为绑定到测试成分，通过对已定义函数的引用来定义这个行为。

例：

```
// 假设在相应的端口类型和成分类型定义中正确定义和声明了端口Port1、Port2、Port3和PCO1。
 :
var MyComponentType MyNewPTC;
 :
MyNewPTC := MyComponentType.create;      // 创建一个新测试成分
 :
connect(MyNewPTC:Port1, mtc:Port3);      // 新测试成分和
map(MyNewPTCt:Port2, system:PCO1);       // 它的环境的连接。
 :
 :
MyNewPTC.start(MyPTCBehaviour());   // 启动新测试成分。
 :
```

在一个启动测试成分（**start**）操作中调用的函数使用如下限制：

- 如果这个函数有参数，那么这些参数只能是输入（**in**）参数，即带有值的参数。
- 这个函数应该有一个**runs on**子句定义来引用与新创建成分相同的成分类型。
- 不能将端口和定时器传递进入该函数。

注意: 当创建测试成分时由于输入（**in**）和输入/出（**inout**）端口启动监听，这是测试成分同时启动测试执行，因此在这些端口的输入队列中可能有消息等待被处理。

22.6 停止测试成分操作

通过使用停止测试成分语句（**stop**），一个测试成分可以停止它自己的执行或是另一个测试成分的执行。如果一个测试成分不停止自身，而是被测试系统中另一个测试成分停止，那么使用这个被停止的成分引用来标识它。一个成分可以通过使用一个简单的**stop**执行语句来停止自身（见章节19.10），也可以通过在**stop**操作中选择自身地址来停止自身，如通过使用**self**操作。停止测试成分操作（**stop**）没有变量（arguments）。

例1：

注意1： **stop**操作可以用于MTC和PTC(s)，而**create**、**start**、**running**和**done**操作只能用于PTC(s)。

```
var MyComponentType MyComp := MyComponentType.create;   // 创建一个新的测试成分
MyComp.start(CompBehaviour());                          // 启动这个新的测试成分
 :
if (date == "1.1.2003") {
    MyComp.stop                 // 在1.1.2003时停止这个新的测试成分
}

 :
if ( cond1 ) {
     :  // 一些行为
    self.stop       // 通过使用self操作，测试成分停止自身
}
else {
```

```
    stop              // 通过使用一个简单的stop操作，测试成分停止自身
}
 :
```

如果被停止的测试成分是MTC，那么所有仍然运行的剩余的PTCs也将被停止，且测试例终止（见章节27.2）。

注意2： 一个PTC可以通过停止MTC来停止测试例的执行。

当一个测试成分终止时，无论是显式地使用**stop**操作或到达最初启动该测试成分的函数中的**return**语句时，还是隐式地到达该测试成分行为定义的结尾，应该释放所有资源。存储在一个已停止的成分引用中的任一变量将成为未定义的。

注意3: 未定义意味着值不能为任意计算流程所使用，也就是说，它与什么都无关，也不认为是空（**null**）。

测试例的终止和最后的测试判定计算在第25章中描述。

关键字**all**仅能被MTC用来停止所有正在运行的PTCs。在这种情况下，MTC不会被停止，在stop语句之后，它继续它自身的执行。

例2：

```
 :
all component.stop      // MTC停止测试例所有的PTCs，但不停止它自身的执行。
 :
```

注意4: 停止PTCs的具体机制在本文的讨论范围之外。

22.7 运行操作

运行操作（**running**）允许在一个测试成分上执行行为来确定在一个不同的测试成分上运行的行为是否已经完成，**running**操作只能用于PTCs 。**running**操作被认为是一个布尔表达式，因此它返回一个布尔值来指出特定的测试成分（或所有的测试成分）是否已经终止。与**done**操作相比，**running**操作可以自由地用在布尔表达式中。

当关键字**all**与**running**操作一起使用时，如果所有已启动且没有被另外一个测试成分显式停止的PTCs都在执行它的行为，则返回真（**true**），否则返回假（**false**）。

当关键字**any**与**running**操作一起使用时，如果至少有一个PTC在执行它的行为，则返回真（**true**），否则返回假（**false**）。

例：

```
if (PTC1.running)                    // 在一个if语句中running的用法
{
    // Do something!
}

while (all component.running != true) { // 在一个循环条件中urunning的用法
    MySpecialFunction()
}
```

22.8 完成操作

完成操作（**done**）允许在一个测试成分上执行行为去确定在一个不同的测试成分上运行的行为是否已经完成，**done**操作只能用于PTCs。

done操作的使用方式应该与接收操作或超时操作（**timeout**）的一样，这就意味着它不会用在布尔表达式中，但它能用来决定**alt**语句中的一个选择对象或作为一个行为描述中的独立语句。在后面这种情况中，一

个**done**操作可以被看作是一个只有一个选择对象的**alt**语句的简写，也就是说，它具有阻塞语义，因而提供了被动等待测试成分终止的能力。

当关键字**all**与**done**操作一起使用时，如果没有PTCs在执行它的行为，则返回真（**true**）；如果没有PTC被创建或启动，它也返回真（**true**）；否则返回假（**false**）。

当关键字**any**与**running**操作一起使用时，如果至少有一个已启动但没有显式地被另一个测试成分被停止的PTC完成了其行为的执行，则返回真（**true**），否则返回假（**false**）。

注意: TTCN-3的**done**操作与TTCN-2的DONE操作有相同的语义。

例：

```
// 选择对象中done操作的使用
:
alt {
    []  MyPTC.done {
            setverdict(pass)
        }

    []  any port.receive {
            goto alt
        }
}
:


// 下面的done操作作为独立语句（stand-alone statement）：
 :
all component.done;
 :

// 有如下含义：
 :
alt {
    []  all component.done {}
}
 :

// 因此，阻塞执行，直到所有并行测试成分都已经终止
```

22.9 使用成分数组

create、**connect**、**start**和**stop**操作不能直接在成分数组上工作，作为代替，应作为参数来提供数组的一个特定元素。对于成分来说，通过一个成分引用数组实现数组的作用，并把有关数组元素赋值给**create**操作的结果。

例：

```
// 这个例子表示了如何通过使用循环、在一个成分引用数组中存储被创建的成分引用，
// 为成分创建（creating）、连接（connecting）和运行（running）数组的结果建模。

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
     :
    var integer i;
    var MyPTCType1  MyPtcType[11];
     :
    for (i:= 0; i<=10; i:=i+1)
    {
        MyPtc [i] := MyPtcType1.create;
        connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCoordination);
        MyPtc [i].start(MyPtcBehaviour());
    }
     :
}
```

22.10 带有成分的any和all的使用总结

关键字**any**和**all**可以和表16给出的配置操作一起使用。

表 16：带有成分的Any 和 All

操作	允许的		例子
	any	**all**	
create			
start			
running	是，但仅来自于MTC	是，但仅来自于MTC	**any component.running** **all component.running**
done	是，但仅来自于MTC	是，但仅来自于MTC	**any component.done** **all component.done**
stop		是，但仅来自于MTC	**all component.stop**

23 通信操作

23.0 概要

TTCN-3支持*基于消息的*（*message-based*）和*基于过程的*（*procedure-based*）通信。而且TTCN-3允许检查输入队列的队头元素和利用*控制操作*去控制对端口的访问。

表 17： TTCN-3 通 信 操 作 一 览 表

通 信 操 作			
通 信 操 作	**关 键 字**	**可用于基于消息的端口**	**可用于基于过程的端口**
基于消息的通信（Message-based communication）			
发送消息	**send**	是	
接收消息	**receive**	是	
消息触发（Trigger on message）	**trigger**	是	
基于过程的通信（Procedure-based communication）			
调用过程调用	**call**		是
接受来自远程实体过程调用	**getcall**		是
回答来自远程实体过程调用	**reply**		是
（对一个已接受的调用）提出例外	**raise**		是
处理来自以前的调用的响应	**getreply**		是
捕获例外（从被调用实体）	**catch**		是
检查输入端口队列顶端元素（Examine top element of incoming port queues）			
检查接收到的消息/调用/例外/应答 msg/call/exception/reply received	**check**	是	是
控制操作（Controlling operations）			
清除端口（Clear port）	**clear**	是	是
清理并访问端口（Clear and give access to port）	**start**	是	是
停止对端口的访问（接收和发送）	**stop**	是	是

23.1 通信操作的通用格式

23.1.0 概要

象**send**和**call**这样的操作是用于测试成分之间以及被测系统SUT和测试成分之间交换信息的。为了解释这些操作的通用格式，可以把它们分成两组：

a) 一个测试成分发送消息（**send**操作）、调用一个过程（**call**操作）、回答一个已接收的调用（**reply**操作）或提出一个列外（**raise**操作）。这些动作合起来起来称为*发送操作*（*sending operations*）；

b) 一个测试成分消息（**receive**操作）、等待一条消息（**trigger**操作）、接受一个过程调用（**getcall**操作）、收到一个以前被调用过程的应答（**getreply**操作）或捕获一个例外（**catch**操作）。这些动作合起来起来称为*接收操作*（*receiving operations*）。

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调用操作情况下，由一个*响应*和*例外处理*部分组成。

发送部分：

- 指定要发生指定操作的端口；
- 定义要传输的信息的值；
- 给定一个在一对多连接的情况下唯一标识通信伙伴（communication partner）的地址表达式（可选的）。

端口名、操作名和值应出现在所有的发送操作中。通信伙伴的标识符（用关键字**to**表示）是可选的，且仅在需要显式地标识接收实体的一对多的情况下才需要被指定。

例1：

发送部分			（可选的）
端口和操作	值 部 分	（可选的）地址表达	处
MyP1.**send**	(MyVariable + YourVariable – 2)	**to** MyPartner;	

仅在基于过程的通信的情况下，才需要响应和例外处理。调用操作（**call**）的响应和例外处理是可选的，它们只有在如下情况下才是必须的：被调用过程返回一个值；或被调用过程有输出（**out**）或输入/出（**inout**）参数值，且在调用成分中需要这些参数的值；或被调用过程可能引起需要调用成分处理的例外。

调用操作的响应和例外处理部分使用**getreply**和**catch**操作提供必需的功能性（functionality）。

例2：

发送部分（Send part）			（可选）响应和例外处理部分 (Optional) response and exception handling part
端口和操作	值部分	（可选） 地址表达式	

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）*赋值*部分组成。

接收部分：

a) 指定操作发生的端口；

b) 定义一个匹配部分，指定将匹配该语句的可接收的输入；

c) 给定一个唯一标识通信伙伴的（可选）地址表达式（在一对多连接的情况下）。

端口名、操作名和值应出现在所有的接收操作中。通信伙伴的标识符（用关键字**from**表示）是可选的，且仅在需要显式标识接收实体的一对多的情况下才需要被指定。

一个接收操作的（可选的）赋值部分是可选的。对于基于消息的端口来说，当需要它去存储接收到的消息的时候，赋值部分是必须的。在基于过程的端口的情况下，它用于存储一个已接收的调用的输入（**in**）和输入/出（**inout**）参数或存储例外。对于赋值部分，需要强类型机制，例如用于存储消息的变量类型应该与输入消息的类型相同。

此外，赋值部分也可以用于把一个消息、例外、应答（**reply**）或调用（**call**）的发送方地址赋给一个变量。这对一对多连接是很有用的，例如，可以接收来自不同成分的相同的消息或调用，但是这个消息、例外、应答（**reply**）或调用（**call**）必须发送回原始的发送成分。

例：

接 收 部 分				（可选）赋 值 部 分		
端 口 和 操 作	匹 配 部 分	（可选）地址表达式		（可选）值 赋 值	（可选）参 数 值 赋 值	（可选）发 送 者 赋 值
MyP1.**getreply**	(AProc:{?} **value** 5)		->		**param** (V1)	**sender** APeer

接 收 部 分				（可选）赋 值 部 分		
端 口 和 操 作	匹 配 部 分	（可选）地址表达式		（可选）值 赋 值	（可选）参 数 值 赋 值	（可选）发送者 赋 值
MyP2.**receive**	(MyTemplate(5,7))	**from** APeer	->	**value** MyVar		

23.2 基于消息的通信

23.2.0 概要

基于消息的通信是基于异步消息交换的通信。它在**send**操作上是非阻塞的，如图11所示，发送方在**send**操作之后立即继续它的处理过程。接收方在**receive**操作上被阻塞，直到它去处理接收消息。

除了**receive**操作外， TTCN-3还提供了一个触发（**trigger**）操作，来根据一定的匹配标准过滤来自给定输入端口上接收信息流的消息。从该端口移去队列头部不满足匹配标准的消息，而不采取进一步的行动。

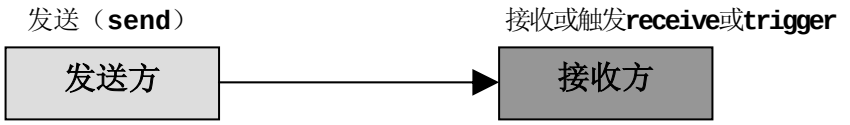


图 11: 异步发送和接收的图示

23.2.1 发送操作

send操作用于把一个值放到一个输出端口队列，可以通过引用一个模板、变量或一个常数来指定这个值，或可用一个表达式（当然是能给出明确值的）内嵌在一个语句行内来定义这个值。如果被发送的值的类型模糊，那么在嵌入地定义值的时候，应使用可选的类型字段。

send操作应该只能用在基于消息的（或混合型的）端口，且要发送的值的类型应该在该端口类型定义的输出类型列表中。

例1：

```
MyPort.send(MyTemplate(5,MyVar));        // 通过MyPort发送带有实参5和MyVar的模板

MyPort.send(5);                          // 发送整型值5
```

在一对多连接的情况下，应该唯一地指定通信伙伴，用关键字**to**表示。

例2：

```
MyPort.send(charstring:"My string") to MyPartner;   // 给存储在变量MyPartner中的成分引用
                                                    // 发送串"My string"

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
                                  // 给MyPartner发送一个算术表达式的结果
```

23.2.2 接收操作

23.2.2.0 概要

receive操作用于从一个输入消息端口队列中接收一个值。可以通过引用一个模板、变量或一个常数来指定这个值，或可用一个表达式（当然是能给出明确值的）内嵌在一个语句行内来定义这个值。在内嵌地定义值的时候，应使用可选的类型字段来避免任何要接收值类型的含糊。**receive**操作应该只能用在基于消息的（或混合型的）端口，且要接收的值的类型应该被包含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队头消息满足所有**Receive**操作相关的匹配机制时，**Receive**操作从相关的输入端口队列中移去该消息。不会发生把输入的值绑定到表达式或模板术语上。

如果匹配不成功，输入端口队列中的队头消息就不能被移去，也就是说，如果**receive**操作是**alt**语句中的一个选择对象且它不成功的话，测试例的执行从**alt**语句中的下一个选择对象开始继续。

匹配标准与要接收消息的类型和值有关。要接收消息的类型和值可以来自于一个模板或一个表达式的结果值（当然是能给出明确值的），应使用**receive**操作匹配标准中的可选类型字段来避免要接收的值类型的模糊。

注意: 通过防止解码器重接收到的不是用编码属性描述来编码的消息中产生一个抽象值，编码属性也以隐式的方式参加匹配。

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MyPort上匹配一个满足由模板MyTemplate定义的条件的一条消息

MyPort.receive(A<B);                    // 匹配一个依赖于A<B结果的布尔值

MyPort.receive(integer:MyVar);          // 在MyPort上匹配一个整型值和MyVar的值

MyPort.receive(MyVar);                  // 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，且可以重新获得发送消息的成分的地址并存储到一个变量中，用符号'->' 和关键字**value**来表示。

也可能重新获得和存储消息发送者的成分引用或地址，用关键字**sender**表示。

例2：

```
MyPort.receive(MyType:?) -> value MyVar;     // 把接收到的消息的值赋给MyVar

MyPort.receive(A<B) -> sender MyPeer;        // 把发送者的地址赋给MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// 把接收到的消息的值赋给MyVar，并把发送者的地址赋给MyPeer。
```

23.2.2.1 接收任意消息

对于不带有用于要接收消息的类型和值匹配标准的参数列表的**receive**操作，如果满足其他所有匹配标准，则移去输入端口队列的队头消息（如果有的话）。

注意: 这与TTCN-2中的OTHERWISE语句等价。

通过*ReceiveAnyMessage*接收到的消息不会赋值给一个变量。

例：

```
MyPort.receive;              // 从MyPort移去队列头部值

MyPort.receive from MyPeer; // 如果发送者是MyPeer的话，从MyPort移去队列的队头值

MyPort.receive -> sender MySenderVar;   // 从MyPort移去队列的队头值，并把发送方地址赋给MySenderVar
```

23.2.2.2 在任意端口上接收

使用关键字**any**在任意端口上接收消息。

例：

```
any port.receive(MyMessage);
```

23.2.3 触发操作

23.2.3.0 概要

触发（**trigger**）操作从相关的输入端口队列中移去头部消息。如果头部消息满足匹配标准的话，**trigger**操作与**receive**操作的一样。如果头部消息不满足匹配标准的话，**trigger**操作会从输入端口队列中移去队头消息，但没有进一步的动作。**trigger**操作应该只能用在基于消息的（或混合型的）端口，且要接收的值的类型应该被包含在该端口类型定义的输入类型列表中。

注意: 章节22.2.2.0中用于**trigger**操作的“注意”仍然有效。

在一个行为描述中，能够把**trigger**操作用作一个独立的语句。在后面的这种情况下，**trigger**操作可以被看做是只有一个选择对象的**alt**语句的简写，也就是说，它有阻塞语义，并因此提供等待下一个消息匹配指定的模板或那个队列上的值的能力。

例1：

```
MyPort.trigger(MyType:?);
// 说明在端口MyPort上，对观察到的第一个带有一个任意值的MyType类型消息的接收将触发该操作
```

trigger操作需要端口名、用于类型和值的匹配标准、一个可选的**from**限制（即通信伙伴选择）以及一个可选的赋值（匹配消息并将发送方测试成分赋给变量）。

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（即满足所有匹配标准的消息）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。另外发送方测试成分的引用将被重新获得，并存储在变量MyPartner中。

MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// 在接收一个后来存储在变量MyVar中的任意整型值时触发。发送方成分的引用将被存储在变量MyPartner中。
```

23.2.3.1 在任意消息上的触发

不带参数（argument）列表的**trigger**操作会在任意消息的接收上触发。因此，它的含义与接收任意消息的含义相同。被*TriggerOnAnyMessag*接受到的消息不会赋值给一个变量。

例：

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 在任意端口上的触发

在任意端口的消息上触发使用关键字**any**。

例：

```
any port.trigger
```

23.3 基于过程的通信

23.3.0 概要

基于过程的通信是去调用远程实体的过程。TTCN-3支持*阻塞的*（*blocking*）和*非阻塞的*（*non-blocking*）基于过程的通信。*阻塞的*基于过程的通信是在调用方和被调用方阻塞，*非阻塞的*基于过程的通信是仅在被调用方阻塞。应该根据第23章中的规则来描述用于*非阻塞的*基过程的通信 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（CALLER）通过**call**操作调用被调用方（CALLEE）中的一个远程过程，被调用方通过**getcall**操作接受这个调用，并通过使用一个**reply**操作来回答调用方或提出（**raise**操作）一个例外来响应调用。调用方通过使用**getreply**操作或**catch**操作来处理应答或例外。在图12中，用虚线表示调用方和被调用方的阻塞。

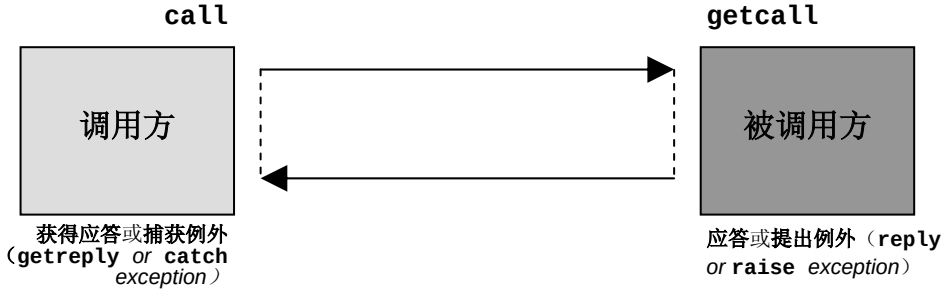


图 12：阻 塞 的 基 于 过 程 的 通 信 的 图 示

*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**call**操作调用被调用方中的一个远程过程，并继续它自身的执行，即并不等待一个应答或例外。被调用方通过**getcall**操作接收这个调用，并执行被请求的过程。如果这个执行不成功的话，被调用方可以产生一个例外去通知调用方。调用方可以通过使用**alt**语句中的**catch**操作来处理例外（exception）。在图13中，用虚线表示被调用方的阻塞（直到调用处理结束并可能提出一个例外）。

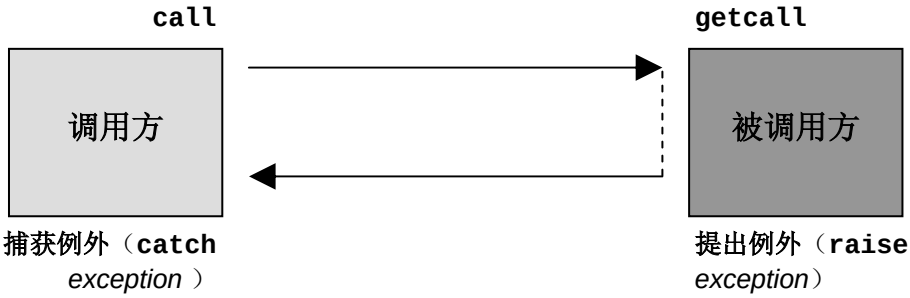


图 13：非阻塞的基于过程的通信的图示

23.3.1 调用操作

23.3.1.0 概要

调用（**call**）操作用于指定一个测试成分调用被测系统SUT或另一个测试成分中的过程。**call**操作应该仅用在基于过程（或混合型）的端口上。调用操作发生的端口的类型定义应该在它的输出（**out**）或输入/出（**inout**）列表中包括过程名，也就是说必须允许在这个端口上调用这个过程。

在**call**操作的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一个特征，也可以是在语句行内嵌入式地特征。这个特征的所有输入（**in**）和输入/出（**inout**）应有一个特定值，也就是说不允许使用匹配机制，如*AnyValue*。

不使用**call**操作的特征参数为输出（**out**）或输入/出（**inout**）参数重新获得变量名。过程返回值和输出（**out**）或输入/出（**inout**）参数对变量的实际赋值，应该在**call**操作的响应和例外处理部分中通过**getreply**和**catch**操作显式地进行。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：

```
// 给定
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// MyProc的一个调用
MyPort.call(MyProc:{ -, MyVar2}) {      // 用于MyProc的调用的嵌入在语句行内的特征模板
    [] MyPort.getreply(MyProc:{?, ?}) { }
}

// 以及MyProc的另一个调用
MyPort.call(MyProcTemplate) {           // 为MyProc的调用使用特征模板
    [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

在一对多连接的情况下，应该唯一地指定通信伙伴，用关键字**to**表示。

例2：

```
MyPort.call(MyProcTemplate) to MyPeer {         // 在MyPeer调用MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

23.3.1.1 处理对一个调用的响应和例外

在*非阻塞的*基于过程的通信的情况下，或者如果没有使用**nowait**选项的话（见章节23.3.1.2），通过使用作为**alt**语句的选择对象的**getreply**操作（见章节23.3.2）和**catch**操作（见章节23.3.6）来完成对**call**操作的响应和例外的处理。

在*阻塞的*基于过程的通信的情况下，在**call**操作的响应和例外处理部分中，通过使用**getreply**操作（见章节23.3.2）和**catch**操作（见章节23.3.6）来完成对**call**操作的响应和例外的处理。

一个**call**操作的响应和例外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来描述对该调用的可能的响应和例外。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，这意味着不允许使用**else**语句分支和调用**alt**可选步。

如果必要的话，可以通过置于选择对象的方括号“[ ]”之间的一个布尔表达式来激活/去激活这个选择对象。

一个**call**操作的响应和例外处理部分的执行就象一个没有任何活动的默认的**alt**语句。这就意味着一个相关的快照包含为这个（可选的）布尔防卫定值所需的所有信息，可以包括被调用过程所运行之端口的顶端元素（如果有的话），也可以包括一个由监视这个调用的（可选的）定时器产生的超时例外（见章节23.3.1.2）。

在响应和例外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。为了避免意外的副作用，将使用与**alt**语句中一样的布尔防卫规则（见章节20.1.1）。

例：

```
// 给定
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
    exception (ExceptionTypeOne, ExceptionTypeTwo);
 :

// MyProc3的调用
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 处理调用的超时例外

call操作可以选择是否包含一个超时，这个超时作为一个明确的浮点类型值或常数来定义，并在启动**call**操作后定义这个时间的长短。超时例外应该由测试系统产生。如果在**call**操作中没有出现超时值部分，那么将不会产生超时例外。

例1：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

  [] MyPort.getreply(MyProc:{?, ?}) { }

  [] MyPort.catch(timeout) {                // 20ms后超时例外
        setverdict(fail);
        stop;
     }
}
```

在**call**操作中使用关键字**nowait**来代替超时例外值，允许不用等待响应、被调用过程提出（raised）的例外或超时例外而继续去调用一个过程。

例2：

```
MyPort.call(MyProc:{5, MyVar}, nowait);      // 调用方测试成分不等待MyProc的终止而继续自身的执行
```

在使用了关键字**nowait**的地方，必须通过随后的**alt**语句中的 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例外。

23.3.1.3 调用不带返回值、输出参数、输入/出参数和例外的阻塞类过程

一个阻塞的过程可以没有返回值，没有输出和输入/出参数，也可以不提出例外。用于这样过程实例的调用操作应该有一个响应和例外处理部分用统一的方式去处理这个阻塞。

例：

```
// 给定
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 调用非阻塞类过程

非阻塞的过程没有输出和输入/出参数，没有返回值，且通过关键字**noblock**在相关的特征定义中指示非阻塞特性。

对于一个非阻塞类的过程的**call**操作应该没有响应和例外处理部分，不提出超时例外，也不使用关键字**noblock**。

非阻塞类的过程可能提出的例外必须使用随后的**alt**语句中的**catch**操作来把它们从端口队列中移去。

23.3.2 getcall操作

23.3.2.0 概要

getcall操作用来说明一个测试成分接受来自被测系统或另一个测试成分的调用。**getcall**操作应该仅用在基于过程的（或混合型的）端口上，且应该在该端口类型定义的允许引入过程列表中包含要被接收的过程调用的特征。

当且仅当满足**getcall**操作操作相关的匹配标准时，**getcall**操作会从输入端口队列中移去队列头部的那个调用。这些匹配标准与要被处理的调用特征和通信伙伴有关，用于特征的匹配标准可以在语句行内嵌入式地说明或从一个特征模板派生。

在一对多连接的情况下，**getcall**操作可能会限制到一个特定的通信伙伴，这个限制用关键字**from**表示。

例1：

```
MyPort.getcall(MyProc(5, MyVar));                // 在MyPort接收MyProc的一个调用

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;  // 在MyPort接受一个来自MyPeer的MyProc的一个调用
```

getcall操作的特征参数不应用来为输入（**in**）和输入/出（**inout**）参数传入变量名。输入（**in**）和输入/出（**inout**）参数值到变量的赋值应该在**getcall**操作的赋值部分中进行。这允许与模板使用方式相同的方式的**getcall**操作中特征模板的使用方式可以为各类型所使用。

getcall操作的赋值部分（可选的）包含输入（**in**）和输入/出（**inout**）参数值到变量的赋值和调用成分地址的取回。关键字**param**用于取回调用的参数值。

当必须取回发送方地址时（例如，在一对多配置中，用来寻找对调用伙伴（calling party）的应答**reply**或例外的地址，使用关键字**sender**。

例2：

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// MyProc的输入（in）或输入/出（inout）参数值赋给MyPar1Var和MyPar2Var。

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// 在MyPort接收MyProc的一个带有输入（in）或输入/出（inout）参数5和MyVar的调用。
// 取回调用伙伴（calling party）的地址并存储在MySenderVar中。

// 下面getcall的例子说明了使用匹配属性和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);
// 第一个输入/出（inout）参数不重要或不被用到

// 下面的例子将解释赋值输入（in）或输入/出（inout）参数到变量中去的可能性。假设下面的特征用于被调用过程：

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// 把参数A、B和E赋值给变量MyVarA、MyVarB和MyVarE。不需考虑输出（out）参数D。

MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// 用于输入（in）或输入/出（inout）参数赋值到变量的替代表示
// 注意，赋值列表中的名字指用在MyProc2特征中的名字。

MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
// 在后面（further）的测试例执行中，只有输入/出（inout）参数值需要被用到。
```

23.3.2.1 接收任意调用

不带用于特征匹配标准的参数列表的**getcall**操作，如果所有满足其他所有匹配标准，将移去输入端口队列头部的调用（如果有的话）。通过*AcceptAnyCall*接收的调用的参数不会赋值给变量。

例：

```
MyPort.getcall;                  // MyPort移去队列头部的调用

MyPort.getcall from MyPartner;  //从端口MyPort移去一个来自MyPartner的调用

MyPort.getcall -> sender MySenderVar;   // 从MyPort移去一个调用并取回调用实体的地址
```

23.3.2.2 在任意端口上获得调用

用关键字**any**表示在任意端口上的**getcall**操作。

例：

any port.**getcall**(MyProc)

23.3.3 应答操作

reply操作用于根据过程特征回答一个以前接受的调用。**reply**操作应该仅用在基于过程的（或混合型的）端口上，且端口的类型定义应该包括**reply**操作所属的过程名。

注意: 不能总是静态地检查一个已接收的调用和一个应答操作之间的关系，对于测试来说，允许指定一个**reply**操作但不带有相关联的**getcall**操作。

reply操作的值部分由一个带有相关实参列表的特征引用和（可选的）返回值组成，这个特征可以定义在特征模板格式中，或在语句行内嵌入式地定义它。特征的所有输入（**in**）和输入/出（**inout**）参数应该有特定的值，也就是说不允许使用象*AnyValue*这样的匹配机制。

在一对多连接的情况下，应该显式地指定通信伙伴，且该通信伙伴应该是唯一的，用关键字**to**表示。

如果要返回一个值给调用方，要用关键字**value**来显式说明。

例：

```
MyPort.reply(MyProc2:{ - ,5});              // 回答接到的MyProc2的一个调用

MyPort.reply(MyProc2:{ - ,5}) to MyPeer;    // 回答接到的来自MyPeer的MyProc2的一个调用

MyPort.reply(MyProc3:{5,MyVar} value 20);   // 回答接到的MyProc2的一个调用
```

23.3.4 获得应答操作

23.3.4.0 概要

getreply操作用于处理来自先前被调用过程的应答， 仅能用在基于过程（或混合型的）端口上。

当且仅当满足相关的**getreply**操作的匹配标准时，**getreply**操作会移去输入端口队列队头的应答。这些匹配标准与被处理过程特征和通信伙伴有关，用于特征的匹配标准可以在语句行内嵌入式地说明，或从一个特征模板上派生。

可以用关键字**value**说明对一个返回值的匹配。

在一个一对多连接的情况下，可能限制一个**getreply**操作到一个特定的通信伙伴，用关键字**from**表示这个限制。

例1：

```
MyPort.getreply(MyProc:{5, ?} value 20);    // 接收一个MyProc的应答，该应答带有两个输出（out）或
                                            // 输入/出（inout）参数和一个返回值20

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer;  // 接收来自MyPeer的MyProc的一个应答
```

getreply操作的特征参数不应用来为输出（**out**）和输入/出（**inout**）参数传入变量名。输出（**out**）和输入/出（**inout**）参数值到变量的赋值应该在**getreply**操作的赋值部分中进行。这允许与模板使用方式相同的方式的**getreply**操作中特征模板的使用方式可以为各类型所使用。

getreply操作的赋值部分（可选的）包含输出（**out**）和输入/出（**inout**）参数值到变量的赋值和应答发送方地址的获得。关键字**param**用于取回应答的参数值。当需要取回发送方地址时，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，两个输出（out）或输入/出（inout）参数赋值给变量MyPar1和MyPar2。

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender MySender;
// 在后面的测试执行中，不考虑第一个参数值。取回发送方成分的地址并存储在变量MySender中。

// 下面的例子描述了把输出（out）或输入/出（inout）参数赋值给变量的可能性。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 获得任意应答

如果满足所有其他匹配标准，那么不带有用于特征匹配标准的**getreply**操作将会从输入端口队列的队头移去一个应答（如果有的话）。通过*GetAnyReply*接收到的参数或返回值不会分配给变量。

例：

```
MyPort.getreply;                 // 移去MyPort队头的应答

MyPort.getreply from MyPeer;     // 移去MyPort队头来自MyPeer的应答

MyPort.getreply -> sender MySenderVar;  // 移去MyPort队头的应答，取回发送方实体的地址
```

23.3.4.2 在任意端口上获得应答

使用关键字**any**在任意端口上获得一个回答。

例：

any port.**getreply**(Myproc)

23.3.5 Raise操作

raise操作用于提出一个例外，仅能在基于过程的（或混合型的）端口上提出例外。例外是对导致例外事件的已接受的过程调用的一个反应。应在被调用过程特征中说明例外的类型。端口的类型定义应该在其已接受过程调用的列表中包含例外所属的过程名。

注意: 不可能总是静态地检查一个已接受的调用和**raise**操作的关系，对于测试来说，允许说明一个**raise**操作与一个**getcall**操作无关。

raise操作的值部分由跟着例外值的特征引用组成。

例外被作为类型描述，因此例外值可能来自一个模板，或是一个表达式的结果（当然表达式可以是一个确定的值）。在必需去避免发送值类型的含糊的情况下，对**raise**操作将使用值定义中的可选类型字段。

在一对多连接的情况下，应该唯一指定通信伙伴，用关键字**to**表示。

例：

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
// 在MyPort提出一个带有算术表达式结果值的例外

MyPort.raise(MyProc, integer:5));   // 为MyProc提出带有整型值5的一个例外。

MyPort.raise(MySignature, "My string") to MyPartner;
// 在MyPort提出带有值"My string"的一个例外，并把它发送给MyPeer
```

23.3.6 捕获操作

23.3.6.0 概要

捕获（**catch**）操作用于捕获一个例外，这个例外由对等实体提出，作为对一个过程调用的响应。**Catch**操作仅能被用在基于过程的（或混合型的）端口上，应该在提出例外的过程特征中说明被捕获例外的类型。例外被说明成类型，因此可以象消息一样对待它，例如，可以用模板来区分相同例外类型的不同值。

当且仅当相关输入端口队列头部的例外满足所有**catch**操作相关的匹配标准时，**catch**操作将移去该队列头部的这个例外。不会发生输入值到表达式术语（terms）或模板例外的绑定。

在一对多连接的情况下，**catch**操作可能被限制到一个特定的通信伙伴，用关键字**from**表示这个限制。

例1：

```
MyPort.catch(MyProc, integer: MyVar);   // 在端口捕获MyProc提出的一个值为MyVar的整型例外

MyPort.catch(MyProc, MyVar);            // 上个例子的一个替换

MyPort.catch(MyProc, A<B);              // 捕获一个布尔型例外

MyPort.catch(MyProc, MyType:{5, MyVar}); // 一个例外值的嵌入式模板

MyPort.catch(MyProc, charstring:"Hello")from MyPeer;    // 从MyPeer捕获"Hello"例外
```

catch操作的赋值部分（可选的）包含例外值的赋值和调用成分地址的取回。用关键字**value**取回一个例外值，当需要取回发送方地址的时候，使用关键字**sender**。

例2：

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// 捕获来自MyPartner的一个例外，并把它的值赋给MyVar。

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// 捕获来自MyPartner的一个例外，把它的值赋给MyVar，并取回发送方地址。
```

catch操作可以是**call**操作的响应和例外处理中的一部分，或用于决定**alt**语句的一个选择对象。如果**catch**操作用在**call**操作的接受部分，关于端口名的信息和用来指示提出例外的过程的特征引用是多余的，因为这些信息跟在**call**操作后面。然而，由于可读性原因（如在复杂的**call**语句的情况下），应该重复这些信息。

23.3.6.1 超时例外

catch操作捕获一个特别的超时例外（**timeout**）。对于被调用过程不在预定的时间内既不响应也不提出例外的情况，超时例外是一个紧急出口。（见章节23.3.1.2）。

例：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {                // 20ms之后，超时例外
        setverdict(fail);
        stop;
     }
}
```

捕获超时（**timeout**）例外应该限制在一个调用的例外处理部分。对于处理超时（**timeout**）例外的**catch**操作来说，不允许更多的匹配标准（包括一个**from**部分）和赋值部分。

23.3.6.2 捕获任何意外

没有参数列表的**catch**允许捕获任意有效的例外。 最通常的情况是不使用关键字**from**和一个赋值部分，这个语句也捕获**timeout**例外。

例：

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

23.3.6.3 在任意端口上的捕获

使用关键字**any**在任意端口上捕获一个例外。

例：

```
any port.catch;
```

23.4 检查操作

23.4.0 概要

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队头元素，但不从该队列移去队头元素。**check**操作必须在基于消息的端口上处理某个类型值，在基于过程的端口上区别要接收的调用、要捕获的例外和来自先前调用的应答。

check操作使用接收操作（**receive**、**getcall**、**getreply**和**catch**）和它们的匹配和赋值部分一起去定义必须检查的条件，并在必要时提取它的值。

要检查的是输入端口队列的队头元素（不可能检查队列的内部）。如果队列是空的，则**check**操作失败。如果队列不空，复制队列的队头元素，并在这个复制的队头元素上执行**check**操作中指定的接收操作。如果接收操作失败，则**check**操作失败，即不满足匹配标准。在这种情况下，丢弃队头元素并按正常方式继续测试执行，为**check**操作的下一条语句或选择对象定值。如果接收操作成功，则**check**操作成功。

按照错误的方式使用**check**操作，如在一个基于消息的端口检查例外，会导致一个测试例错误。

注意: 在大多数情况先，可以静态地检查**check**操作的正确用法，即在编译前检查。

例：

```
MyPort1.check(receive(5));  // 检查一个值为5的整型消息。

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));
// 在端口MyPort上检查返回值为20且两个输出（out）或输入/出（inout）参数的值为5和MyVar的来自过程MyProc的一个应答

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 检查任意操作

没有参数列表的**check**操作允许检查在输入端口队列中是否有元素等待处理。*CheckAny*操作允许通过使用一个**from**子句区别不同的发送方（在一对多连接的情况下），以及通过使用带有一个**sender**子句的简写的赋值部分来重新获得发送方。

例：

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 在任意端口上的检查

使用关键字**any**在任意端口上进行检查。

例：

```
any port.check;
```

23.5 控制通信端口

23.5.0 概要

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**：移去一个输入端口队列的内容；
- **start**：启动对一个端口的监听和对端口的访问；
- **stop**： 停止对端口的监听和取消在该端口上对发送操作的允许。

23.5.1 清除端口操作

clear操作移去指定端口的输入（*incoming*）队列的内容。如果输入队列已经是空的，那该操作什么也不做。

例：

```
MyPort.clear;   // 清除端口MyPort
```

23.5.2 启动端口操作（The Start port operation）

如果一个端口定义为允许接收操作，如**receive**、**getcall**等， **start**操作清理（clears）指定端口的输入队列，并启动对通过该端口上的通信流的监听。 如果定义该端口允许发送操作，那么象**send**、**call**、**raise**等操作可以在这个端口上执行。

例：

```
MyPort.start;   // 启动MyPort
```

默认地，一个成分在被创建时，其所有端口都被隐式地启动了。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被启动没有被停止的端口。

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**，**stop**操作导致停止对指定端口的监听。如果定义端口允许发送操作，那么停止端口则不允许执行**send**、**call**、**raise**等操作。

例1：

```
MyPort.stop;    // 停止MyPort
```

注意: 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应该在端口的工作被挂起之前全部执行了。

例2：

```
MyPort.receive (MyTemplate1) -> RecPDU;  // 解码接收到的值，并和MyTemplate1匹配，将匹配值存如变量RecPDU

MyPort.stop;                  // 没有定义在stop操作后的接收操作被执行（除非通过后面的start操作重新启动该端口）

MyPort.receive (MyTemplate2);       // 这个操作没有被执行
```

23.6 any和all与端口一起使用

如表18所示，关键字**any**和**all**可以和配置和通信操作一起使用。

表 18：与 端 口 一起 的 Any 和 All

操　　作	被允许		例子
	any	all	
receive, trigger, getcall, getreply, catch, check)	是		any port.receive
connect / map			
start, stop, clear		是	all port.start

24 定时器操作

24.0 概要

TTCN-3支持许多定时器操作，这些操作可以用在测试例、函数、可选步和模板控制中。

假设每个TTCN-3中声明定时器的范围单元维护它自己的*运行的定时器列表*（*running-timers list*）和*超时列表*（*timeout-list*），即所有运行的定时器的列表和所有超时的定时器列表。超时列表是测试例执行时照的快照的一部分，如果在定时器范围单元中一个定时器被启动、停止、超时或执行**timeout**操作，那么更新超时列表。

注意1： *运行的定时器列表*和*超时列表*只是一个概念上的（conceptual）列表，并不限制定时器的实现。也可能使用其他的数据结构，如集合（set），此时对超时事件的访问不受超时事件发生的顺序限制。

注意2： 对每个测试成分，假设在相应的成分类型定义中声明了处理定时器启动/停止和定时器超时事件的一个特殊的*运行的定时器列表*和*超时列表*。

当一个定时器到期时（从概念上讲的选择对象事件集合的快照处理之前），超时事件被放在声明该定时器的范围单元的超时列表中。定时器立即变为不活动。在任意时间，对于任意特定定时器，仅有一个登录条目会出现在声明定时器的范围单元的的超时列表中。

在测试成分被显式或隐式停止时，所有运行中的定时器被自动取消。

表 19：TTCN-3 定 时 器 操 作 一 览 表

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**
读取定时器经过的时间	**Read**
检查定时器是否运行	**Running**
超时时间	**Timeout**

24.1 启动定时器操作

启动定时器操作（**start**）用来指出一个定时器应该开始运行，定时器的值应该是非负浮点数（即大于等于0.0）。当一个定时器被启动了，它的名字就被添加到运行的定时器列表中（对给定的范围单元）。

例：

```
MyTimer1.start;          // 启动MyTimer1，带有一个默认的持续时间
MyTimer2.start(20E-3);  // 启动MyTimer2，持续时间为20ms

// 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:=1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i==4; i:=1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

如果没有给定默认的持续时间或想要重写定时器声明中描述的默认值，应该使用可选的定时器值参数。当重写定时器持续时间，新的值仅用于当前这个定时器实例，以后这个定时器的任意**start**操作，如果没有指定持续时间的话，将使用默认持续时间。

启动一个定时器值为0.0的定时器意味着这个定时器立刻就超时了。启动一个带有负数定时器值的定时器，如定时器的值是一个表达式的结果，或者不带有指定的定时器值，将会导致一个运行错误。

定时器时钟运行从浮点值零（0.0）到持续时间参数说明的最大值。

可以对一个正在运行的定时器应用**start**操作，在这种情况下，定时器被停止并重新启动。在超时列表中用于这个定时器的任意条目都将被从这个超时列表中移去。

24.2 停止定时器操作

停止（**stop**）操作用来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把它移去。一个被停止的定时器变成不活动的，它的经过时间被设为浮点值零（0.0）。

尽管没有任何影响，但停止一个不活动的定时器是一个有效的操作。超时列表中用于这个定时器任意条目将被从该超时列表中移去。

关键字**all**可以用来停止调用**stop**操作的范围单元中所有可见的定时器。

例：

```
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

读操作（**read**）用来取回指定的定时器启动后经过的时间，并把这个时间存储到指定的变量中去，这个变量的类型应该是浮点型（**float**）

例：

```
var float Myvar;
MyVar := MyTimer1.read; // 把MyTimer1启动后经过的时间赋给MyVar
```

对一个不活动的定时器应用**read**操作，即不在正在运行的定时器列表中的定时器，将返回值零。

24.4 运行定时器操作

运行定时器（**running**）操作用来检查一个定时器是否在给定范围单元的正在运行的定时器列表中（即该定时器已经被启动，且没有超时或被停止）。如果该定时器在列表中，返回真值（**true**），否则返回假（**false**）。

例：

```
if (MyTimer1.running) { … }
```

24.5 超时操作

超时操作（**timeout**）允许检查一个测试成分的范围单元内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的到时。

处理**timeout**操作时，如果指定定时器的名字，则根据TTCN-3的范围规则搜索这个成分或模块控制的超时列表。如果有与该定时器名匹配的超时事件，从超时列表中移去该事件，**timeout**操作成功。**timeout**操作不应用在一个布尔表达式中，但它能用来决定**alt**语句中的一个选择对象，或作为一个行为描述中的独立的语句。在后面的这种情况下，可以把**timeout**操作认为是只有一个选择对象的**alt**语句的简写，也就是说，它具有阻塞语义，并因此提供被等待定时器超时的能力。

注意: TTCN-3的**timeout**操作与TTCN-2的TIMEOUT操作有相同的语义。

例1：

```
MyTimer1.timeout;   // 检查先前启动的定时器MyTimer1的超时
```

如果超时列表为空的话，使用关键字**any**的**timeout**操作成功（不是明确指定的定时器）。

例2：

```
any timer.timeout; // 检查任意以前启动的定时器的超时
```

24.6 与定时器一起使用的any和all的总结

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：带有 Any 和 All 的定时器操作

操作	被允许		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 概要

判定操作允许使用**getverdict**和**setverdict**操作来设置和取回判定。这些操作应该仅用于测试例、可选步和函数中。

表 21：TTCN-3 测试判定操作一览表

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

活动的配置的每个测试成分应该维护自己的本地判定。这个本地判定是测试成分实例化时为每个测试成分创建的对象，它用于在每个测试成分中（即在MTC和每个PTC中）追踪单个的判定。

注意: 与TTCN-2不一样，不能把最终的判定分配给一个测试成分，因此分配一个判定从不中断行为执行所在的测试成分的执行。如果必要的话，应使用语句来显式完成。

25.1 测试例判定

另外，在每个测试成分（即在MTC和每个PTC中）终止执行时，有一个全局判定被更新，这个全局判定对于**getverdict**和**setverdict**操作来说是不可访问的。当测试例终止执行时，这个测试例返回判定值。如果这个返回的判定没有显式地保存在控制部分（如赋值给一个变量），那么就它就被丢掉了。

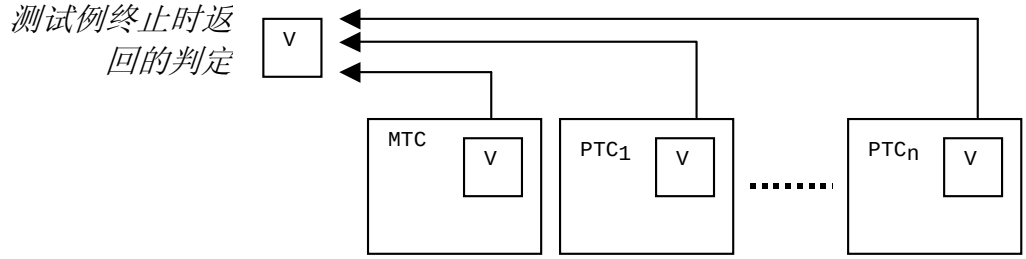


图 14：判 定 之 间 关 系 的 图 示

注意: TTCN-3不描述执行本地判定和测试例判定的实际机制。这些机制是依赖于实现的。

25.2 判定值和重写规则

25.2.0 概要

判定有五个不同的值：通过（**pass**）、失败（**fail**）、不确定的（**inconc**）、空（**none**）和错误（**error**），即判定类型**verdicttype**的不同值（见章节6.1）。

注意: **inconc**意味着一个不确定的判定。

使用**setverdict**操作应仅带有值**pass**、**fail**、**inconc**和**none**。

例1：

```
setverdict(pass);
setverdict(inconc);
```

可以通过使用**getverdict**操作来取回本地判定的值。

例2：

```
MyResult := getverdict; // MyResult是一个判定类型verdicttype的变量
```

当一个测试成分被实例化时，它的本地判定对象被创建且设置为空值（**none**）。

当改变本地判定值时（即使用**setverdict**操作），这个改变的结果将遵循表22中所列的重写规则（overwriting rules）。测试例判定在一个测试成分终止时隐式地更新，这个隐式操作的结果将遵循表22中所列的重写规则。

表 22：判 定 的 重写 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

例子 3:

```
 :
setverdict(pass);    // 本地判定设为pass
 :
setverdict(fail);    // 直到执行到这行，将导致本地判定值被改写为fail
 :                   // 当PTC终止测试例时，设判定为fail
```

25.2.1 错误判定

错误判定（**error**）是特别的，因为它是由测试系统设置的、用来指出发生了一个测试例错误（即运行时间错误）。它不能由**setverdict**操作来设置，也不会通过**getverdict**操作来返回。没有其它判定值可以重写一个错误（**error**）判定，这就意味着一个错误判定（**error**）只能是一个执行（**execute**）测试例操作的结果。

26 外部动作

在一些测试情况下，有些到SUT的电子接口可能被忽略，或从推理的角度讲是未知的（如管理接口），但是刺激SUT完成特定行为时可能是必要的（如发送一条消息到测试系统）。而且测试执行人员也可能要求特定的动作（如改变环境条件，如温度、供电电压等）。

要求的动作可以定义作一个串。

例1：

```
action("Send MyTemplate on lower PCO"); // SUT动作的非形式化描述
```

或作为描述SUT所发消息结构的模板的一个引用

例2：

```
action(MyTemplate); // 这等价于TTCN-2的隐式发送语句
```

在这两种情况中，仅有被要求的动作自身的非形式化的描述，而没有SUT所做的来触发这个动作的描述。

可以在测试例、函数、可选步和模板控制中描述SUT动作。

27 模块控制部分

27.0 概要

测试例定义在模块定义部分，而模块控制部分管理它们的执行。如果要在行为定义中使用定义在一个模块控制部分的所有的变量、定时器等（如果有的话），应该通过参数化来将它们传入测试例。也就是说，TTCN-3不支持任意种类的全程变量和定时器。

在每个测试例启动时，测试配置将被重新设置。这就意味着当测试例停止时，毁掉以前的测试例中由**create**、**connect**等操作管理的所有成分和端口（因此对新测试例来说是不“可见的”， 'visible'）

27.1 测试例的执行

使用执行（**execute**）语句调用一个测试例。作为测试例的执行结果，返回一个测试判定（**none**、**pass**、**inconclusive**、**fail**或**error**），且可以把这个判定赋值给一个变量来进行进一步的处理。

可选择地，**execute**语句允许通过定时器持续时间来监管测试例（见章节27.5）。

例：

```
execute(MyTestCase1());                      // 执行MyTestCase1，不存储返回的测试判定和时间监管

MyVerdict := execute(MyTestCase2());         // 执行MyTestCase2，把结果判定存储在变量MyVerdict中

MyVerdict := execute(MyTestCase3(),5E-3);    // 执行MyTestCase3，把结果判定存储在变量MyVerdict中
                                             // 如果测试例在5ms内没有终止，MyVerdict将得到值'error'
```

27.2 测试例的终止（Termination of test cases）

测试例和MTC一起终止。在MTC（显式或隐式地）终止时，所有运行的并行测试成分（PTC）都将被测试系统终止。

注意1： 停止所有PTC的具体机制是一个特定的工具，所以在本文讨论范围之外。

根据章节25定义的规则，基于不同测试成分的本地判定来计算测试例最终的判定。当测试成分自己终止时或被自身、另一个测试成分或测试系统停止时，该测试成分的实际本地判定变成它的最终的本地判定。

注意2： 为了避免由于被延时的PTC停止所引起的计算测试判定的紊乱情况，MTC应该确保所有的PTC在停止自己以前已经停止了（使用**done**语句）。

27.3 测试例的控制执行

那些表11和表12中定义的编程语句，可以用在模块控制部分中说明测试例执行顺序或测试例可能运行的次数等事项

例1：

```
module MyTestSuite () {
     :
    control {
         :
        // 做10次这个测试
        count:=0;
        while (count < 10)
        {   execute (MySimpleTestCase1());
            count := count+1;
        }
```

```
    }
}
```

如果没有使用编程语句，那么默认为按照它们在模块控制部分中的出现的先后顺序执行测试例。

注意: 这并不排除某些工具可能希望重写这个默认顺序而允许用户或工具去选择一个不同的执行顺序。

测试例返回一个**verdicttype**类型的值，因此，根据测试例的结果控制执行顺序是可能的。

例2：

```
if (execute (MySimpleTestCase()) == pass) { execute (MyGoOnTestCase) }
else { execute (MyErrorRecoveryTestCase) };
```

27.4 测试例选择

可以使用布尔表达式选择和去选择要执行的测试例。当然，这包括返回布尔类型（**boolean**）值的函数的使用。

注意： 这与TTCN-2中所谓的测试选择表达式是等价的。

例1：

```
module MyTestSuite () {
     :
    control {
         :
        if (MySelectionExpression1()) {
            execute(MySimpleTestCase1());
            execute(MySimpleTestCase2());
            execute(MySimpleTestCase3());
        }
        if (MySelectionExpression2()) {
            execute(MySimpleTestCase4());
            execute(MySimpleTestCase5());
            execute(MySimpleTestCase6());
        }
         :
    }
}
```

另外一种作为一个组来执行测试例的方法是把这些测试例集中在一个函数中，并从该模块的控制部分执行这个函数。

例2：

```
function MyTestCaseGroup1()
{   execute(MySimpleTestCase1());
    execute(MySimpleTestCase2());
    execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{   execute(MySimpleTestCase4());
    execute(MySimpleTestCase5());
    execute(MySimpleTestCase6());
}
 :
control
{   if (MySelectionExpression1()) { MyTestCaseGroup1(); }
    if (MySelectionExpression1()) { MyTestCaseGroup2(); }
     :
}
```

27.5 控制部分中定时器的使用

可以用定时器来监管测试例的执行，可以在**execute**语句中使用一个显式的超时来完成这种监管。如果测试例没有在定时器的持续时间内结束，那么测试例执行的结果应该是一个错误判定，同时测试系统终止该测试例。用于测试例监管的这个定时器是一个系统定时器，且不必声明或启动它。

例1：

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// 如果7ms内MyTestCase没有完成执行的话，返回的判定将会是错误（error）
```

也可以显式地使用定时器操作来控制测试例的执行。

例2：

```
// 使用运行的定时器操作的例子
while (T1.running or x<10)  // T1是一个以前启动了的定时器
{   execute(MyTestCase());
    x := x+1;
}

// 启动和超时操作使用的例子

timer T1 := 1;
 :
execute(MyTestCase1());
T1.start;
T1.timeout; // 在执行下一个测试例前暂停
execute(MyTestCase2());
```

28 描述属性

28.0 概要

可以使用**with**语句来关联属性和TTCN-3的语言要素。**with**语句的参数（即实际属性）被简单地定义作一个自由文本串。

有四中属性：

a) 显示（**display**）：允许与特定的表示格式有关的显示属性的说明；

b) 编码（**encode**）：允许引用特定的编码规则；

c) 变量（**variant**）： 允许引用特定的编码变量；

c) 扩展（**extension**）：允许用户定义的属性说明。

28.1 显示属性

所有的TTCN-3语言要素都可以有用来说明如何显示特别的语言要素的显示（**display**）属性，如在一个表格格式中显示特定的语言要素。

可以在ES 201 873-2 [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TR 101 873-3 [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。

其它的显示（**display**）属性可以由用户定义。

注意： 因为用户定义的属性是非标准化的，所以不同工具对这些属性的解释可能不同，甚至并不支持用户定义的属性。

28.2 值的编码

28.2.0 概要

编码规则定义一个特别的值、模板等如何被编码、如何在一个通信端口上被传输以及如何解码接收到的信号。TTCN-3没有默认的编码机制，这就意味着以TTCN-3的一些外部规则来定义编码规则或编码指令。

在TTCN-3中，可以使用编码（**encode**）属性和变量（**variant**）属性来说明通用的或特别的编码规则。

28.2.1 编码属性

编码（**encode**）属性允许用于TTCN-3定义中某些引用的编码规则或编码指令的关联。

与ASN.1编码属性有关的专用属性串见附录D。

定义实际编码规则的属性串（如prose, functions等）在本文讨论范围之外。如果没有引用特定的规则，那么编码应该是各实现自身的事。

在大多数情况下，用层次的方法使用编码属性。顶层是完整的模块，下面一层是一个组，最下层是一个单个的类型或定义：

a) 模块（**module**）：编码适用于模块中定义的所有类型，包括TTCN-3类型（嵌入类型）；

b) 组（**group**）：编码适用于一组用户定义的类型定义；

c) 类型或定义（**type or definition**）：编码适用于一个单个的用户定义的类型或定义；

d) 字段（**field**）：编码适用于记录类型（**record**）、集合类型（**set**）或模板（**template**）中的一个字段。

例：

```
module MyTTCNmodule
{   :
    import from MySecondModule {
        type MyRecord
    }
    with { encode "MyRule 1" } // 将根据MyRule 1编码MyRecord的实例。

    :
    type charstring MyType; // 通常根据全局规则编码
    :
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer     field1,     // 将根据Rule 3编码field1
            boolean     field2,     // 将根据Rule 3编码field2
            Mytype      field3      // 将根据Rule 3编码field3
        }
        with { encode (field1, field2) "Rule 3" }
        :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.2 变量属性

使用变量（**variant**）属性说明当前说明了的编码方案的一个改进，而不是它的替代。

例：

```
module MyTTCNmodule1
{   :
    type charstring MyType; // 通常根据全局规则编码
    :
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer     field1,     // 将根据Rule 2编码field1，使用编码变量 “length form 3”
            Mytype      field3      // 将根据Rule 2编码field3，使用任意可能的长度编码格式
        }
        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）变量（**variant**）属性（见章节E.2.1）：

a) 当用于整型数和枚举类型时，“8比特”（"8 bit"）和“无符号8比特”（"unsigned 8 bit"）意味着整型值或与枚举类型关联的整型数在系统中按8-比特（单字节）表示来处理。

b) 当用于整型数和枚举类型时，“16比特”（"16 bit"）和“无符号16比特”（"unsigned 16 bit"）意味着整型值或与枚举类型关联的整型数在系统中按16-比特（双字节）表示来处理。

c) 当用于整型数和枚举类型时，“32比特”（"32 bit"）和“无符号32比特”（"unsigned 32 bit"）意味着整型值或与枚举类型有关的整型数在系统中按32-比特（四字节）表示来处理。

d) 当用于整型数和枚举类型时，“64比特”（"64 bit"）和“无符号64比特”（"unsigned 64 bit"）意味着整型值或与枚举类型有关的整型数在系统中按64-比特（八字节）表示俩处理。

e) "IEEE754 float" 、 "IEEE754 double" 、 "IEEE754 extended float" 和 "IEEE754 extended double" 用于浮点类型时，意味着根据标准IEEE 754来编码和解码值（见附录F）。

下列串是为字符型（**char**）、通用字符型（ **universal char**）、字符串型（ **charstring**）和通用字符串型（**universal charstring**）（标准化）预定义的变量（**variant**）属性（见章节clause E.2.2）：

a) 当用于通用字符（**char**）或通用字符串（**universal char**）类型时，"UTF-8"意味着都应该根据ISO/IEC 10646 [6]中附录R定义的UCS转换格式（UTF-8）分别编码和解码值的每一个字符。

b) 当用于通用字符（**char**）或通用字符串（**universal char**）类型时，" UCS-2"意味着都应该根据UCS-2编码表示形式（见ISO/IEC 10646 [6]的章节14.1）分别编码和解码值的每一个字符。

c) 当用于通用字符（**char**）或通用字符串（**universal char**）类型时，"UTF-16"意味着都应该根据ISO/IEC 10646 [6]中附录Q定义的UCS转换格式（UTF-16）分别编码和解码值的每一个字符。

d) 当用于字符（**char**）、通用字符（ **universal char**）、字符串（ **charstring**）和通用字符串（**universal charstring**）类型时，"8 bit "意味着都应该根据ISO/IEC 8859 (一个8-比特编码）中描述的编码表示分别编码和解码值的每一个字符。

下列串是为结构化类型（标准化）预定义的变量（**variant**）属性（见章节clause E.2.3）：

a) 当用于一个记录类型时，"IDL:fixed FORMAL/01-12-01 v.2.6"意味着把值作为一个IDL定点十进制值（见附录F）。

这些变量属性可以被用于与更多的高层描述的编码属性的结合。例如，在一个自身具有全局编码属性"BER:1997"（见章节clause D.1.5.1）的模块中描述的带有"UTF-8"变量属性的一个通用字符串（**universal**

charstring），将导致串中值的每个字符先根据UTF-8规则被编码，然后根据更全局的BER规则编码这个UTF-8值。

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该模块外部的引用源中描述这些无效的编码规则。

28.3 扩展属性

所有TTCN-3语言要素都可以带有用户描述）扩展（**extension**）属性。

注意： 因为用户定义的属性没有被标准化，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语句可以把属性和一个单个的语言要素关联在一起，也可以通过诸如列表属性语句中结构类型字段的之类的方法把属性和多个的语言要素关联在一起，而这个属性语句与一个单个类型定义关联的或通过一个**with**语句与环境范围单元或语言要素组（**group**）关联。

例：

```
// 将把MyPDU1作为PDU显示
type record MyPDU1 { … } with { display "PDU"}

// 将把MyPDU1作为带有特定的应用扩展属性MyRule的PDU显示
type record MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}

// 下列组定义 …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with {display "PDU"}    // 将把组MyPDUs的所有类型作为PDU显示

// 与下面的定义相同
group MyPDUs {
    type record MyPDU3 { … } with { display "PDU"}
    type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 属性的重写规则

一个较低范围单元中的属性定义会重写（overwriting）较高范围的通用属性定义。

例1：

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// 在下面，MyRecordA根据规则RuleA编码，而不是RuleB
type record MyRecordB
{
     :
```

```
    field   MyRecordA
} with { encode "RuleB" }
```

放在另一个**with**语句的范围内的**with**语句将重写最外面的那个**with**语句，这同样适用于带有组的**with**语句使用的情况。当重写方案用在引用与单个定义相结合时要小心。通用的规则指的是应根据属性出现的顺序指定和重写这些属性。

```
// with语句重写方案使用的例子
group MyPDUs
{
    type record MyPDU1 { … }
    type record MyPDU2 { … }

    group MySpecialPDUs
    {
        type record MyPDU3 { … }
        type record MyPDU4 { … }
    }
    with {extension "MySpecialRule"}    // MyPDU3和MyPDU4将具有特定的应用扩展属性MySpecialRule
}
with
{
    display "PDU";      // 组MPDUs的所有类型都将作为PDU被显示，且
    extension "MyRule"; // （如果没有被重写）具有扩展属性MyRule
}

// 与下面相同…
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，可在较高范围中重写较低范围的属性定义 。

例2：

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// 在下面，根据RuleB编码MyRecordA
type record MyRecordB
{
    :
    fieldA  MyRecordA
} with { encode override "RuleB" }
```

override指令迫使所有较低范围内的类型具有指定的属性。

28.6 改变引入语言元素的属性

通常，一个语言元素与它的属性一起被引入。在一些情况下，当引入语言元素时，这些属性可能不得不改变，例如，可以在一个模块中把一个类型显示做ASP，而在引入它的另一个模块中应该被显示为PDU。对于这样的情况，允许在**import**语句中改变属性。

例：

```
import from MyModule {
    type MyType
}
with { display "ASP" } // MyType将被显示作ASP
```

```
import from MyModule {
    group MyGroup
}
with {
    display "PDU";                  // 通过默认，所有类型都将被显示为PDU
    extension "MyRule"
}
```

附录A (范式):
BNF和静态语义

A.1 TTCN-3 BNF

A.1.0 概要

本附录定义了TTCN-3使用扩展的BNF（自此以后仅称为BNF）的语法。

A.1.1 语法描述的转换

表A.1为TTCN-3定义了描述扩展BNF语法的元符号（metanotation）：

表A.1：语法的元符号

::=	被定义为
abc xyz	Abc后跟着xyz
|	选择对象（alternative）
[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
(...)	文本的分组（textual grouping）
Abc	非终结符（non-terminal symbol）abc
"abc"	终结符（terminal symbol）abc

A.1.2 语句终结符

通常，所有TTCN-3语言构件（如定义、声明、语句和操作）由分号（;）终止。如果语言构件使用右括号（}）结束或接一个右括号，即该语言构件式语句块、操作或声明中的最后一个语句的话，分号是可选的。

A.1.3 标识符

TTCN-3标识符是大小写敏感的，而且可能只能包含小写字母（a-z）、大写字母（A-Z）和数字（0-9），也允许使用下划线（_）。一个标识符应该以字母开头（即非数字和下划线）。

A.1.4 注释

注释可以写在TTCN－3说明中的任意位置。

注释块应该以符号对/*开始，以符号对*/结束。

例1：

```
/* 这是一个注释块
占两行 */
```

注释块不应嵌套。.

```
/* 这不是 /* 一个合法的 */ 注释 */
```

注释行应该以符号对//开始，以一个<*新行*>为结束。

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，但是不能嵌套在一个语句中。

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终结符和保留字列在表A.2和A.3中。

表A.2: TTCN-3特定终结符列表

块开始/结束符	**{ }**
列表开始/结束符	**( )**
可选对象符	**[ ]**
到达符号（在一个范围内）	**..**
行注释和块注释	**/* */ //**
行/语句终止符	**;**
算术操作符	**+ / -**
串连接操作符	**&**
等价操作符	**!= == >= <=**
串开始结束符	**" '**
通配符/匹配符	**? ***
赋值符	**:=**
通信操作赋值	**->**
比特串、十六进制串和八位组串值	**B H O**
浮点指数	**E**

应该把表10中定义的附录C描述的预定义函数标识符看作是保留字。

表 A.3：TTCN-3保留字列表

action	fail	noblock	self
activate	false	none	send
address	float		sender
all	for	not	set
alt	from	not4b	setverdict
altstep	function	nowait	signature
and		null	start
and4b	getverdict		stop
any	getcall	objid	subset
anytype	getreply	octetstring	superset
	goto	of	system
bitstring	group	omit	
boolean		on	template
	hexstring	optional	testcase
call		or	timeout
catch	if	or4b	timer
char	ifpresent	out	to
charstring	import	override	trigger
check	in		true
clear	inconc	param	type
complement	infinity	Modulepar	
component	inout	pass	union
connect	integer	pattern	universal
const	interleave	port	unmap
control		procedure	
create	label		value
	language	raise	valueof
deactivate	length	read	var
default	log	receive	variant
disconnect		record	verdicttype
display	map		
do	match	recursive	while
done	message	rem	with
	mixed	repeat	
else	mod	reply	xor
encode	modifies	return	xor4b
enumerated	module	running	
error	mtc	runs	
except			
exception			
execute			
extension			
external			

表A.3中列出的TTCN-3终止符不应用作TTCN-3模块中的标识符，且应全部小写。

A.1.6 TTCN-3句法BNF形式

A.1.6.0 TTCN模块

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
 BeginChar
 [ModuleDefinitionsPart]
 [ModuleControlPart]
 EndChar
 [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4. ModuleIdentifier ::= Identifier
5. DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{" DefinitiveObjIdComponentList "}"
6. DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7. DefinitiveObjIdComponent ::=      NameForm |
 DefinitiveNumberForm |
 DefinitiveNameAndNumberForm
8. DefinitiveNumberForm ::= Number
9. DefinitiveNameAndNumberForm ::= Identifier "(" DefinitiveNumberForm ")"
```

A.1.6.1 模块定义部分

A.1.6.1.0 概要

```
10. ModuleDefinitionsPart ::= ModuleDefinitionsList
11. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
12. ModuleDefinition ::=    (TypeDef |
 ConstDef |
 TemplateDef |
 ModuleParDef |
 FunctionDef |
 SignatureDef |
 TestcaseDef |
 AltstepDef |
 ImportDef |
 GroupDef |
 ExtFunctionDef |
 ExtConstDef) [WithStatement]
```

A.1.6.1.1 Typedef定义

```
13. TypeDef ::= TypeDefKeyword TypeDefBody
14. TypeDefBody ::= StructuredTypeDef | SubTypeDef
15. TypeDefKeyword ::= "type"
16. StructuredTypeDef ::= RecordDef |
 UnionDef |
 SetDef |
 RecordOfDef |
 SetOfDef |
                           EnumDef |
 PortDef |
 ComponentDef
17. RecordDef ::= RecordKeyword StructDefBody
18. RecordKeyword ::= "record"
19. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
 BeginChar
 [StructFieldDef {"," StructFieldDef}]
EndChar
20. StructTypeIdentifier ::= Identifier
21. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
22. StructDefFormalPar ::=  FormalValuePar
/* 静态语义——FormalValuePar应该确定一个参数 */
23. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec] [OptionalKeyword]
24. StructFieldIdentifier ::= Identifier
25. OptionalKeyword ::= "optional"
26. UnionDef ::= UnionKeyword UnionDefBody
27. UnionKeyword ::= "union"
28. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
 BeginChar
 UnionFieldDef {"," UnionFieldDef}
 EndChar
29. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
30. SetDef ::= SetKeyword StructDefBody
31. SetKeyword ::= "set"
32. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
33. OfKeyword ::= "of"
34. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
35. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
36. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
 BeginChar
 EnumerationList
 EndChar
37. EnumKeyword ::= "enumerated"
38. EnumTypeIdentifier ::= Identifier
39. EnumerationList ::= Enumeration {"," Enumeration}
40. Enumeration ::= EnumerationIdentifier ["(" Number ")"]
41. EnumerationIdentifier ::= Identifier
42. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword) [ArrayDef] [SubTypeSpec]
43. SubTypeIdentifier ::= Identifier
44. SubTypeSpec ::= AllowedValues | StringLength
/* 静态语义——AllowedValues应该与字段的子类型相同类型 */
45. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
46. ValueOrRange ::= RangeDef | SingleConstExpression
```

```
/* 静态语义——RangeDef产生式应该仅与基于integer、char、universal char、charstring、universal charstring或
float类型*/
/* 静态语义——当细分地定义字符串或通用字符串类型范围时，不应该在相同的SubTypeSpec中混合它们的指定义*/
47. RangeDef ::= LowerBound ".." UpperBound
48. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/* 静态语义——StringLength应该仅与串类型一起使用或限制set of和record of类型。SingleConstExpressionhe
UpperBound应该取非负整数值（假设UpperBound包含无限大） */
49. LengthKeyword ::= "length"
50. PortType ::=    [GlobalModuleId Dot] PortTypeIdentifier
51. PortDef ::= PortKeyword PortDefBody
52. PortDefBody ::= PortTypeIdentifier PortDefAttribs
53. PortKeyword ::= "port"
54. PortTypeIdentifier ::= Identifier
55. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
56. MessageAttribs ::= MessageKeyword
 BeginChar
 {MessageList [SemiColon]}+
 EndChar
57. MessageList ::= Direction AllOrTypeList
58. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
59. MessageKeyword ::= "message"
60. AllOrTypeList ::= AllKeyword | TypeList
61. AllKeyword ::= "all"
62. TypeList ::= Type {"," Type}
63. ProcedureAttribs ::= ProcedureKeyword
 BeginChar
 {ProcedureList [SemiColon]}+
 EndChar
64. ProcedureKeyword ::= "procedure"
65. ProcedureList ::= Direction AllOrSignatureList
66. AllOrSignatureList ::= AllKeyword | SignatureList
67. SignatureList ::= Signature {"," Signature}
68. MixedAttribs ::= MixedKeyword
 BeginChar
 {MixedList [SemiColon]}+
 EndChar
69. MixedKeyword ::= "mixed"
70. MixedList ::= Direction ProcOrTypeList
71. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
72. ProcOrType ::= Signature | Type
73. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
 BeginChar
 [ComponentDefList]
 EndChar
74. ComponentKeyword ::= "component"
75. ComponentType ::=   [GlobalModuleId Dot] ComponentTypeIdentifier
76. ComponentTypeIdentifier ::= Identifier
77. ComponentDefList ::= {ComponentElementDef [SemiColon]}
78. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
79. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
80. PortElement ::= PortIdentifier [ArrayDef]
81. PortIdentifier ::= Identifier
```

A.1.6.1.2 常量定义

```
82. ConstDef ::= ConstKeyword Type ConstList
83. ConstList ::= SingleConstDef {"," SingleConstDef}
84. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* 静态语义——ConstantExpression的值应该与该常量已经声明的类型相同*/
85. ConstKeyword ::= "const"
86. ConstIdentifier ::= Identifier
```

A.1.6.1.3 模板定义

```
87. TemplateDef ::=     TemplateKeyword BaseTemplate [DerivedDef] AssignmentChar TemplateBody
88. BaseTemplate ::= (Type | Signature) TemplateIdentifier ["(" TemplateFormalParList ")"]
89. TemplateKeyword ::= "template"
90. TemplateIdentifier ::= Identifier
91. DerivedDef ::= ModifiesKeyword TemplateRef
92. ModifiesKeyword ::= "modifies"
93. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
94. TemplateFormalPar ::=   FormalValuePar | FormalTemplatePar
/* 静态语义——FormalValuePar应该确定一个参数 */
95. TemplateBody ::= SimpleSpec | FieldSpecList | ArrayValueOrAttrib
```

```
/* 静态语义—在TeplateBody中，ArrayValueOrAttrib可被用于array、record、record of和set of类型中 */
96. SimpleSpec ::= SingleValueOrAttrib
97. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
98. FieldSpec ::= FieldReference AssignmentChar TemplateBody
99. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
100. StructFieldRef ::= StructFieldIdentifier
101. ParRef ::= SignatureParIdentifier
/* 操作语义 – SignatureParIdentifier应该是来自相关的特征定义的一个形式化的参数标识符 */
102. SignatureParIdentifier ::= ValueParIdentifier
103. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* 静态语义—ArrayRef可以被选择地应用于array类型和ASN.1的SET OF和SEQUENCE OF类型以及TTCN的record of和set of
类型。在一个ASN.1或TTCN比特串类型内，可使用相同的表示方法。 The same notation can be used for a Bit
reference inside an itstring type */
104. FieldOrBitNumber ::= SingleExpression
/* 静态语义—SingleExpression决定一个整形值*/
105. SingleValueOrAttrib ::=    MatchingSymbol [ExtraMatchingAttributes] |
 SingleExpression [ExtraMatchingAttributes] |
 TemplateRefWithParList
/* 静态语义—VariableIdentifier（通过singleExpression访问）仅可在嵌入式模板定义中用来引用当前范围内的变量 */
106. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
107. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
108. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
109. NotUsedSymbol ::= Dash
110. MatchingSymbol ::=     Complement |
 AnyValue |
 AnyOrOmit |
 ValueOrAttribList |
 Range |
 BitStringMatch |
 HexStringMatch |
 OctetStringMatch |
 CharStringMatch |
 SubsetMatch |
 SupersetMatch
111. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch IfPresentMatch
112. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
113. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
114. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
115. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
116. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
117. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
118. CharStringMatch ::= PatternKeyword Cstring
119. PatternKeyword ::= "pattern"
120. Complement ::= ComplementKeyword ValueList
121. ComplementKeyword ::= "complement"
122. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
123. SubsetMatch ::= SubsetKeyword ValueList
/* 静态语义 – Subset匹配应该仅与set of类型一起使用 */
124. SubsetKeyword ::= "subset"
125. SupersetMatch ::= SupersetKeyword ValueList
/* 静态语义 - Superset匹配应该仅与set of类型一起使用 */
126. SupersetKeyword ::= "superset"
127. AnyValue ::= "?"
128. AnyOrOmit ::= "*"
129. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
130. LengthMatch ::= StringLength
131. IfPresentMatch ::= IfPresentKeyword
132. IfPresentKeyword ::= "ifpresent"
133. Range ::= "(" LowerBound ".." UpperBound ")"
134. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
135. UpperBound ::= SingleConstExpression | InfinityKeyword
/* 静态语义—LowerBound和UpperBound应该仅取integer、char、universal char或float类型值，可能仅出现
SingleConstExpression */
136. InfinityKeyword ::= "infinity"
137. TemplateInstance ::= InLineTemplate
138. TemplateRefWithParList ::=     [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList]
| TemplateParIdentifier
139. TemplateRef ::=    [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
140. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar] TemplateBody
/* 静态语义—当类型隐含无二义性时，字段可能只是被省略 */
141. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
142. TemplateActualPar ::=              TemplateInstance
/* 静态语义—当相应的形参不是模板类型时，TemplateInstance产生式应该确定一个或多个SingleExpressions */
143. TemplateOps ::= MatchOp | ValueofOp
144. 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，且模板的每个字段应该解析为一个单值*/
```

```
145. MatchKeyword ::= "match"
146. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
147. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 函数定义

```
148. FunctionDef ::= FunctionKeyword FunctionIdentifier
 "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
StatementBlock
149. FunctionKeyword ::= "function"
150. FunctionIdentifier ::= Identifier
151. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
152. FunctionFormalPar ::= FormalValuePar |
 FormalTimerPar |
 FormalTemplatePar |
 FormalPortPar
153. ReturnType ::= ReturnKeyword Type
154. ReturnKeyword ::= "return"
155. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
156. RunsKeyword ::= "runs"
157. OnKeyword ::= "on"
158. MTCKeyword ::= "mtc"
159. StatementBlock ::=     BeginChar [FunctionStatementOrDefList] EndChar
160. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
161. FunctionStatementOrDef ::= FunctionLocalDef |
 FunctionLocalInst |
 FunctionStatement
162. FunctionLocalInst ::= VarInstance | TimerInstance
163. FunctionLocalDef ::= ConstDef
164. FunctionStatement ::= ConfigurationStatements |
 TimerStatements |
 CommunicationStatements |
 BasicStatements |
 BehaviourStatements |
 VerdictStatements |
 SUTStatements
165. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
166. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
PreDefFunctionIdentifier
167. PreDefFunctionIdentifier ::= Identifier
/* 静态语义 – 标识符是本文附录C中预定义的TTCN－3函数标识符之一 */
168. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
169. FunctionActualPar ::=  TimerRef |
 TemplateInstance |
 Port |
 ComponentRef
/* 静态语义——当相应的形参不是模板类型，TemplateInstance产生式应该解析为一个或多个单表达式，即与原表达式产生式等价
*/
```

A.1.6.1.5 过程特征定义

```
170. SignatureDef ::= SignatureKeyword SignatureIdentifier
 "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
 [ExceptionSpec]
171. SignatureKeyword ::= "signature"
172. SignatureIdentifier ::= Identifier
173. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
174. SignatureFormalPar ::=     FormalValuePar
175. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
176. ExceptionKeyword ::= "exception"
177. ExceptionTypeList ::= Type {"," Type}
178. NoBlockKeyword ::= "noblock"
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 测试例定义

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
 "("[TestcaseFormalParList] ")" ConfigSpec
StatementBlock
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
 FormalTemplatePar
```

```
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
187. SystemKeyword ::= "system"
188. TestcaseInstance ::=   ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")" ["," 
TimerValue] ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
 TemplateInstance
/* 静态语义——When the corresponding formal parameter is not of template type the TemplateInstance
production shall resolve to one or more SingleExpressions i.e., equivalent to the Expression
production */
```

A.1.6.1.7 可选步定义

```
193. AltstepDef ::= AltstepKeyword AltstepIdentifier
 "("[AltstepFormalParList] ")" [RunsOnSpec]
 BeginChar
 AltstepLocalDefList
 AltGuardList
 EndChar
194. AltstepKeyword ::= "altstep"
195. AltstepIdentifier ::= Identifier
196. AltstepFormalParList ::= FunctionFormalParList
/* 静态语义—— 所有的形式参数必须是制参数，即在参数中 */
197. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
198. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef
199. AltstepInstance ::=    AltstepRef "(" [FunctionActualParList] ")"
200. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 引入定义

```
201. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | (BeginChar ImportSpec EndChar))
202. ImportKeyword ::= "import"
203. AllWithExcepts ::= AllKeyword [ExceptsDef]
204. ExceptsDef ::= ExceptKeyword BeginChar ExceptSpec EndChar
205. ExceptKeyword ::= "except"
206. ExceptSpec ::= {ExceptElement [SemiColon]}
/* 静态语义－任意产生式成分（ExceptGroupSpec, ExceptTypeDefSpec等）可以仅在ExceptSpec产生式中出现一次 */
207. ExceptElement ::=  ExceptGroupSpec |
 ExceptTypeDefSpec |
 ExceptTemplateSpec |
 ExceptConstSpec |
 ExceptTestcaseSpec |
 ExceptAltstepSpec |
 ExceptFunctionSpec |
 ExceptSignatureSpec |
 ExceptModuleParSpec
208. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
209. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
210. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
211. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
212. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
213. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
214. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
215. ExceptSignatureSpec ::= SignatureKeyword (SignatureRefList | AllKeyword)
216. ExceptModuleParSpec ::= ModuleParKeyword (ModuleParRefList | AllKeyword)
217. ImportSpec ::=     {ImportElement [SemiColon]}
218. ImportElement ::=  ImportGroupSpec |
 ImportTypeDefSpec |
 ImportTemplateSpec |
 ImportConstSpec |
 ImportTestcaseSpec |
 ImportAltstepSpec |
 ImportFunctionSpec |
 ImportSignatureSpec |
 ImportModuleParSpec
219. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
220. ModuleId ::=   GlobalModuleId [LanguageSpec]
/* 静态语义——如果引用模块包含TTCN-3标识符号，则LanguageSpec可以直接被省略 */
221. LanguageKeyword ::= "language"
222. LanguageSpec ::= LanguageKeyword FreeText
223. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
224. RecursiveKeyword ::= "recursive"
```

```
225. ImportGroupSpec ::= GroupKeyword (GroupRefList | AllGroupsWithExcept)
226. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
227. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
228. FullGroupIdentifier := GroupIdentifier {Dot GroupIdentifier} [ExceptsDef]
229. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
230. ExceptFullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
231. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
232. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
233. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
234. TypeDefIdentifier ::=  StructTypeIdentifier |
 EnumTypeIdentifier |
 PortTypeIdentifier |
 ComponentTypeIdentifier |
 SubTypeIdentifier
235. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
236. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
237. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
238. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
239. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
240. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
241. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
242. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
243. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
244. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
245. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
246. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
247. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
248. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
249. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
250. ImportSignatureSpec ::= SignatureKeyword (SignatureRefList | AllSignaturesWithExcept)
251. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
252. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
253. ImportModuleParSpec ::= ModuleParKeyword (ModuleParRefList | AllModuleParWithExcept)
254. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
255. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 组定义

```
256. GroupDef ::= GroupKeyword GroupIdentifier
 BeginChar
 [ModuleDefinitionsPart]
 EndChar
257. GroupKeyword ::= "group"
258. GroupIdentifier ::= Identifier
```

A.1.6.1.10 外部函数定义

```
259. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier "("[FunctionFormalParList]
")" [ReturnType]
260. ExtKeyword ::= "external"
261. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 外部常量定义

```
262. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
263. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 模块参数定义

```
264. ModuleParDef ::= ModuleParKeyword "{" ModuleParList "}"
265. ModuleParKeyword ::= "modulepar"
266. ModuleParList ::= ModulePar {SemiColon ModulePar}
267. ModulePar ::= ModuleParType ModuleParIdentifier [AssignmentChar ConstantExpression] {","
ModuleParIdentifier [AssignmentChar ConstantExpression]}
/* 静态语义——ConstantExpression的值应该与参数声明的类型相同的类型 */
268. ModuleParType ::= Type
269. ModuleParIdentifier ::= Identifier
```

A.1.6.2 控制部分

A.1.6.2.0 概要

```
270. ModuleControlPart ::=  ControlKeyword
 BeginChar
 ModuleControlBody
 EndChar
 [WithStatement] [SemiColon]
271. ControlKeyword ::= "control"
272. ModuleControlBody ::= [ControlStatementOrDefList]
273. ControlStatementOrDefList ::=  {ControlStatementOrDef [SemiColon]}+
274. ControlStatementOrDef ::=  FunctionLocalInst |
 ControlStatement |
 FunctionLocalDef
275. ControlStatement ::=   TimerStatements |
 BasicStatements |
 BehaviourStatements |
 SUTStatements
```

A.1.6.2.1 变量实例化

```
276. VarInstance ::= VarKeyword Type VarList
277. VarList ::= SingleVarInstance {"," SingleVarInstance}
278. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
279. VarInitialValue ::= Expression
280. VarKeyword ::= "var"
281. VarIdentifier ::= Identifier
282. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 定时器实例化

```
283. TimerInstance ::= TimerKeyword TimerList
284. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
285. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
286. TimerKeyword ::= "timer"
287. TimerIdentifier ::= Identifier
288. TimerValue ::= Expression
/* 静态语义——当表达式解析到SingleExpression时，它必须解析到一个浮点类型值。当将默认定时器值赋给定时器数组的初始化过
程中，表达式应该解析到CompoundExpression */
289. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
 TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.2.3 测试成分操作

```
290. ConfigurationStatements ::=    ConnectStatement |
 MapStatement |
 DisconnectStatement |
 UnmapStatement |
 DoneStatement |
 StartTCStatement |
 StopTCStatement
291. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
292. CreateOp ::= ComponentType Dot CreateKeyword
293. SystemOp ::= SystemKeyword
294. SelfOp ::= "self"
295. MTCOp ::= MTCKeyword
296. DoneStatement ::= ComponentId Dot DoneKeyword
297. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword) ComponentKeyword
298. DoneKeyword ::= "done"
299. RunningOp ::= ComponentId Dot RunningKeyword
300. RunningKeyword ::= "running"
301. CreateKeyword ::= "create"
302. ConnectStatement ::= ConnectKeyword PortSpec
303. ConnectKeyword ::= "connect"
304. PortSpec ::= "(" PortRef "," PortRef ")"
305. PortRef ::= ComponentRef Colon Port
306. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
307. DisconnectStatement ::= DisconnectKeyword PortSpec
308. DisconnectKeyword ::= "disconnect"
309. MapStatement ::= MapKeyword PortSpec
310. MapKeyword ::= "map"
```

```
311. UnmapStatement ::= UnmapKeyword PortSpec
312. UnmapKeyword ::= "unmap"
313. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "(" FunctionInstance ")"
/* 静态语义－函数实例可能只有参数 */

/* 静态语义－函数实例不应该有定时器参数*/
314. StartKeyword ::= "start"
315. StopTCStatement ::= StopKeyword | ComponentIdentifier Dot StopKeyword |
 AllKeyword ComponentKeyword Dot StopKeyword
316. ComponentIdentifier ::= VariableRef | FunctionInstance
/* 静态语义－与VariableRef关联的变量或者与FunctionInstance关联的返回类型必须是成分类型 */
```

A.1.6.2.4 端口操作

```
317. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
318. CommunicationStatements ::=    SendStatement |
 CallStatement |
 ReplyStatement |
 RaiseStatement |
 ReceiveStatement |
 TriggerStatement |
 GetCallStatement |
 GetReplyStatement |
 CatchStatement |
 CheckStatement |
 ClearStatement |
 StartStatement |
 StopStatement
319. SendStatement ::= Port Dot PortSendOp
320. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
321. SendOpKeyword ::= "send"
322. SendParameter ::= TemplateInstance
323. ToClause ::= ToKeyword AddressRef
324. ToKeyword ::= "to"
325. AddressRef ::= VariableRef | FunctionInstance
/* 静态语义—VariableRef和FunctionInstance的返回值必须是地址或成分类型 */
326. CallStatement ::= Port Dot PortCallOp [PortCallBody]
327. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
328. CallOpKeyword ::= "call"
329. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* 静态语义－只有输出参可以被省略或被说明时带有匹配属性 */
330. CallTimerValue ::= TimerValue | NowaitKeyword
/* 静态语义－值必须是浮点类型 */
331. NowaitKeyword ::= "nowait"
332. PortCallBody ::=   BeginChar
 CallBodyStatementList
 EndChar
333. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
334. CallBodyStatement ::= CallBodyGuard StatementBlock
335. CallBodyGuard ::= AltGuardChar CallBodyOps
336. CallBodyOps ::= GetReplyStatement | CatchStatement
337. ReplyStatement ::= Port Dot PortReplyOp
338. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
339. ReplyKeyword ::= "reply"
340. ReplyValue ::= ValueKeyword Expression
341. RaiseStatement ::= Port Dot PortRaiseOp
342. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
343. RaiseKeyword ::= "raise"
344. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
345. PortOrAny ::= Port | AnyKeyword PortKeyword
346. PortReceiveOp ::=  ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－如果ReceiveParameter选项出现的话， 可以仅PortRedirect选项也出现 */
347. ReceiveOpKeyword ::= "receive"
348. ReceiveParameter ::= TemplateInstance
349. FromClause ::= FromKeyword AddressRef
350. FromKeyword ::= "from"
351. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
352. PortRedirectSymbol ::= "->"
353. ValueSpec ::= ValueKeyword VariableRef
354. ValueKeyword ::= "value"
355. SenderSpec ::= SenderKeyword VariableRef
/* 静态语义－变量引用必须是地址或成分类型 */
356. SenderKeyword ::= "sender"
357. TriggerStatement ::= PortOrAny Dot PortTriggerOp
358. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－如果ReceiveParameter选项出现的话， 可以仅PortRedirect选项也出现*/
```

```
359. TriggerOpKeyword ::= "trigger"
360. GetCallStatement ::= PortOrAny Dot PortGetCallOp
361. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
[PortRedirectWithParam]
/* 静态语义－如果ReceiveParameter选项出现的话， 可以仅PortRedirectWithParam选项也出现 */
362. GetCallOpKeyword ::= "getcall"
363. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
364. RedirectSpec ::= ValueSpec [ParaSpec] [SenderSpec] |
 ParaSpec [SenderSpec] |
 SenderSpec
365. ParaSpec ::= ParaKeyword ParaAssignmentList
366. ParaKeyword ::= "param"
367. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
368. AssignmentList ::= VariableAssignment {"," VariableAssignment}
369. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* 静态语义－parameterIdentifiers必须来自于相应的特征定义 */
370. ParameterIdentifier ::= ValueParIdentifier |
 TimerParIdentifier |
 TemplateParIdentifier |
 PortParIdentifier
371. VariableList ::= VariableEntry {"," VariableEntry}
372. VariableEntry ::= VariableRef | NotUsedSymbol
373. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
374. PortGetReplyOp ::=     GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
 [FromClause] [PortRedirectWithParam]
/* 静态语义－如果选项ReceiveParameter出现的话，可以仅PortRedirectWithParam选项也出现 */
375. GetReplyOpKeyword ::= "getreply"
376. ValueMatchSpec ::= ValueKeyword TemplateInstance
377. CheckStatement ::= PortOrAny Dot PortCheckOp
378. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
379. CheckOpKeyword ::= "check"
380. CheckParameter ::=     CheckPortOpsPresent | FromClausePresent | RedirectPresent
381. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
382. RedirectPresent ::= PortRedirectSymbol SenderSpec
383. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
384. CatchStatement ::= PortOrAny Dot PortCatchOp
385. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* 静态语义－如果选项CatchOpParameter出现的话，可以仅PortRedirect选项也出现 */
386. CatchOpKeyword ::= "catch"
387. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
388. ClearStatement ::= PortOrAll Dot PortClearOp
389. PortOrAll ::= Port | AllKeyword PortKeyword
390. PortClearOp ::= ClearOpKeyword
391. ClearOpKeyword ::= "clear"
392. StartStatement ::= PortOrAll Dot PortStartOp
393. PortStartOp ::= StartKeyword
394. StopStatement ::= PortOrAll Dot PortStopOp
395. PortStopOp ::= StopKeyword
396. StopKeyword ::= "stop"
397. AnyKeyword ::= "any"
```

A.1.6.2.5 定时器操作

```
398. TimerStatements ::=    StartTimerStatement | StopTimerStatement | TimeoutStatement
399. TimerOps ::= ReadTimerOp | RunningTimerOp
400. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
401. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
402. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
403. ReadTimerOp ::= TimerRef Dot ReadKeyword
404. ReadKeyword ::= "read"
405. RunningTimerOp ::= TimerRefOrAny Dot RunningKeyword
406. TimeoutStatement ::= TimerRefOrAny Dot TimeoutKeyword
407. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
408. TimeoutKeyword ::= "timeout"
```

A.1.6.3 类型

```
409. Type ::= PredefinedType | ReferencedType
410. PredefinedType ::= BitStringKeyword |
 BooleanKeyword |
 CharStringKeyword |
 UniversalCharString |
 CharKeyword |
 UniversalChar |
 IntegerKeyword |
```

```
 OctetStringKeyword |
 ObjectIdentifierKeyword |
 HexStringKeyword |
 VerdictTypeKeyword |
 FloatKeyword |
 AddressKeyword |
 DefaultKeyword |
 AnyTypeKeyword
411. BitStringKeyword ::= "bitstring"
412. BooleanKeyword ::= "boolean"
413. IntegerKeyword ::= "integer"
414. OctetStringKeyword ::= "octetstring"
415. ObjectIdentifierKeyword ::= "objid"
416. HexStringKeyword ::= "hexstring"
417. VerdictTypeKeyword ::= "verdicttype"
418. FloatKeyword ::= "float"
419. AddressKeyword ::= "address"
420. DefaultKeyword ::= "default"
421. AnyTypeKeyword ::= "anytype"
422. CharStringKeyword ::= "charstring"
423. UniversalCharString ::= UniversalKeyword CharStringKeyword
424. UniversalKeyword ::= "universal"
425. CharKeyword ::= "char"
426. UniversalChar ::= UniversalKeyword CharKeyword
427. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
428. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
 EnumTypeIdentifier |
 SubTypeIdentifier |
 ComponentTypeIdentifier
429. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
430. TypeActualPar ::= ConstantExpression
431. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
432. ArrayBounds ::= SingleConstExpression
/* 静态语义——ArrayBounds will resolve to a non negative value of integer type */
```

A.1.6.4 值

```
433. Value ::= PredefinedValue | ReferencedValue
434. PredefinedValue ::= BitStringValue |
 BooleanValue |
 CharStringValue |
 IntegerValue |
 OctetStringValue |
 ObjectIdentifierValue |
 HexStringValue |
 VerdictTypeValue |
 EnumeratedValue |
 FloatValue |
 AddressValue |
 OmitValue
435. BitStringValue ::= Bstring
436. BooleanValue ::= "true" | "false"
437. IntegerValue ::= Number
438. OctetStringValue ::= Ostring
439. ObjectIdentifierValue ::= ObjectIdentifierKeyword "{" ObjIdComponentList "}"
/* ReferencedValue必须是类型对象标识符 */
440. ObjIdComponentList ::= {ObjIdComponent}+
441. ObjIdComponent ::= NameForm |
 NumberForm |
 NameAndNumberForm
442. NumberForm ::= Number | ReferencedValue
/* 静态语义——referencedValue必须是整形且具有非负值 */
443. NameAndNumberForm ::= Identifier "(" NumberForm ")"
444. NameForm ::= Identifier
445. HexStringValue ::= Hstring
446. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
447. EnumeratedValue ::= EnumerationIdentifier
448. CharStringValue ::= Cstring | Quadruple
449. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
450. Group ::= Number
451. Plane ::= Number
452. Row ::= Number
453. Cell ::= Number
454. FloatValue ::= FloatDotNotation | FloatENotation
455. FloatDotNotation ::= Number Dot DecimalNumber
456. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
```

```
457. Exponential ::= "E"
458. ReferencedValue ::= ValueReference [ExtendedFieldReference]
459. ValueReference ::=     [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier) |
 ValueParIdentifier |
 ModuleParIdentifier |
 VarIdentifier
460. Number ::= (NonZeroNum {Num}) | "0"
461. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
462. DecimalNumber ::= {Num}+
463. Num ::= "0" | NonZeroNum
464. Bstring ::= "'" {Bin} "'" "B"
465. Bin ::= "0" | "1"
466. Hstring ::= "'" {Hex} "'" "H"
467. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
468. Ostring ::= "'" {Oct} "'" "O"
469. Oct ::= Hex Hex
470. Cstring ::= """ {Char} """
471. Char ::= /* 参考 – 由相关的CharacterString类型定义的一个字符。对于字符串来说，字符来自于ISO/IEC 10646中定义的该字符集合。对于通用字符串类型，字符来自于ISO/IEC 10646中定义的任意字符集。*/
472. Identifier ::= Alpha{AlphaNum | Underscore}
473. Alpha ::= UpperAlpha | LowerAlpha
474. AlphaNum ::= Alpha | Num
475. 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"
476. 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"
477. ExtendedAlphaNum ::= /* 参考 – 图形字符来自ISO/IEC 10646基本拉丁文的图形字符或来自拉丁－1补充字符集（从字符(0,0,0,33)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)
*/
478. FreeText ::= """ {ExtendedAlphaNum} """
479. AddressValue ::= "null"
480. OmitValue ::= OmitKeyword
481. OmitKeyword ::= "omit"
```

A.1.6.5 参数化

```
482. InParKeyword ::= "in"
483. OutParKeyword ::= "out"
484. InOutParKeyword ::= "inout"
485. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
486. ValueParIdentifier ::= Identifier
487. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
488. PortParIdentifier ::= Identifier
489. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
490. TimerParIdentifier ::= Identifier
491. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type TemplateParIdentifier
492. TemplateParIdentifier ::= Identifier
```

A.1.6.6 With语句

```
493. WithStatement ::= WithKeyword WithAttribList
494. WithKeyword ::= "with"
495. WithAttribList ::= "{" MultiWithAttrib "}"
496. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
497. SingleWithAttrib ::= AttribKeyword [OverrideKeyword] [AttribQualifier] AttribSpec
498. AttribKeyword ::=  EncodeKeyword |
 VariationKeyword |
 DisplayKeyword |
 ExtensionKeyword
499. EncodeKeyword ::= "encode"
500. VariationKeyword ::= "variant"
501. DisplayKeyword ::= "display"
502. ExtensionKeyword ::= "extension"
503. OverrideKeyword ::= "override"
504. AttribQualifier ::= "(" DefOrFieldRefList ")"
505. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
506. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef | PredefinedType
/* 静态语义－DefOrFieldRef必须移用模块、组或与with语句相关联的定义中的定义或字段 */
507. DefinitionRef ::=  StructTypeIdentifier |
 EnumTypeIdentifier |
 PortTypeIdentifier |
 ComponentTypeIdentifier |
 SubTypeIdentifier |
 ConstIdentifier |
```

```
 TemplateIdentifier |
 AltstepIdentifier |
 TestcaseIdentifier |
 FunctionIdentifier |
 SignatureIdentifier |
             VarIdentifier |
             TimerIdentifier |
                PortIdentifier |
                ModuleParIdentifier |
 FullGroupIdentifier
508. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword BeginChar GroupRefList EndChar]) |
 ( TypeDefKeyword AllKeyword [ExceptKeyword BeginChar TypeRefList] EndChar)
|
 ( TemplateKeyword AllKeyword [ExceptKeyword BeginChar TemplateRefList] EndChar)
|
 ( ConstKeyword AllKeyword [ExceptKeyword BeginChar ConstRefList] EndChar)
|
 ( AltstepKeyword AllKeyword [ExceptKeyword BeginChar AltstepRefList] EndChar)
|
 ( TestcaseKeyword AllKeyword [ExceptKeyword BeginChar TestcaseRefList] EndChar) |
 ( FunctionKeyword AllKeyword [ExceptKeyword BeginChar FunctionRefList] EndChar) |
 ( SignatureKeyword AllKeyword [ExceptKeyword BeginChar SignatureRefList] EndChar) |
 ( ModuleParKeyword AllKeyword [ExceptKeyword BeginChar ModuleParRefList] EndChar)

509. AttribSpec ::= FreeText
```

A.1.6.7 行为语句

```
510. BehaviourStatements ::=    TestcaseInstance |
 FunctionInstance |
 ReturnStatement |
 AltConstruct |
 InterleavedConstruct |
 LabelStatement |
 GotoStatement |
 RepeatStatement |
             DeactivateStatement |
 AltstepInstance
/* 静态语义-不应该从一个现存的正在执行的测试例或测试例调用的函数链中调用TestcaseInstance，即只能从控制部分或控制部分
调用的函数中初始化测试例 */
511. VerdictStatements ::= SetLocalVerdict
512. VerdictOps ::= GetLocalVerdict
513. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* 静态语义－SingleExpression必须解析到一个判定类型值 */

/* 静态语义——SetLocalVerdict不应该用来赋error值 */
514. SetVerdictKeyword ::= "setverdict"
515. GetLocalVerdict ::= "getverdict"
516. SUTStatements ::= ActionKeyword "(" (FreeText | TemplateRefWithParList) ")"
517. ActionKeyword ::= "action"
518. ReturnStatement ::= ReturnKeyword [Expression]
519. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
520. AltKeyword ::= "alt"
521. AltGuardList ::= {GuardStatement [SemiColon]}+ [ElseStatement [SemiColon]]
522. GuardStatement ::= AltGuardChar (AltstepInstance | GuardOp StatementBlock)
523. ElseStatement ::= "["ElseKeyword "]" StatementBlock
524. AltGuardChar ::= "[" [BooleanExpression] "]"
525. GuardOp ::=    TimeoutStatement |
 ReceiveStatement |
 TriggerStatement |
 GetCallStatement |
 CatchStatement |
 CheckStatement |
 GetReplyStatement |
 DoneStatement
/* 静态语义——GuardOp用在模块控制部分。可以制含有timeoutStatement */
526. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList EndChar
527. InterleavedKeyword ::= "interleave"
528. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
529. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
530. InterleavedGuard ::= "[" "]" GuardOp
531. InterleavedAction ::= StatementBlock
/* 静态语义——StatementBlock可以不包含loop语句、跳转（goto）、激活（activate）、去激活（deactivate）、停止
（stop）、返回（return）或调用函数 */
532. LabelStatement ::= LabelKeyword LabelIdentifier
533. LabelKeyword ::= "label"
```

```
534. LabelIdentifier ::= Identifier
535. GotoStatement ::= GotoKeyword LabelIdentifier
536. GotoKeyword ::= "goto"
537. RepeatStatement ::= "repeat"
538. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
539. ActivateKeyword ::= "activate"
540. DeactivateStatement ::= DeactivateKeyword ["(" Expression ")"]
/* 静态语义－表达式应该取默认类型的值 */
541. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 基本语句

```
542. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct
543. Expression ::= SingleExpression | CompoundExpression
/* 静态语义—Expression不应该包含模块控制部分中的配置、激活操作或判定操作 */
544. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* 静态语义—在CompoundExpression中，ArrayExpression可以用于Arrays、record、record of和set of类型 */
545. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
546. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
547. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
548. ArrayElementExpressionList ::= NotUsedOrExpression {"," NotUsedOrExpression}
549. NotUsedOrExpression ::= Expression | NotUsedSymbol
550. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
551. SingleConstExpression ::= SingleExpression
/* 静态语义—SingleConstExpression不应该包含变量或模块参数，而应该在编译的时候解析到一个常量值 */
552. BooleanExpression ::= SingleExpression
/* 静态语义—BooleanExpression应该解析到一个布尔类型值 shall resolve to a Value of type Boolean */
553. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* 静态语义－在CompoundConstExpression中，ArrayConstExpression可以用于Arrays、record、record of和set of类
型 */
554. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
555. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
556. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
557. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
558. Assignment ::= VariableRef AssignmentChar Expression
/* 操作语义 – 赋值的RHS上的表达式应该取LHS类型的一个明确的值 */
559. SingleExpression ::= SimpleExpression {LogicalOp SimpleExpression}
/* 操作语义 – 如果SimpleExpressions和LogicalOp都存在，那么SimpleExpressions应该取可兼容类型的特定值 */
560. SimpleExpression ::= ["not"] SubExpression
/* 操作语义 – not操作符的操作数应该是布尔类型类型（TTCN或ASN.1）或是布尔类型的派生类型 */
561. SubExpression ::= PartialExpression [RelOp PartialExpression]
/* 操作语义 – 如果PartialExpressions和RelOp都存在，那么PartialExpressions应该取可兼容类型的特定的值 */
/* 操作语义 – 如果RelOp是"<" | ">" | ">=" | "<=" 那么每个SubExpression应该取一个特定的整型、枚举或浮点类型值
（这些值既可以是TTCN值也可以是ASN.1值 values) */
562. PartialExpression ::= Result [ShiftOp Result]
/* 操作语义 – 每个结果应该解析到一个特定的值。如果存在多个结果，右手边的操作数应该是整型，且如果shift op是'<<'或
'>>'，那么右手的操作数应该解析为一个bitstring、hexstring或octetstring类型。如果shift op是'<@'或'@>'，那么左手
边的操作数应该是bitstring、hexstring、charstring或universal charstring类型 */
563. Result ::= SubResult {BitOp SubResult}
/* 操作语义 - If both SubResults and the BitOp exist then the SubResults shall evaluate to specific
values of compatible types */
564. SubResult ::= ["not4b"] Product
/* 操作语义 – 如果not4b操作符存在，那么操作数应该是bitstring/octetstring或hexstring类型 */
565. Product ::= Term {AddOp Term}
/* 操作语义 – 每个Term应该解析到一个特定的值。如果存在多个Term而且AddOp解析到StringOp，那么terms应该解析到 shall
bitstring、hexstring、octetstring、 charstring或universal charstring类型相同的类型。如果多个Term存在且
AddOp不解析到StringOp，那么Terms应该解析到整型或浮点类型。 shall both resolve to type integer or float. */
566. Term ::= Factor {MultiplyOp Factor}
/* 操作语义 －每个Factor应该解析到一个特定的值。如果多个Factor存在，那么Factors应该解析到整型或浮点类型。 */
567. Factor ::= [UnaryOp] Primary
/* 操作语义 – Primary应该解析到一个特定的值。如果UnaryOp存在且不是"not"，那么Primary应该解析到布尔类型。如果
UnaryOp是"+"或"-"，那么Primary应该解析到整型或浮点类型。如果UnaryOp是not4b，那么Primary应该解析到bitstring、
hexstring或octetstring类型 */
568. Primary ::= OpCall | Value | "(" SingleExpression ")"
569. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | ArrayOrBitRef | TypeDefIdentifier))
| ArrayOrBitRef }+
/* 操作语义: 仅当使用ExtendedFieldReference的VarInstance或ReferencedValue类型是anytype类型，应该使用
TypeDefIdentifier */
570. OpCall ::=     ConfigurationOps |
 VerdictOps |
 TimerOps |
 TestcaseInstance |
 FunctionInstance |
```

```
 TemplateOps |
 ActivateOp
571. AddOp ::= "+" | "-" | StringOp
/* 操作语义 - "+"或 "-"操作符的操作数应该是整型或浮点型（即TTCN或ASN.1预定义类型）或整型或浮点型的派生类型（即子类型）) or derivations of integer or float (i.e., subrange) */
572. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* 操作语义 - "*"、"/"、rem或mod操作符的操作符应该是整型或浮点型（即TTCN或ASN.1预定义类型）或整型或浮点型的派生类型（即子类型）) or derivations of integer or float (i.e., subrange) */
573. UnaryOp ::= "+" | "-"
/* 操作语义 - "+"或"-"操作符的操作数应该是整型或浮点型（即TTCN或ASN.1预定义类型）或整型或浮点型的派生类型（即子类型）) or derivations of integer or float (i.e., subrange) */
574. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* 操作语义 – 操作符的优先权在表7中定义 */
575. BitOp ::= "and4b" | "xor4b" | "or4b"
/* 操作语义 - and4b、or4b或xor4b操作符的操作数应该是bitstring、hexstring或octetstring类型（TTCN或ASN.1）或这些类型的派生类型 */
576. LogicalOp ::= "and" | "xor" | "or"
/* 操作语义 – and或xor应该是布尔类型（TTCN或ASN.1）或是布尔类型的派生类型 */
/* 操作语义 - 操作符的优先权在表7中定义*/
577. StringOp ::= "&"
/* 操作语义 – 串操作符的操作数应该是bitstring, hexstring、octetstring或character string类型 */
578. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
579. LogStatement ::= LogKeyword "(" [FreeText] ")"
580. LogKeyword ::= "log"
581. LoopConstruct ::= ForStatement |
 WhileStatement |
 DoWhileStatement
582. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
 StatementBlock
583. ForKeyword ::= "for"
584. Initial ::= VarInstance | Assignment
585. Final ::= BooleanExpression
586. Step ::= Assignment
587. WhileStatement ::=     WhileKeyword "(" BooleanExpression ")"
 StatementBlock
588. WhileKeyword ::= "while"
589. DoWhileStatement ::= DoKeyword StatementBlock
 WhileKeyword "(" BooleanExpression ")"
590. DoKeyword ::= "do"
591. ConditionalConstruct ::=   IfKeyword "(" BooleanExpression ")"
 StatementBlock
 {ElseIfClause}[ElseClause]
592. IfKeyword ::= "if"
593. ElseIfClause ::= ElseKeyword IfKeyword "(" BooleanExpression ")" StatementBlock
594. ElseKeyword ::= "else"
595. ElseClause ::= ElseKeyword StatementBlock
```

A.1.6.9 其它产生式

```
596. Dot ::= "."
597. Dash ::= "-"
598. Minus ::= Dash
599. SemiColon ::= ";"
600. Colon ::= ":"
601. Underscore ::= "_"
602. BeginChar ::= "{"
603. EndChar ::= "}"
604. AssignmentChar ::= ":="
```

附录 B （范式）：
匹配引入值

B.1 模板匹配机制

B.1.0 概要

本附录描述了可以用在TTCN-3模板中的匹配机制（且仅用于模板中）。

B.1.1 匹配特殊值

特定的值是基本的TTCN-3模板匹配机制。模板中的特定值是不包含任意匹配机制或通配符的表达式。除非描述了另外的匹配机制，否则当且仅当引入字段值具有与模板中表达式取值相同的精确值，模板字段匹配相应的引入字段值。

例：

```
// 给定消息类型定义
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}       // 整型数组的特定值
}
```

B.1.1.1 省略值

关键字**omit**表示一个可选的模板字段应该忽略，倘若模板字段是可选的，那它可以用于所有类型的值。

例：

```
template Mymessage MyTemplate:=
{   :
    :
    field3 := omit,     // omit this field
    :
}
```

B.1.2 匹配机制代替值

B.1.2.0 概要

下面的匹配机制可以用来代替精确的值。

B.1.2.1 值列表

值列表描述接收引入值的列表，它可以用于所有类型的值。当且仅当引入字段值匹配值列表中的任意值时，使用一个值列表的模板的字段匹配相应的引入字段。在值列表中的每个字段应该与使用这个匹配机制的模板字段中声明的类型一样的类型。

例：

```
template Mymessage MyTemplate:=
{
    field1 := (2,4,6),                  // 整型值列表
    field2 := ("String1", "String2"),   // 字符串值列表
    :
    :
}
```

B.1.2.2 值列表的补集

关键字**complement**表示不作为引入值接收的值列表（即它时值列表的补集）。它可以用于所有类型的左右值。

列表中的每个值应该使用补集的模板字段的声明的类型。当且仅当引入字段不匹配值列表中列出的任意值时，使用补集的模板字段匹配相应的引入字段。当然，值列表可以时一个单个的值。

例：

```
template Mymessage MyTemplate:=
{
    complement (1,3,5),      // 不可接收的整型值列表
    :
    field3 not(true)        // 将匹配false
    :
}
```

B.1.2.3 任意值

匹配符"?" (*AnyValue*)用来指出可以接收的任意值，它可以用在所有类型的任意值上。当且仅当引入字段取特定类型的一个单个值时，使用任意值机制的模板字段匹配相应的引入字段。

例：

```
template Mymessage MyTemplate:=
{
    field1 := ?,    // 将匹配任意的整型r
    field2 := ?,    // 将匹配任意的非空字符串值
    field3 := ?,    // 将匹配true或false
    field4 := ?     // 将匹配整型的任意序列。
}
```

B.1.2.4 任意值或空

用匹配符号“*”（*任意值或空*）来指明任意可接收有效的引入值，包括那些值的省略。如果声明模板字段为可选的，那么它可以用于所有类型的值。

当且仅当引入字段取值为指定类型的任意元素或引入字段被省略掉了，那么使用这个符号的模板字段匹配相应的引入字段。

例：

```
template Mymessage MyTemplate:=
{   :
    field3 := *,    // 将匹配true、flase或省略字段
    :
```

```
}
```

B.1.2.5 值域

当使用整型或浮点型（整型或浮点型的子类型）时，值域（Ranges）指明可接收值的有界范围。一个有界的值应该是：

a) 无穷大或负无穷大；

b) 取值为特定的整型或浮点型值的表达式。

下界应该放在值域的左边，上界放在右边，且下界应该小于上界。当且仅当引入字段的值等于值域内的一个值的时候，使用值域的模板字段匹配相应的引入字段。

当边界用在**char**、**universal char**、**charstring**或**universal charstring**类型模板或模板字段中的时候，边界应该根据类型的编码字符集表取有效的字符位置（例如给定的位置不能是空）。上下界之间的空位置被认为不是特定值域的有效值。

例：

```
template Mymessage MyTemplate:=
{
    field1 := (1 .. 6), // 整型的值域
    :
    :
    :
}
// field1的其他条目可以是(-infinity to 8)或(12 to infinity)
```

B.1.2.6 超集

超集（SuperSet）是仅用于匹配**set of**类型值的一个操作，超集用关键字**superset**标识。当且仅当引入字段包含定义在超集中的所有元素至少一次（可以多次）时，超集匹配相应的引入字段。超集参数的类型应该是使用超集机制的字段声明的类型。

例：

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// 匹配包含任意顺序和位置的1、2和3至少一次的任意整数序列
```

B.1.2.7 子集

子集（SubSet）是仅用于匹配**set of**类型值的一个操作，子集用关键字**subset**标识。

当且仅当引入字段包含定义在子集中的所有元素至多一次时，子集匹配相应的引入字段。子集参数的类型应该是使用子集机制的字段声明的类型。

例：

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// 匹配包含任意顺序和位置的1、2和3至多一次的任意整数序列
```

B.1.3 值内部的匹配机制

B.1.3.0 概要

下面的匹配机制可以用在string、record、record of、set、set of和array类型的明确值中。

B.1.3.1 任意元素

用匹配符号“?”（*任意元素*）来指明该符号可以代替一个串（除了字符串以外）、一个**record of**、一个**set of**或一个数组。它应该仅用于串类型、**record of**类型、**set of**类型或数组类型值中。

例：

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10???'B,     // 每个“?”可以是0或1
    field4 := {1, ?, 3}     // ?可以是任意整型值
}
```

注意： “?”在field4中可以被解释为任意整型的*任意值*，或是**record of**、**set of**或数组的*任意元素*，因此上面的所有解释都将导致相同的匹配而不会引起问题。

B.1.3.1.1 使用单个字符通配符

如果在字符串中使用通配符“?”它应该使用字符模式（见附录B.1.5）。例如，"abcdxyz"、"abccxyz"和"abcxxyz"都与模式"abc?xyz"匹配。然而，"abcxyz"和"abcdefxyz"并不与模式"abc?xyz"匹配。

B.1.3.2 任意数目的元素或无元素

匹配符“*”（*任意元素或空*）用来表示代替任意数目的连续的string（不包括字符串）、record、record of、set、set of和array类型元素。“*”符号根据其前后的模式来匹配尽可能长的元素序列。

例：

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10*11'B,     //  “*”可以是任意比特序列（可以为空）
    field4 := {*, 2, 3}     //  “*”可以是任意数目的整数，或被省略
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

如果“*”出现在string、record、record of、set、set of和array的最高层，那么它应该被解释为*任意元素或空*（*AnyElementsOrNone*）。

注意： 这个规则防止了另一种可能的解释，即代替string、record、record of、set、set of和array中一个元素的*任意值或空*（*AnyValueOrNone*）。

B.1.3.2.1 使用多字符通配符

如果在字符串中使用通配符“*”那么应该使用字符模式（见附录B.1.5）。例如，"abcxyz"、"abcdefxyz"和"abcabcxyz"等与模式"abc*xyz"匹配。

B.1.4 匹配值的属性

B.1.4.0 概要

下面的属性可以与匹配机制相关。

B.1.4.1 长度限定

长短限定属性用来限制串值的长短以及**set of**、**record of**和数组结构的元素个数。它应该仅用做下面机制的一个属性：*补集*（Complement）、*任意值*（*AnyValue*）、*任意值或空*（*AnyValueOrNone*）、*任意元素*（*AnyElement*）和*任意元素或空*（*AnyElementsOrNone*）。它也可以与**ifpresent**属性结合使用。**Length**的语法见章节6.2.3和6.3.3。

在串值的情况下，根据本文主体部分中的表4定义解释长度的单位。对于**set of**、**record of**类型和数组类型，其长度单位是重复类型。界限应该由解析为特定的非负整型值的表达式指定。换句话说，可以使用关键字**infinity**来表示没有长度上限。

模板的长度说明不应该与相应类型的长度限制（如果有的话）矛盾。使用长度作为符号的一个属性的模板字段才与相应的引入字段相匹配当且仅当引入字段匹配该符号以及相关的属性的时候。如果引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，则长度属性匹配。仅有一个长度值的时候，只有接收到的字段的长度精确等于指定值的时候长度属性才匹配。

对于被省略了的字段，总是认为长度属性匹配（即带有**omit**是冗余的）。如果带有*AnyValueOrNone*和**ifpresent**，则给引入字段加了一个限制。

例：

```
template Mymessage MyTemplate:=
{
    field1 := complement (4,5) length (1 .. 6), // 与(1,2,3,6) 相同
    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的话，**ifpresent**表示可以做一个匹配（即不省略）。如声明该类型是可选的话，这个属性可以与所有匹配机制一起使用。

当且仅当引入字段与使用**ifpresent**的模板字段匹配或引入字段为空的时候，该模板字段匹配相应的引入字段。

例：

```
template Mymessage:MyTemplate:=
{   :
    field2 := "abcd" ifpresent, // 如果没有被省略，则匹配"abcd"
    :
    :
}
```

注意： *AnyValueOrNone* 与**? Ifpresent**的含义一样。

B.1.5 匹配字符模式

B.1.5.0 概要

字符模式可以用在模板中来定义需要接收的字符串的格式。字符模式可以用来匹配**charstring**和**universal charstring**类型值。除了文字字符外，字符模式语序元字符“?”和“*”的使用，用它们分别表示任意字符或任意数目的字符。

例1：

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

这个模板可以匹配这样的字符串，由'ab'开始，之后接任意两个字符，之后再接字符'xyz'，之后接任意数目的字符。

如果要逐字地解释任意元字符（metacharacter），应该在其前使用元字符'\'。

例2：

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

这个模板可以匹配这样的字符串，由'ab'开始，之后接任意字符，之后再接字符'?xyz'，之后接任意数目的字符。

用于TTCN-3模式的元字符列表列表见表B.1。

表B.1: TTCN-3 模 式 元 字 符 列 表

元 字 符	描 述
?	匹配任意字符
*	匹配任意字符零或多次
\	导致下面的元字符解释为一个文字（见下面注释）
[ ]	匹配指定结合中的任意字符，详见附录B.1.5.1
-	仅在方括号“[”和“]”可用，并允许描述一定范围的字符，相见B.1.5.1
^	仅在方括号“[”和“]”可用，导致匹配这个字符后面的字符集的补集中任意字符，详见B.1.5.1
\q{ group, plane, row, cell }	匹配四元组描述的通用字符
{reference}	插入定义串的引用用户，并把它解释为一个正则表达式，详见B.1.5.2
\d	匹配任意数字位（等价于[0-9]）
\w	匹配任意文字数字的字符（等价于[0-9a-zA-Z]）
\t	匹配C0控制字符HT（见ISO/IEC 6429 [13]）
\n	匹配C0控制字符LF（见ISO/IEC 6429 [13]）
\r	匹配C0控制字符CR（见ISO/IEC 6429 [13]）
\"	匹配双引号
|	用来表示两个可选的表达式
( )	用来给一个表达式建组
#(n, m)	匹配前面的表达式至少n次，但是不能多于m次，详见B.1.5.3
注意： 因此，反斜号字符可以用一对其间不带空格的反斜号来匹配。	

B.1.5.1 设置表达式

设置表达式（set expression）使用'[" "]'符号分隔。除了字符文字，可以使用分隔符“-”来描述字符范围。也可以通过在方括号后的第一个字符前放置“^”来对设置表达式进行取反。

例：

```
template charstring RegExp1:= pattern '[a-z]';  // 将匹配从a到z的任意字符

template charstring RegExp2:= pattern '[^a-z]'; // 将匹配除a到z外的任意字符

template charstring RegExp3:= pattern '[A-E][0-9][0-9][0-9]YKE';

// RegExp3将匹配以A到E之间字符开始后接三个数字位以及字母YKE的字符串
```

B.1.5.2 引用表达式

除了直接的串值，也可以在模式语句内使用对存在的模板、常量或变量的引用。引用用'{" "}'字符括起来，这个引用应该解析到字符串类型之一。

例：

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern '{MyString}';
```

这个模板会匹配包含字符'ab'且其后紧接任意字符的任意字符串。实际上，显式或通过引用来接在关键字**pattern**后的任意字符串将按照本节中定义的规则解释。

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

这个模板会匹配这样的任意字符串，以字符'ab'开始，后面接任意字符，再后面接字符'de'，最后接ISO10646-1中定义的字符（1，1，65，7），即字符的group=1，plane=1，row=65，cell=7。

B.1.5.3 匹配表达式多次

使用'#(n, m)'来说明应该匹配前面的表达式多次，着表示必须至少匹配前面的表达式n次，但是不能多于m次。

例：

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)';    // 至少匹配字符从a到z9次，但是不能多于11次

template charstring RegExp5:= pattern '[a-z]#(9)';        // 精确地匹配字符从a到z9次
                                                          //
template charstring RegExp6:= pattern '[a-z]#(9, )';      // 至少匹配字符从a到z9次
                                                          //
template charstring RegExp7:= pattern '[a-z]#(, 11)';     // 匹配字符从a到z不多于11次
                                                          //
```

附录C （范式）：
TTCN-3预定义函数

本附录定义了TTCN-3预定义函数。

C.1 Integer到character

```
int2char(integer value) return char
```

这个函数把一个0…127（8比特编码）之间的整数转换为一个ISO/IEC 646 [5]中的字符值。这个整型值描述这个字符的8比特编码。

如果这个参数值是负数或大于127的整数，则函数返回–1。

C.2 Character到integer

```
char2int(char value) return integer
```

这个函数把ISO/IEC 646 [5]中的一个字符值转换为一个0 … 127之间的整型值，这个整数值描述了这个字符的8比特编码。

C.3 Integer到universal character

```
int2unichar(integer value) return universal char
```

这个函数把0 … 2 147 483 647（32比特编码）范围内的一个整型值转换为一个ISO/IEC 10646 [6]中的字符值，这个整型值描述了这个字符的32比特编码。

如果参数值是一个负数或大于2 147 483 647，则函数返回四元组(255, 255, 255, 255)。

C.4 Universal character到integer

```
unichar2int(universal char value) return integer
```

这个函数把ISO/IEC 10646 [6]中的一个通用字符值转换为0 … 2 147 483 647范围中的一个整型值。这个整型值描述该字符的32比特编码。

C.5 Bitstring到integer

```
bit2int(bitstring value) return integer
```

这个函数把一个**bitstring**值转换为一个**integer**值。

出于转换的目的，一个比特串应该被解释做基数为2的正整数值。最右边的位为最低位，最左边的位为最高位，比特的0和1分别表示十进制的0和1。

C.6 Hexstring到integer

```
hex2int(hexstring value) return integer
```

这个函数把一个**hexstring**值转换位一个单一的整型值。

出于转换的目的，一个十六进制串应该被解释位以16位基数的正整数值。最右边的十六进制位为最低位，最左边的十六进制位为最高位，十六进制数字0 .. F分别表示十进制值0 .. 15。

C.7 Octetstring到integer

```
oct2int(octetstring value) return integer
```

这个函数把一个**octetstring**值转换为一个整型值。

出于转换的目的，一个八位组串应该被解释位以16位基数的正整数值。最右边的十六进制位为最低位，最左边的十六进制位为最高位。因为一个八位组由两个16进制数字组成，所以给出的十六进制数字的数目应该乘2。十六进制数字0 .. F分别表示十进制值0 .. 15。

C.8 Charstring到integer

```
str2int(charstring value) return integer
```

这个函数把一个表示一个整数值的字符串转换为一个等价的整数值。如果这个串表示的不是一个有效的整型值，函数返回值零（0）。

EXAMPLES:

```
str2int("66")   // 将返回整数66
str2int("-66")  // 将返回整数-66
str2int("abc")  // 将返回整数值0
str2int("0")    // 将返回整数值0
```

C.9 Integer到bitstring

```
int2bit(integer value, length) return bitstring
```

这个函数把一个整型值转换为一个单个的比特串值。这个结果串是长度为length的比特串。

出于转换的目的，一个比特串应该被解释位以2位基数的正整数值。最右边的十六进制位为最低位，最左边的十六进制位为最高位，比特的0和1分别表示十进制的0和1。如果转换产生的值的长度小于长度参数所描述的长度，则再该比特产的左边添加零。如果该值是负数或着结果比特串的长度比长度参数中描述的长度长，则产生测试例错误。

C.10 Integer到hexstring

```
int2hex(integer value, length) return hexstring
```

这个函数把一个整型值转换为一个单个的十六进制串值。这个结果串是长度为length的十六进制串。

出于转换的目的，一个十六进制串应该被解释位以16位基数的正整数值。最右边的十六进制位为最低位，最左边的十六进制位为最高位，十六进制数0 … F分别表示十进制的0到15。如果转换产生的值的长度小于长度参数所描述的长度，则再该比特产的左边添加零。如果该值是负数或着结果十六进制串的长度比长度参数中描述的长度长，则产生测试例错误。

C.11 Integer到octetstring

int2oct(integer value, length) **return octetstring**

这个函数把一个整型值转换为一个单个的八位组串值。这个结果串是长度为length的八位组串。

出于转换的目的，一个八位组串应该被解释位以16位基数的正整数值。最右边的十六进制位为最低位，最左边的十六进制位为最高位。因为一个八位组是由两个十六进制数字组成，所以给出的十六进制数字的数目应该乘2。十六进制数0 … F分别表示十进制的0到15。如果转换产生的值的长度小于长度参数所描述的长度，则再该比特产的左边添加零。如果该值是负数或着结果十六进制串的长度比长度参数中描述的长度长，则产生测试例错误。

C.12 Integer到charstring

int2str(integer value) **return charstring**

这个函数把一个整型值转换为与它等价的串（这个返回串的基数总是十进制的）。

EXAMPLES:

```
int2str(66)     // 将返回字符串"66"
int2str(-66)    // 将返回字符串"-66"
int2str(0)      // 将返回整数值"0"
```

C.13 Length到string type

lengthof(any_string_type value) **return integer**

这个函数返回**bitstring**、**hexstring**、**octetstring**或任意字符串类型值的长度，每个串类型的长度单位在本文主体部分的表4中定义。

应该通过计数每个合成字符以及其上合成音节字符（包括填充符）来计算**universal charstring**的长度（见ISO/IEC 10646 [**错误！未找到引用源。**]，章节23和24）。

例：

```
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 结构化类型中的元素数

```
sizeof(structured_type value) return integer
```

这个函数返回**record**、**record of**、**set**、**set of**类型声明的元素数目或是这些类型或数组的常量、变量、模板的元素数目。这个函数不应该用于不带有长度子分类的**record of**或**set of**类型。在**record of**和**set of**类型值或模板或数组的情况下，要返回的实际值是最后定义的元素的后续数目（该元素的索引值加1）。

注意： 只计算作为函数参数的TTCN-3对象的元素，也就是说，嵌入的类型/值在决定返回值的时候不予以考虑。

例：

```
// 给定
type record MyPDU
    {   boolean field1 optional,
        integer field2
    };
type record of integer MyPDU1;

template MyPDU MyTemplate
    { field1 omit,
     field2 5
    };

var integer numElements;

// 那么
numElements := sizeof(MyPDU);       // 返回2
numElements := sizeof(MyTemplate);  // 返回1
numElements := sizeof(MyPDU1);      // 因为没有约束MyPDU1，故返回错误


// 给定
type record length(0..10) of integer MyRecord;
var MyRecord MyRecordVar;
MyRecordVar := { 0, 1, omit, 2, omit };


// 那么
numElements := sizeof(MyRecordVar);
// 返回4，不考虑元素MyRecordVar[2]没有被定义
```

C.15 IsPresent函数

```
ispresent(any_type value) return boolean
```

当且仅当引用字段的值在被引用数据对象的实际实例中出现，这个函数返回**true**。**ispresent**的参数应该是对被定义为可选的（**optional**）数据对象中一个字段的引用。

```
// 给定
type record MyRecord
    {   boolean field1 optional,
        integer field2
    }
// 并给定MyPDU是一个MyRecord类型模板，且received_PDU也是MyRecord类型的模板，那么
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// 如果MyPDU的实际实例的field1出现，则返回true
```

C.16 IsChosen函数

```
ischosen(any_type value) return boolean
```

当且仅当数据对象描述了为给定数据对象实际选中的**union**类型的变量，该函数范围值**true**。

例：

```
// 给定
type union MyUnion
    {   PDU_type1   p1,
        PDU_type2   p2,
        PDU_type    p3
    }

// 且给定MyPDU是一个MyUnion类型模板，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，则返回true
```

C.17 Regexp函数

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
character_string_type
```

这个函数返回输入字符串instr中第n次匹配expression的子串内容，输入串instr可以是任意字符串类型。返回的字符串类型是instr的源类型。Expression是附录B.1.5中描述的字符模式。要返回的组的数目由groupno描述，且应该是一个正整数。组数目根据它们在组的方括号中出现的顺序进行赋值，从0开始，步长为1。如果在输入串中没有子串满足所有的条件（即模式和组数目）返回空串。

例：

```
// 给定
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)'

// 那么表达式
var charstring mystring := regexp(myinput, mypattern,1)
// 将返回'17'。
```

C.18 Bitstring到charstring

```
bit2str (bitstring value) return charstring
```

这个函数把一个单个的比特串值转换为一个字符串值。结果的字符串具有与比特串相同的长度，并仅包含字符'0'和'1'。

出于转换的目的，应该把一个比特串转换为一个字符串。比特串的每一位根据其值为0或1，转换为一个字符'0'或'1'。结果字符串中的字符的连续顺序与比特串中比特位的顺序相同。

例：

```
bit2str ('1110101'B) will return "1110101"
```

C.19 Hexstring到charstring

hex2str (hexstring value) **return charstring**

这个函数把一个单个的十六进制串值转换为一个字符串值。结果的字符串具有与十六进制串相同的长度，并仅包含字符'0'到'9'和'A'到'F'。

出于转换的目的，应该把一个十六进制串转换为一个字符串。十六进制数字的每一位根据其值为0到9或A到Z，转换为一个字符'0'到'9'或'A'到'F'。结果字符串中的字符的连续顺序与十六进制串中比特位的顺序相同。

例：

hex2str ('AB801'H) will return "AB801"

C.20 Octetstring到character string

oct2str (octetstring value) **return charstring**

这个函数把一个八位组串转换位一个字符串。结果的字符串应该具有与输入的八位组串相同的长度。八位组应该作为ISO/IEC 646 [5]编码被解释（根据IRV），并把结果字符保存在返回值中。大于7F的八位组值将引起错误。

例：

oct2str ('4469707379'H) = "Dipsy"

注意： 返回的字符串可以包含步在双引号中表示的非图形的字符。

C.21 Character string到octetstring

str2oct (**charstring** value) **return octetstring**

这个函数把一个字符串值转换为一个八位组串。结果八位组串与输入的字符串值等长。八位组的每一位应包含该字符串的相应字符的ISO/IEC 646 [5]编码（根据IRV）

例：

str2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'H

C.22 Bitstring到hexstring

bit2hex (bitstring value) **return hexstring**

这个函数把一个单个的比特串值转换位一个单个的十六进制串值。结果的十六进制串值表示与该比特串相同的值。

出于转换的目的，一个比特串应该被转换为一个十六进制串，此时，比特串应该从最右边的位开始分成4位的组。每个组4比特，并按照下面去转换位一个十六进制数字。

'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.

当最左边的组的比特位不够4位的时候，用'0'B从左边开始填充知道它正好包含4比特数字，然后转换它。结果的十六进制串中，十六进制数字的连续顺序与比特串中4位分组的顺序相同。

例：

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 Hexstring到octetstring

```
hex2oct (hexstring value) return octetstring
```

这个函数把一个单个的十六进制串值转换位一个单个的八位组制串值。结果的八位组串表示与该十六进制串相同的值。

出于转换的目的，一个十六进制串应该被转换为一个八位组串，此时，当十六进制串的长度是2的倍数的话，这个八位组串包含与原十六进制串相同顺序的十六进制数字序列。否则的话，这个八位组串以0位最高位（最左边的位）后面接与原十六进制串相同顺序的十六进制数字序列。

例：

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 Bitstring到octetstring

```
bit2oct (bitstring value) return octetstring
```

这个函数把一个单个的比特串值转换为一个八位组。这个八位组串结果表示与原比特串相同的值。

出于转换的目的，遵循下面的规则：bit2oct(value)=hex2oct(bit2hex(value))。

例：

```
bit2oct ('111010111'B)= '01D7'O
```

C.25 Hexstring到bitstring

```
hex2bit (hexstring value) return bitstring
```

这个函数把一个单个的十六进制串值转换位一个单个的比特串值。结果的比特串值表示与原十六进制串相同的值。

出于转换的目的，一个十六进制串应该被转换为一个比特串，此时，十六进制串的数字转换位如下的比特组：

'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.

比特串结果中，4比特组的连续顺序与原十六进制串的十六进制数字顺序相同。

例：

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 Octetstring到hexstring

```
oct2hex (octetstring value) return hexstring
```

这个函数把一个单个的八位组串值转换位一个单个的十六进制串值。结果的十六进制串值表示与原八位组串相同的值。

出于转换的目的，一个八位组串应该被转换为一个与该八位组串的十六进制数字顺序相同的十六进制串，

例：

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 Octetstring到bitstring

```
oct2bit (octetstring value) return bitstring
```

这个函数把一个单个的八位组串值转换为一个比特串。这个比特串结果表示与原八位组串相同的值。

出于转换的目的，遵循下面的规则： oct2bit(value)=hex2bit(oct2hex(value))。

例：

```
oct2bit ('01D7'O)='000111010111'B
```

C.28 Integer到float

```
int2float (integer value) return float
```

这个函数把一个整型值转换位一个浮点型值。

例：

```
int2float(4) = 4.0
```

C.29 Float到integer

```
float2int (float value) return integer
```

这个函数通过删掉浮点参数的小数部分并返回整数部分从而把一个浮点型值转换为一个整型值。

例：

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 随机数生成函数

```
rnd ([float seed]) return float
```

rnd函数返回一个大于等于0且小于1的一个（伪）随机数。通过一个可选的种子值来初始化这个随机数产生器。之后，如果提供了 种子的话，上一个产生的数字将用作下一个随机数的种子。第一次使用**rnd**的时候，没有前一个的初始化，将使用由系统时间计算出来的一个值作为种子。

注意： 如果**rnd**函数每次使用相同的种子值来进行初始化，那么它应该重复产生相同的随机数序列。

使用下面的公式来产生一个给定范围内的随机数：

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
```

```
// 这里upperbound和lowerbound表示范围的上下界。
```

C.31 子串函数

```
substr (any_string_type value, integer index, returncount) return input_string_type
```

这个函数返回**bitstring**、**hexstring**、**octetstring**或任意字符串类型值的一个子串。这个子串的类型就是输入值的源类型，使用第二个参数（index）定义返回子串在串中的起始点，索引从0开始。第三个输入参数定义要返回的子串的长度，长度的单位在表4中定义。

例：

```
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"
```

附录D （范式）：
其它类型与TTCN-3一起使用

D.1 ASN.1与TTCN-3 一起使用

这个附录定义了可选的与TTCN-3一起的ASN.1的使用。

D.1.0 概要

TTCN-3为在TTCN-3模块中使用1997版的ASN.1提供了一个清晰的接口（定义在ITU-T Recommendation X.680 series [7], [8], [9],[10]中）。引入到TTCN-3模块的语言标识符为：

"ASN.1:1997" 代表ASN.1 1997版
"ASN.1:1994" 代表ASN.1 1994版
"ASN.1:1988" 代表ASN.1蓝本版

注意1： 如果基于这些ASN.1版本的协议模块与TTCN-3一起使用，语言标识符"ASN.1:1994"和"ASN.1:1988"表示代替基于ITU-T Recommendations的ASN.1版本表示，而把它们包含在本文中的唯一目的是为它们分配唯一的标识符。

注意2： 当支持"ASN.1:1988"时，应该根据ITU-T Recommendation X.208（蓝本）的语法和句法规则引入ASN.1条目。

注意3： ASN.1:1994和ASN.1:1988参考见附录F.

当ASN.1与TTCN-3一起使用时，ITU-T Recommendation X.680 [7]中章节11.18中所列的关键字不能在一个TTCN-3模块中的标识符。ASN.1关键字应该遵循ITU-T Recommendation X.680 [7]的要求。

D.1.1 ASN.1和TTCN-3类型等价

D.1.1.0 概要

认为表D.1中所列的ASN.1类型月它们的TTCN-3中对应类型等价。

表D.1: ASN.1 和 TTCN-3 类 型 等 价 列 表

ASN.1 类 型	映射到TTCN-3中的等价类型
BOOLEAN	**boolean**
INTEGER	**integer**
REAL (note1)	**float**
OBJECT IDENTIFIER	**objid**
BIT STRING	**bitstring**
OCTET STRING	**octetstring**
SEQUENCE	**record**
SEQUENCE OF	**record of**
SET	**set**
SET OF	**set of**
ENUMERATED	**enumerated**
CHOICE	**union**
VisibleString	**char**(note2), **charstring**
IA5String	**char**(note2), **charstring**
UniversalString	**universal char**(note2), **universal charstring**

注意1： 如果没有限定基数或者显式或隐式地限定基数为10的话，ASN.1的REAL类型与TTCN-3de **float** 等价。ASN.1表示法允许显式的限定，例如通过内部的子类型来限定，不过从ASN.1-TTCN-3映射点的角度看，显式的限定是一个ASN.1值表示方法。也可以通过给定协议的文本描述来定义隐式的限定，也就是说在ASN.1模块之外。然而，在这两种情况中，如果基数在ASN.1中描述，那么TTCN-3的值表示方法可以独立使用。（见章节D.1.2.0中的注意3 ）。

注意2： 只有ASN.1中精确的长度为一个字符的字符子类型才由TTCN-3的基本字符类型等价，例如IA5String (SIZE (1))与TTCN-3的char类型等价，但是IA5String (SIZE (0..1))却不与TTCN-3的char类型等价。

表D.1中给出在一个TTCN-3类型中使用的所有TTCN-3操作符、函数、匹配机制、值标识符等也可以与相应的ASN.1类型一起使用。

D.1.1.1 标识符

在ASN.1标识符到TTCN-3标识符的转换过程中，所有的连线符“-”都应该变成下划线“_”。

例：

```
MyASN1module DEFINITIONS ::=
BEGIN
        Missleading-ASN1-Name::=    INTEGER     -- ASN.1类型标识符使用'-'

END


module MyTTCNModule
{
        import from MyASN1module language "ASN.1:1997" all;

        const Missleading_ASN1_Name ExampleConst:= 1;       // TTCN-3引用ASN.1类型使用下划线
}
```

D.1.2 ASN.1数据结构和值

D.1.2.0 概要

可以在TTCN-3模块中使用ASN.1类型和值。使用一个单独的ASN.1模块来做ASN.1定义，在这个模块中，根据ITU-T Recommendation X.680 [7]的章节 9.3和9.4通过ASN.1的类型和值的引用来对它们进行引用。

例1：

```
MyASN1module DEFINITIONS ::=
BEGIN
     Z::=    INTEGER             -- 简单类型定义


     BMessage::= SET             -- ASN.1类型定义
     {
         name    Name,
         title   VisibleString,
         date    Date
     }

     johnValues Bmessage ::=     -- ASN.1值定义
     {
         name    "John Doe",
         title   "Mr",
         date    "April 12th"
     }

     DefinedValuesForField1 Z ::= {0 | 1} –ASN.1子类型定义
END
```

ASN.1模块应该与ITU-T Recommendation X.680 series [7]、[8]、[9]和[10]的语法相一致。ASN.1类型和值一旦被声明，则它们在TTCN-3模块中可以采用与来自其他TTCN-3模块的常规TTCN-3类型和值相同的使用方法去使用它们（应该引入所需的定义）。当引入ASN.1条目到TTCN-3模块中去的时候，为每个引入的ASN.1条目生成一个相关的类型和值，并根据相关类型和值要求的规则去进行这些基于引入的ASN.1条目的TTCN-3定义和赋值。而且，当匹配基于ASN.1声明的常量、变量、模板或单行表达式时，匹配机制应该使用相关的类型。

通过应用和下面的转换规则从ASN.1条目中派生相关的类型和值（顺序与各个转换的执行顺序相符）：

1) 忽略任意扩政制作者和例外说明。

2) 忽略任意用户定义的限定（见ITU-T Recommendation X.682 [9]的章节9）。

3) 忽略任意的内容限定（见ITU-T Recommendation X.682 [9]的章节9）。

4) 忽略任意的模式限定（见ITU-T Recommendation X.680 [7]的章节48.9）。

5) 从所有使用代替涉及类型表示的值集的方法来限定子分类的限定类型创建等价的子类型

6) 在包含关键字" COMPONENTS OF"的任意的SEQUENCE类型上根据ITU-T Recommendation X.680 [7]章节24.4、在任意的SET类型上根据ITU-T Recommendation X.680 [7]的章节26.2执行COMPONENTS OF转换。

7) 使用通过扩展EMBEDDED PDV相关类型中的内部子类型得到一个完整定义而得的EMBEDDED PDV相关类型来代替任意的EMBEDDED PDV类型（见ITU-T Recommendation X.680 [7]章节32.5）。

8) 使用通过扩展EXTERNAL类型相关类型中的内部子类型得到一个完整的类型定义而得的EXTERNAL相关类型来代替EXTERNAL类型（见ITU-T Recommendation X.680 [7]章节33.5 ）（见注意3）。

9) 使用通过扩展CHARACTER STRING相关类型中的内部子类型定义到一个完整的类型定义而得的CHARACTER STRING相关类型来代替CHARACTER STRING类型（见ITU-T Recommendation X.680 [7] 章节39.5）。

10) 使用通过使用相关的ASN.1类型（见ITU-T Recommendation X.681 [8]的附录C.7）代替INSTANCE OF DefinedObjectClass以及根据表D.1使用相应的TTCN-3等价类型代替相所有应的ASN.1类型而得到的相关类型来代替INSTANCE OF类型。其结果类型是TTCN-3相关类型。

11) 忽略任意剩余的内部子类型（见注意4）。

12) 忽略ASN.1类型中任意指定的数字和比特。在ASN.1值中，用它的值来代替指定的数字，用不以一或多个0结束的比特串来代替任意指定比特或指定比特串，即标识名称出现的比特位置1，其它位置0。

13) 使用通过选择类型引用的类型来代替任意的选择类型；如果指明的选中类型（ITU-T Recommendation X.680 [7]的章节29.1中的Type）是一个受限类型，那么必须在指明的选中类型的父类型做选择。

14) 把任意RELATIVE-OID类型或值转换为一个**objid**类型或值（见注意5）。

15) 使用按如下方法获得的下面各受限类型的相关类型来代替它们（见注意6）：

BMPString: **universal charstring** (char (0,0,0,0) .. char (0,0,255,255));
UTF8String: **universal charstring;**
NumericString: **charstring** 限制到ITU-T Recommendation X.680 [7]章节36.2中给出的字符集；
PrintableString: **charstring** 限制到ITU-T Recommendation X.680 [7]章节36.4中给出的字符集；
TeletexString and T61String: **universal charstring** 限制到ITU-T Recommendation T.61 中给出的字符集（见参考书目）；
VideotexString: **universal charstring** 限制到ITU-T Recommendations T.100 [14]和T.101 [15]中给出的字符集；
GraphicString: **universal charstring;**
General String: **universal charstring.**

16) 使用**charstring** 类型或值来代替任意GeneralizedTime和UTCTime类型或值。

17) 使用**universal charstring**类型或值代替任意ObjectDescripto类型或值。

18) 使用引入对象类（object class）的字段类型符的ASN.1条目来代替它们（见ITU-T Recommendation X.681 [8]章节14）；出于（也仅出于）转换的目的，开放类型（open types）必须使用元类型“OPEN TYPE”代替。

19) 使用引用对象符号信息的ASN.1条目来代替对象符号的所有信息（见ITU-T Recommendation X.681 [8]章节15）。

20) 还原表格（table）约束（见ITU-T Recommendation X.682 [9]章节10）为列表（list）子类型，并忽略所有相关的约束（见注意7）

21) 用下面的相关的TTCN-3类型来代替NULL类型的所有出现：
type enumerated <*identifier*> { NULL }，其中<*identifier*>是根据附录D.1.1.1转换的ASN.1类型引用。

22) 用**anytype**来代替所有对开放类型的引用。

23) 用表D.1中ASN.1类型的等价类型来代替它们，并基于相关类型使用TTCN-3值来代替ASN.1的值（见注意8）。必须使用**anytype**来代替元类型“OPEN TYPE”。

注意1： 仅相关联的类型不允许基于ASN.1 类型的正确的编码值。然而，用于系统来生成正确编码的额外信息依赖于实现，且对用户来说是不可见的，不必基于ASN.1类型和值对它们做适当的声明和赋值。

注意2： 当引入ENUMERATED类型时，用户分配给枚举成分的整型数字也一同被引入。

注意3： EXTERNAL类型的数据-值（data-value）字段可以在编码器判断的时候被编码做一个single-ASN1-type、octet-aligned或arbitrary类型（见ITU-T Recommendation X.690 [11]章节8.18.1）；如果在匹配的时候用户想要实施一个给定的编码格式或仅允许一个特定的编码格式的话，应该对该类型或给定的常量、变量、模板或模板字段使用适当的编码属性（见附录D.1.5.2）。

注意4： 用户在定义TTCN-3值或被内部子类型限制的基于ASN.1类型的模板时，应考虑内部子类型。

注意5： **objid**类型等价限于语法上，且仅用于值表示符。objid值的情况下三个值的前两个值是受限的（见ITU-T Recommendation X.660 [16]）。这个限制并不适用于基于引用的RELATIVE-OID类型的值。

注意6: VisibleString、IA5String和UniversalString有它们自己的TTCN-3等价类型，并且直接进行替代。

注意7: 用户在声明值和模板的时候，应该考虑相关的约束（也可以由工具隐式地处理）。

注意8: 在结构化的ASN.1类型（SET、EXTERNAL等）值中省略可选字段等价于在结构化的TTCN-3值中显式地省略字段。

例2：

```
module MyTTCNModule
{
        import from MyASN1module language "ASN.1:1997" all;

        const Bmessage MyTTCNConst:= johnValues;
        const DefinedValuesForField1 Value1:= 1;
}
```

注意9： 不能从TTCN-3表示符直接访问ASN.1类型和值外的其他定义（即信息目标类或信息目标集）。在TTCN-3模块中引用它们之前，应该先把这些的定义解析为ASN.1模块中的一个类型或值。

D.1.2.1 ASN.1标识符的范围

引入的ASN.1标识符遵循引入TTCN-3类型和值所使用的相同的范围规则（见章节5.3）。

D.1.3 ASN.1中的参数化

允许在TTCN-3模块中引用参数化的ASN.1类型和值定义。然而在TTCN-3模块中使用的所有ASN.1参数化定义应该和他们的实参一起被提供（开放类型实不允许的），而且提供的实参在编译的时候被解析。

TTCN-3核心语言不支持只用于ASN.1特定对象的形参或实参的ASN.1条目的引入，因此，涉及不能在TTCN-3核心语言中直接定义的ASN.1特定的参数化应该在TTCN-3的使用前在ASN.1部分中被解析。这些特定的对象是：

a) 信息对象类（Information Object classes）；

b) 信息对象（Information Objects）；

c) 信息对象集（Information Object Sets）；

例如，因为下面定义了一个采用ASN.1对象集作为实参的TTCN-3类型，所以是不合法的。

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1模块定义

    -- 信息对象类定义
    MESSAGE ::= CLASS { &msgTypeValue   INTEGER UNIQUE,
                                        &MsgFields}

    -- 信息对象定义
    setupMessage MESSAGE ::= {  &msgTypeValue       1,
                                &MsgFields          OCTET STRING}

    setupAckMessage MESSAGE ::= {   &msgTypeValue   2,
                                    &MsgFields      BOOLEAN}

    -- 信息对象集定义
    MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

    -- 由对象集约束的ASN.1类型
    MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
    {
        code    MESSAGE.&msgTypeValue({ MsgSet}),
        Type MESSAGE.&MsgFields({ MsgSet})
}
END

module MyTTCNModule
{
    // TTCN-3模块定义
    import from MyASN1module language "ASN.1:1997" all;

    // 带有对象集作为参数的非法的TTCN-3类型
    type record Q(MESSAGE MyMsgSet) ::= {   Z                   field1,
                                            MyMessage(MyMsgSet) field2}
}
```

为了使这个定义合法，必须使用下面的特别的ASN.1类型My Message1，这用来解析信息对象集参数化并能够直接用在TTCN－3模块中。

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1模块定义

    …

    MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

    -- 特别的ASN.1类型来移去目标集的参数化
    MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
    // TTCN-3模块定义
```

```
    import from MyASN1module language "ASN.1:1997" all;

    // 不带有对象集作为参数的合法的TTCN-3类型
    type record Q := {  Z              field1,
                        MyMessage1     field2}
}
```

D.1.4 定义ASN.1消息模板

D.1.4.0 概要

如果在ASN.1中定义消息，那么对**send**和**receive**事件来说，可以使用ASN.1值语法来描述对SEQUENCE（或可能是SET）的使用。

例：

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1模块定义

    -- 消息定义
    MyMessageType::= SEQUENCE
    {   field1  [1] IA5STRING,          // 象TTCN-3字符串
        field2  [2] INTEGER OPTIONAL,   // 象TTCN-3整型
        field3  [4] Field3Type,         // 象TTCN-3记录
        field4  [5] Field4Type          // 象TTCN-3数组
    }

    Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET STRING},
    Field4Type::= SEQUENCE OF BOOLEAN


    -- 可以有下面的值
    myValue MyMessageType::=
    {
        field1      "A string",
        field2      123,
        field3      {field31 '11011'B, field32 456789, field33 'FF'O},
        field4      {true, false}
    }
END
```

D.1.4.1 ASN.1使用TTCN-3模板语法接收消息

在标准的ASN.1语法中不支持匹配机制，因此，如果要在ASN.1接收消息中使用匹配机制的话，应该使用TTCN-3的用于接收模板的语法作为替代。注意，这个语法包括了为引用ASN.1的SEQUENCE、SET等中各成分的成分引用。

例：

```
import from MyASN1module language "ASN.1:1997" {
    type myMessageType
}

// 在TTCN-3中使用匹配机制的一个消息模板可以是
template myMessageType  MyValue:=
{
    field1 :=           "A"<?>"tr"<*>"g",
    field2 :=           *,
    field3.field31 :=   '110??'B,
    field3.field32 :=   ?,
    field3.field33 :=   'F?'O,
    field4.[0] :=       true,
    field4.[1] :=       false
}

// 下面的语法是有效的等价
template myMessageType  MyValue:=
```

```
{
    field1 := "A"<?>"tr"<*>"g",          // 带有匹配符的串
    field2 := *,                         // 任意整数或空
    field3 := {'110??'B, ?, 'F?'O},
    field4 := {?, false}
}
```

D.1.4.2 模板字段的排序

当TTCN-3模板用于ASN.1类型时，模板中字段顺序的意义依赖于用来定义消息类型的ASN.1构造的类型。例如，如果使用了消息字段，应该按照模板中的顺序发送或配配消息字段。如果使用SEQUENCE或SEQUENCE OF，则可以按照任意顺序发送或匹配的消息字段。

D.1.5 编码信息

D.1.5.0 概要

TTCN-3允许对编码规则以及入各种TTCN-3语言元素相关的编码规则变化的引用，它也可能去定义无效的编码。根据下面的语法使用**with**语句描述编码信息：

例：

```
module MyModule
{
     :
    import from MyASN1module language "ASN.1:1997" {
        type myMessageType
    }
    with {
        encode:= "PER-BASIC-ALIGNED:1997" // MyMessageType所有的实例应该使用PER:1997编码
    }
     :
} // 模块接收
with { encode "BER:1997" } // 整个模块的默认编码（测试套）是BER:1997
```

D.1.5.1 ASN.1编码属性

下面的串是用于ASN.1的预定义（标准）的编码属性：

a) "BER:1997"意味着根据ITU-T Recommendation X.690 (BER) [11]编码；

b) "CER:1997"意味着根据ITU-T Recommendation X.690 (CER) [11]编码；

c) "DER:1997"意味着根据ITU-T Recommendation X.690 (DER) [11]编码；

d) "PER-BASIC-UNALIGNED:1997"意味着根据(Unaligned PER) ITU-T Recommendation X.691 [12]编码；

e) "PER-BASIC-ALIGNED:1997"意味着根据ITU-T Recommendation X.691 (Aligned PER) [12]编码

f) "PER-CANONICAL-UNALIGNED:1997"意味着根据(Canonical Unaligned PER) ITU-T Recommendation X.691 [12]编码；

g) "PER-CANONICAL-ALIGNED:1997"意味着根据ITU-T Recommendation X.691 (Canonical Aligned PER) [12]编码。

D.1.5.2 ASN.1变量属性

下面的串是预定义（标准）的变量属性，当它们仅于ASN.1编码属性结合使用时具有预定义的含义（见附录D.1.5.1）。当预定义属性与其他属性结合使用时或用于不带有属性的TTCN-3对象时，对预定义属性的处理超出了本文的讨论范围（见注意1）：

a) "length form 1"意味着在BER、CER和DER编码的情况下应仅使用长度八位组的短格式（见ITU-T Recommendation X.690 [11]的章节8.1.3）来编码和解码给定的值，或在PER编码（见注意2）任意格式的情况下仅使用单个八位组的长度决定因子（见Recommendation X.691 [12]的章节10.9）来编码和解码给定值。

b) "length form 2"意味着在BER、CER和DER编码的情况下应仅使用长度八位组的长格式（见ITU-T Recommendation X.690 [11]的章节8.1.3）来编码和解码给定的值，或在PER编码（见注意2）任意格式的情况下仅使用两个八位组的长度决定因子（见Recommendation X.691 [12]的章节10.9）来编码和解码给定值。

c) "length form 3"意味着在BER、CER和DER编码的情况下应仅使用长度八位组的无限形式（见ITU-T Recommendation X.690 [11]的章节8.1.3）来编码和解码给定的值。

d) "REAL base 2"意味着应该根据REAL二进制编码格式编码或匹配给定的值。这个属性可以仅用于常量、变量或模板，且当用于任意种类分组的时候（对组或对整个引入语句），它应该只作用于TTCN-3对象。

e) "single-ASN1-type"、"octet-aligned"和"arbitrary"意味着应该使用指定的属性编码格式编码基于ASN.1 EXTERNAL类型的给定值，或者仅接收特定的选择时匹配该值（见Recommendation X.690 [11]的章节8.18）。这个属性可以用于ASN.1的EXTERNAL类型以及仅基于这些类型的常量、变量或模板字段；当用在任意种类分组（对于组或整个引入语句）时，它应该仅作用于这些TTCN-3对象。如果不满足Recommendation X.690 [11]的章节8.18.6到8.18.8中的条件集合和指定的属性的话，将一起一个运行期间错误。

f) "TeletexString"意味着给定值应该作为ASN.1的TeletexString类型被编码和解码（见Recommendation X.690 [11]章节8.20和Recommendation X.691 [12]章节26）。

g) "VideotexString"意味着给定值应该作为ASN.1的VideotexString类型被编码和解码（见Recommendation X.690 [11]章节8.20和Recommendation X.691 [12]章节26）。

h) "GraphicString"意味着给定值应该作为ASN.1的GraphicString类型被编码和解码（见Recommendation X.690 [11]章节8.20和Recommendation X.691 [12]章节26）。

i) "GeneralString"意味着给定值应该作为ASN.1的GeneralString类型被编码和解码（见Recommendation X.690 [11]章节8.20和Recommendation X.691 [12]章节26）。

注意1： 这些属性在实现的时候可以重复使用，不在本节中描述的带有不同含义的指定的编码规则可能导被忽略或导致一个警告/错误指示。然而，这个策略的应用依赖于实现。

注意2： 变量属性的应用可能导致无效的ASN.1编码（例如，对BER的原始值使用无限长度格式或不使用最小数目的长度八位组数）。这么做是有目的的，且用户应该给用于谨慎接收的常量、变量、模板或模板字段分配这些变量属性。

附录E（资料）
有用的类型库

E.1 限制

加到这个库的类型名在整个语言和该库范围内应该是唯一的（即不应该是附录C中定义的名字之一）。这个库中定义的名字不应该被TTCN-3用户用作本附录给出的定义之外的其他定义的标识符。

注意： 因此本附录中给出的类型定义可以在TTCN-3模块中重复，但是没有不同于本附录描述的类型可以使用本附录中用到的标识符来定义它。

E.2 有用的TTCN-3类型

E.2.1 有用的简单基本类型

E.2.1.0 带符号的和不带符号的单字节整数

这些类型支持带符号的–128到127和不带符号的0到255范围内的整型值，用于这些类型的值符号与用于整型的值符号相同。这些类型的值以其在系统中表示的单字节格式编码和解码，独立于实际使用的表示格式。

注意： 这些类型值的编码可以相同，也可以互不相同，而且可以与依赖于实际使用的编码规则的整型（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文研究之外。

定义这些类型为：

```
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

这些类型支持带符号的–32768到32767和不带符号的0到65535范围内的整型值，用于这些类型的值符号与用于整型的值符号相同。这些类型的值以其在系统中表示的双字节格式编码和解码，独立于实际使用的表示格式。

注意： 这些类型值的编码可以相同，也可以互不相同，而且可以与依赖于实际使用的编码规则的整型（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文研究之外。

这些类型的类型定义是：

```
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节整数

这些类型支持带符号的–2147483648到2147483647和不带符号的0到4294967295范围内的整型值，用于这些类型的值符号与用于整型的值符号相同。这些类型的值以其在系统中表示的四字节格式编码和解码，独立于实际使用的表示格式。

注意： 这些类型值的编码可以相同，也可以互不相同，而且可以与依赖于实际使用的编码规则的整型（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文研究之外。

这些类型的类型定义是：

```
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

这些类型支持带符号的–9223372036854775808到9223372036854775807和不带符号的0到18446744073709551615范围内的整型值，用于这些类型的值符号与用于整型的值符号相同。这些类型的值以其在系统中表示的八字节格式编码和解码，独立于实际使用的表示格式。

注意： 这些类型值的编码可以相同，也可以互不相同，而且可以与依赖于实际使用的编码规则的整型（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文研究之外。

这些类型的类型定义是：

```
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

对于二进制浮点运算，这些类型支持ANSI/IEEE标准754（见参考书目）。IEEE 754浮点类型（float）支持基数为10、8位指数、23位尾数和一个符号位的浮点数，IEEE 754的双精度型（double）支持基数为10、11位指数、52位尾数和一个符号位的浮点数，IEEE 754的`extfloat`类型支持基数为10、最小11位指数、最小32位尾数和一个符号位的浮点数，IEEE 754的`extdouble`类型支持基数为10、最小15位指数、最小64位尾数和一个符号位的浮点数。

应该根据IEEE 754定义编码和解码这些类型的值，用于这些类型值的记法与（基数为10的）浮点类型（float）值的记法相同。

注意： 这个类型值的精确编码依赖于实际所使用的编码规则，但编码规则的细节超出了本文研究范围。

用于这些类型的类型定义为：

```
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 有用的字符串类型

E.2.2.0 UTF-8字符串“utf8string”

这个类型支持TTCN-3的**`universal charstring`**类型（见章节6.1.1的d)段落）的所有字符集。该类型值的特点是均为来自这个字符集的`0`、`1`或多个字符，并根据ISO/IEC 10646 [6]附录R中定义的UCS转换格式8（UCS Transformation Format 8，UTF-8）去整个地编码和解码该类型的值（如对一个个字符分别进行）。这个类型值的记法与用于**`universal charstring`**类型值的记法相同。

用于这个类型的定义为：

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 BMP字符串“bmpstring”

这个类型支持ISO/IEC 10646 [6]的Basic Multilingual Plane (BMP)字符集。BMP代表通用多八位组编码字符集（Universal Multiple-octet coded Character Set）的group 00中plane 00的所有字符。该类型值的特点是均为自来BMP的0、1或多个字符，并根据UCS-2编码表示格式（见ISO/IEC 10646 [6]章节14.1）去整个地编码和解码该类型的值（如对一个个字符分别进行）。这个类型的值记法与用于**universal charstring**类型的值记法相同。

注意： 类型"bmpstring"支持TTCN-3的**universal charstring**类型的子集。

用于这个类型的定义为：

```
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”

这个类型支持通用多八位组编码字符集的group 00中plane 00到plane 16的所有字符（见ISO/IEC 10646 [6]）。该类型值的特点是来均为来自这个字符集的0、1或多个字符，并根据ISO/IEC 10646 [6]附录Q中定义的UCS转换格式16（UCS Transformation Format 16，UTF-16）去整个地编码和解码该类型的值（如对一个个字符分别进行）。这个类型值的记法与用于**universal charstring**类型值的记法相同。

注意： 类型" utf16string"支持TTCN-3的**universal charstring**类型的子集。

用于这个类型的定义为：

```
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”

这个类型支持多部分标准ISO/IEC 8859中定义的所有字符表中的所有字符（见参考书目）。该类型值的特点是来均为来自ISO/IEC 8859字符集的0、1或多个字符，，并根据ISO/IEC 8859中描述的编码表示（8-bit编码）去整个地编码和解码该类型的值（如对一个个字符分别进行）。这个类型值的记法与用于**universal charstring**类型值的记法相同。

注意1： 类型"iso8859string"支持TTCN-3的**universal charstring**类型的子集。

注意2： 在每个ISO/IEC 8859字母表中，字符集表中的下面部分（位置02/00到07/14）与ISO/IEC 646 [5]字符集，因此，所有额外的特殊语言字符仅定义在该字符表的上部分（位置10/00到15/15）。因为类型"iso8859string"定义做了TTCN-3的**universal charstring**类型的一个子集，所以任意ISO/IEC 8859字母表的任意编码字符表示可以被映射到一个来自ISO/IEC 10646 [6]基本拉丁文（Basic Latin）或Latin-1补充字符表的等价字符（按8比特编码时的带有相同编码表示的字符）。

用于这个类型的定义为：

```
type universal charstring iso8859string ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                                        with { variant "8 bit" };
```

E.2.3 有用的结构化类型

E.2.3.0 定点十进制数字文字

这个类型支持IDL Syntax and Semantics version 2.6（见参考书目）中定义定点十进制数字文字（Fixed-point decimal literal）的使用，使用一个整数部分、一个小数点和一个分数部分来描述它。整数和分数部分都是由一个十进制（基数为10）数字串组成。数字的位数存贮在“digits”中，分数部分的长度由“scale”给出，而数

字本身存贮在“value_”中。这个类型值的记法与用于记录类型值的记法相同，该类型值应该编码和解码为IDL定点十进制值。

注意： 这个类型值的精确编码依赖于实际所使用的编码规则，但编码规则的细节超出了本文研究范围。

用于这个类型的定义为：

```
type record IDLfixed {
                        unsignedshort   digits,
                        short   scale,
                        charstring  value_
            }
                    with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

附录F（资料）
参考书目

- 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".
- ITU-T Recommendation X.208: "Specification of Abstract Syntax Notation One (ASN.1)".
- ISO/IEC 8859-1: "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 - IDL Syntax and Semantics". Version 2.6, FORMAL/01-12-01, December 2001.
- IEEE 754 (1985): "Binary Floating-Point Arithmetic".
- ETSI ES 201 873-4 (V2.2.0): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 操作语义".
- ITU-T Recommendation T.61 (1988): "Character repertoire and coded character sets for the international teletex service".

历史

文档历史		
V1.1.1	2001年3月	出版
V1.1.2	2001年7月	出版
V2.2.0	2002年5月	成员通过过程 MV 20020712: 2002-05-14 to 2002-07-12
V2.2.1	2003年2月	出版