МЕЖДУНАРОДНЫЙ СОЮЗ ЭЛЕКТРОСВЯЗИ

МСЭ-Т Z.140

СЕКТОР СТАНДАРТИЗАЦИИ
ЭЛЕКТРОСВЯЗИ МСЭ

(03/2006)

СЕРИЯ Z: ЯЗЫКИ И ОБЩИЕ АСПЕКТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ СИСТЕМ ЭЛЕКТРОСВЯЗИ

Методы формального описания (FDT) – Нотация по тестированию и управлению тестированием (TTCN)

Нотация по тестированию и управлению тестированием версии 3 (TTCN-3): Базовый язык

Рекомендация МСЭ-Т Z.140

РЕКОМЕНДАЦИИ МСЭ-Т СЕРИИ Z

ЯЗЫКИ И ОБЩИЕ АСПЕКТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ СИСТЕМ ЭЛЕКТРОСВЯЗИ

МЕТОДЫ ФОРМАЛЬНОГО ОПИСАНИЯ (FDT)	
Язык спецификации и описания (SDL)	Z.100–Z.109
Применение методов формального описания	Z.110–Z.119
Диаграмма последовательности сообщений (MSC)	Z.120–Z.129
Расширенный язык описания объектов (eODL)	Z.130–Z.139
Нотация тестирования и управления тестированием (TTCN)	**Z.140–Z.149**
Нотация требований пользователя (URN)	Z.150–Z.159
ЯЗЫКИ ПРОГРАММИРОВАНИЯ	
CHILL: язык высокого уровня МСЭ-Т	Z.200–Z.209
ЯЗЫК "ЧЕЛОВЕК–МАШИНА"	
Общие принципы	Z.300–Z.309
Базисный синтаксис и диалоговые процедуры	Z.310–Z.319
Расширенный язык MML для видеотерминалов	Z.320–Z.329
Спецификация интерфейса "человек–машина"	Z.330–Z.349
Информационно-ориентированные интерфейсы "человек–машина"	Z.350–Z.359
Интерфейсы "человек–машина" для управления сетями электросвязи	Z.360–Z.379
КАЧЕСТВО	
Качество программного обеспечения электросвязи	Z.400–Z.409
Аспекты качества рекомендаций, относящихся к протоколам	Z.450–Z.459
МЕТОДЫ	
Методы проверки и тестирования	Z.500–Z.519
ПРОМЕЖУТОЧНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ	
Среда распределенной обработки	Z.600–Z.609

Для получения более подробной информации просьба обращаться к перечню Рекомендаций МСЭ-Т.

Рекомендация МСЭ-Т Z.140

Нотация по тестированию и управлению тестированием версии 3 (TTCN-3): Базовый язык

Резюме

В настоящей Рекомендации определяется базовый язык TTCN-3 (нотация по тестированию и управлению тестированием-3), предназначенная для спецификации типов тестов, независимых от платформ, методов тестирования, уровней протоколов и протоколов. TTCN-3 может использоваться для спецификации всех типов тестов реагирующих систем через различные порты связи. Типичными областями применения являются тестирование протоколов (включая протоколы систем подвижной связи и интернет), тестирование служб (включая дополнительные услуги), тестирование модулей, тестирование платформ, основанных на COBRA, и тестирование прикладных программных интерфейсов (API). Спецификация тестовых примеров для протоколов физического уровня выходит за рамки данной Рекомендации.

Базовый язык TTCN-3 может быть выражен во множестве форматов представления. В то время как эта Рекомендация определяет Базовый язык, МСЭ-Т Рек. Z.141 определяет Табличный формат для TTCN (TFT) и МСЭ-Т Рек. Z.142 определяет Графический формат для TTCN (GFT). Спецификация этих форматов выходит за рамки данной Рекомендации. Базовый язык удовлетворяет трем целям:

1) как обобщенный основанный на тексте тестовый язык;
2) как стандартизированный формат обмена тестовыми наборами TTCN между инструментами TTCN;
3) как семантическая основа (и где уместно, синтаксическая основа) для различных форматов представления.

Базовый язык может использоваться независимо от форматов представления. Однако ни табличный формат, ни графический формат не могут использоваться без базового языка. Использование и реализация этих форматов представления должны быть на основе базового языка.

Источник

Рекомендация МСЭ-Т Z.140 утверждена 16 марта 2006 года 17-й Исследовательской комиссией МСЭ-Т (2005–2008 гг.) в соответствии с процедурой, изложенной в Рекомендации МСЭ-Т A.8.

ПРЕДИСЛОВИЕ

Международный союз электросвязи (МСЭ) является специализированным учреждением Организации Объединенных Наций в области электросвязи и информационно-коммуникационных технологий (ИКТ). Сектор стандартизации электросвязи МСЭ (МСЭ-Т) – постоянный орган МСЭ. МСЭ-Т отвечает за изучение технических, эксплуатационных и тарифных вопросов и за выпуск Рекомендаций по ним с целью стандартизации электросвязи на всемирной основе.

На Всемирной ассамблее по стандартизации электросвязи (ВАСЭ), которая проводится каждые четыре года, определяются темы для изучения Исследовательскими комиссиями МСЭ-Т, которые, в свою очередь, вырабатывают Рекомендации по этим темам.

Утверждение Рекомендаций МСЭ-Т осуществляется в соответствии с процедурой, изложенной в Резолюции 1 ВАСЭ.

В некоторых областях информационных технологий, которые входят в компетенцию МСЭ-Т, необходимые стандарты разрабатываются на основе сотрудничества с ИСО и МЭК.

ПРИМЕЧАНИЕ

В настоящей Рекомендации термин "администрация" используется для краткости и обозначает как администрацию электросвязи, так и признанную эксплуатационную организацию.

Соблюдение положений данной Рекомендации осуществляется на добровольной основе. Однако данная Рекомендация может содержать некоторые обязательные положения (например, для обеспечения функциональной совместимости или возможности применения), и в таком случае соблюдение Рекомендации достигается при выполнении всех указанных положений. Для выражения требований используются слова "следует", "должен" ("shall") или некоторые другие обязывающие выражения, такие как "обязан" ("must"), а также их отрицательные формы. Употребление таких слов не означает, что от какой-либо стороны требуется соблюдение положений данной Рекомендации.

ПРАВА ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

МСЭ обращает внимание на вероятность того, что практическое применение или выполнение настоящей Рекомендации может включать использование заявленного права интеллектуальной собственности. МСЭ не занимает какую бы то ни было позицию относительно подтверждения, действительности или применимости заявленных прав интеллектуальной собственности, независимо от того, доказываются ли такие права членами МСЭ или другими сторонами, не относящимися к процессу разработки Рекомендации.

На момент утверждения настоящей Рекомендации МСЭ не получил извещение об интеллектуальной собственности, защищенной патентами, которые могут потребоваться для выполнения настоящей Рекомендации. Однако те, кто будет применять Рекомендацию, должны иметь в виду, что вышесказанное может не отражать самую последнюю информацию, и поэтому им настоятельно рекомендуется обращаться к патентной базе данных БСЭ по адресу: http://www.itu.int/ITU-T/ipr/.

СОДЕРЖАНИЕ

Рекомендация МСЭ-Т Z.140

Нотация по тестирования и управлению тестированием версии 3 (TTCN-3): Базовый язык

1 Область применения

В настоящей Рекомендации определяется Базовый язык TTCN-3. TTCN-3 может использоваться для спецификации всех типов тестов реагирующих систем через различные порты связи. Типичными областями применения являются тестирование протоколов (включая протоколы систем подвижной связи и Интернет), тестирование служб (включая дополнительные услуги), тестирование модулей, тестирование платформ на базе CORBA, тестирование API и др. Применение TTCN-3 не ограничено тестированием на соответствие; этот язык может использоваться для многих других видов тестирования, в том числе для тестирования взаимодействия, устойчивости, ухудшений, системности и интеграции. Спецификация тестовых примеров для протоколов физического уровня выходят за рамки данной Рекомендации.

Нотация TTCN-3 предназначена для использования при спецификации тестовых примеров, которые не зависят от методов тестирования, уровней и протоколов. Для TTCN-3 определены разные форматы представления, такие как табличный формат представления (Рек. МСЭ-Т Z.141 [1]) и графический формат представления (Рек. МСЭ-Т Z.142 [2]). Спецификация этих форматов выходит за рамки данной Рекомендации.

Хотя при проектировании TTCN-3 учитывалась возможная реализация трансляторов и компиляторов TTCN-3, средства реализации Выполнимых тестовых последовательностей (ETS) из числа Абстрактных тестовых последовательностей (ATS) в данной Рекомендации не рассматриваются.

2 Справочная литература

В перечисленных ниже Рекомендациях МСЭ-Т и другой справочной литературе содержатся положения, которые посредством ссылок на них в этом тексте составляют основные положения данной Рекомендации. На момент опубликования, действовали указанные редакции документов. Все Рекомендации и другая справочная литература, являются предметом корректировки, в связи с чем пользователям данной Рекомендации настоятельно рекомендуется изыскать возможность для использования самых последних изданий Рекомендации и справочной литературы, перечисленной ниже. Регулярно публикуется перечень действующих Рекомендаций МСЭ-Т. Ссылка на документ в рамках этой Рекомендации не даёт ему, как отдельному документу, статуса Рекомендации.

[1] ITU-T Recommendation Z.141 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Tabular presentation format (TFT)*.

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

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

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

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

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

[7] ITU-T Recommendation X.290 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts*.

ISO/IEC 9646-1:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts*.

[8] ITU-T Recommendation X.292 (2002), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)*.

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

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

ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.

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

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

3 Определения и сокращения

3.1 Определения

Для целей настоящей Рекомендации применяются термины, определенные в Рекомендации МСЭ-Т X.290 [7], Рекомендации МСЭ-Т X.292 [8], а также следующие термины:

3.1.1 действительный параметр: Значение, выражение, шаблон или ссылка на имя (идентификатор), которое должно передаваться в качестве параметра вызываемому объекту (функции, тестовой последовательности, altstep и так далее) таким образом, как это определяется в месте вызова.

Примечание. – Количество, порядок и типы всех действительных параметров, которые должны передаваться при каком-либо вызове, должны согласовываться со списком формальных параметров, которые определяются для вызываемого объекта.

3.1.2 базовые типы: набор заранее определенных типов TTCN-3, которые описываются в разделах 6.1.0 и 6.1.1.

Примечание. – Ссылка на базовые типы производится по их имени.

3.1.3 совместимый тип: TTCN-3 не имеет строгой классификации по типам, но этот язык требует совместимости типов.

Примечание. – Переменные, константы, шаблоны и т. п. имеют совместимые типы, если они удовлетворяют условиям, указанным в разделе 6.7.

3.1.4 порт связи: Абстрактный механизм, облегчающий связь между тестовыми компонентами. Порт связи моделируется в виде очереди типа FIFO в направлении приема. Порты могут быть на базе сообщений, на базе процедур или на базе и того и другого.

3.1.5 типы данных: Общее название простых базовых типов, базовых типов строк, структурированных типов, специального типа данных **anytype** и основанных на них типах данных, которые определяются пользователем (см. Таблицу 3).

3.1.6 определенные типы (определенные в TTCN-3 типы): набор всех предопределенных типов TTCN-3 (базовые типы, все структурированные типы, тип **anytype**, типы адрес, порт и компонент и тип по умолчанию), а также всех типов, определенных пользователем либо в модуле, либо импортированных из других модулей TTCN-3.

3.1.7 динамическая параметризация: Тип параметризации, в котором действительные параметры зависят от событий, которые происходят в момент выполнения программы, то есть значение действительного параметра является значением, которое получено во время выполнения программы или же связано с полученнным значением при помощи логического соотношения.

3.1.8 особое состояние: При коммуникациях на основе процедур: особое состояние (если оно определено) порождается отвечающим объектом, когда на удаленный вызов процедуры он не может дать обычный ожидаемый ответ.

3.1.9 формальный параметр: введенное имя или введенная ссылка на шаблон (идентификатор), которая не была разрешена во время определения объекта (функции, тестовой последовательности, altstep и так далее), а определяется при его вызове.

Примечание. – Действительные значения или шаблоны (или же их имена), которые должны использоваться вместо формальных параметров, передаются от места вызова объекта (см. также определение действительного параметра).

3.1.10 глобальная видимость: Атрибут объекта (параметр модуля, константа, шаблон и так далее), на идентификатор которого можно ссылаться везде в пределах того модуля, где он определен, включая все функции, тестовые последовательности и altstep, которые определены в том же самом модуле и в управляющей части данного модуля.

3.1.11 заявление о соответствии реализации (ЗСР): см. Рекомендацию МСЭ-Т X.290 [7].

3.1.12 дополнительная информация о реализации для тестирования (ДИРТ): см. Рекомендацию МСЭ-Т X.290 [7].

3.1.13 тестируемая реализация (ТР): см. Рекомендацию МСЭ-Т X.290[7].

3.1.14 известные типы: набор всех предопределенных типов TTCN-3, типов, которые определены в модуле TTCN-3, а также типов, импортированных в данный модуль из других модулей TTCN-3 или из модулей, не являющихся модулями TTCN-3.

3.1.15 левая сторона (выражения присваивания): значение или идентификатор шаблона переменной или же название поля в значении структурированного типа или переменной шаблона (включая, при наличии, индекс массива), которое располагается с левой стороны от символа присваивания (:=).

Примечание. – Константа, параметр модуля, таймер, название поля структурированного типа или заголовок шаблона (включая тип шаблона, название и список формальных параметров), которые располагаются с левой стороны от символа присваивания (:=) в декларациях и/или в модифицированных определениях шаблонов, не относятся к области рассмотрения данного определения так как не являются частью выражения присваивания.

3.1.16 локальная видимость: атрибут объекта (константы, переменной и так далее), на идентификатор которого можно ссылаться только в пределах функции, тестовой последовательности или altstep, в которой он определен.

3.1.17 главный тестовый компонент (MTC): см. Рекомендацию МСЭ-Т X.292 [8].

3.1.18 передача параметра по значению: способ передачи параметров, при котором значения параметров оцениваются перед входом в параметризованный объект.

Примечание. – Передаются только значения аргументов, а изменения аргументов, которые производятся в пределах вызванного объекта и с точки зрения вызывающего объекта не оказывают влияния на действительные аргументы.

3.1.19 передача параметра по ссылке: способ передачи параметров, при котором перед входом функцию, altstep и так далее аргументы не оцениваются и вызывающая процедура (функция, altstep и так далее) передает вызываемой процедуре ссылку на параметры.

Примечание. – Все изменения, которые производятся с аргументами в пределах вызванной процедуры, с точки зрения вызывающего объекта оказывают влияние на действительные аргументы.

3.1.20 параллельный тестовый компонент (PTC): см. Рекомендацию МСЭ-Т X.292 [8].

3.1.21 правая сторона (выражения присваивания): значение, ссылка на шаблон или идентификатор параметра подписи, которое располагается с правой стороны от символа присваивания (:=).

Примечание. – Выражения и ссылки на шаблоны, которые располагаются с правой стороны от символа присваивания (:=) в константе, параметре модуля, таймере, шаблоне или в модифицированном определении шаблона, не относятся к области рассмотрения данного определения так как не являются частью выражения присваивания.

3.1.22 корневой тип: базовый тип, структурированный тип, специальный тип данных, специальный конфигурационный тип или специальный тип по умолчанию, который служит как основа для определенного пользователем типа TTCN-3.

3.1.23 статическая параметризация: вид параметризации, при котором действительные параметры не зависят от событий, происходящих во время выполнения программы; это означает, что они известны во время компиляции или, в случае модульных параметров, до начала выполнения тестовой последовательности (например, известны на основании спецификации тестовой последовательности, включая импортированные определения, или же системе тестирования известно их значение до времени выполнения)

Примечание. – Все типы известны в момент компиляции, то есть осуществляется статическое связывание.

3.1.24 сильная типизация: строгое соблюдение совместимости для эквивалентности типа имени без каких-либо исключений.

3.1.25 Тестируемая система (SUT): см. Рекомендацию МСЭ-Т X.290 [7].

3.1.26 шаблон: шаблоны TTCN-3 представляют специальные структуры данных, предназначенные для тестирования, которые используются либо для передачи набора различных значений, либо для проверки того, соответствует ли набор полученных значений спецификации шаблона.

3.1.27 тестовое поведение (или поведение): вариант тестирования или функция, которая запускается над тестируемым компонентом при выполнении операторов компонента **execute** или **start** и при рекурсивном вызове всех функций и **altstep**.

Примечание. – В процессе выполнения варианта тестирования каждый компонент тестирования обладает собственным поведением и, следовательно, в системе тестирования могут одновременно исполняться несколько поведений тестирования (то есть, вариант тестирования может рассматриваться как коллекция поведений тестирования).

3.1.28 тестовый пример: см. Рекомендацию МСЭ-Т X.290 [7].

3.1.29 ошибка варианта тестирования: см. Рекомендацию МСЭ-Т X.290 [7].

3.1.30 тестовая последовательность (test suite): набор модулей TTCN-3, которые содержат полностью определенный набор вариантов тестирования, которые могут по выбору дополняться одним или более модулем управления TTCN-3.

3.131 система тестирования: см. Рекомендацию МСЭ-Т X.290 [7].

3.1.32 интерфейс системы тестирования: компонент тестирования, который обеспечивает отображение портов, имеющихся в (абстрактной) системе тестирования TTCN-3, на те, которые присутствуют в тестируемой системе (SUT).

3.1.33 совместимость типа: особенность языка, которая позволяет использовать значения, выражения или шаблоны заданного типа в качестве действительных значений другого типа (например, в выражениях присваивания, в качестве действительных параметров при вызове функции, при ссылке на шаблон и так далее, или же в качестве возвращаемого функцией значения).

Примечание. – Как тип, так и текущее значение данного значения, выражения или шаблона должны являться совместимыми с другим типом.

3.1.34 параметризация значения: способность передавать значение или шаблон в параметризованный объект в качестве действительного параметра.

Примечание. – Данный параметр с действительным значением затем завершает спецификацию данного объекта.

3.1.35 тип, определенный пользователем: тип, который определяется как подтип базового типа или же с помощью декларирования структурированного типа.

Примечание. – На типы, определенные пользователем, ссылаются с помощью их идентификаторов (названий, имен).

3.1.36 нотация значения: нотация, при помощи которой идентификатор ассоциируется с заданным значением или диапазоном определенного типа.

Примечание. – Значениями могут являться константы или переменные.

3.2 Сокращения

В настоящей Рекомендации используются следующие сокращения:

API	Application Programming Interface	Прикладной программный интерфейс
ATS	Abstract Test Suite	Абстрактная тестовая последовательность
BMP	Basic Multilingual Plane	Базовая многоязыковая плоскость
BNF	Backus-Naur Form	Форма Бэкуса–Наура

CORBA	Common Object Request Broker Architecture	Общая архитектура посредника объектных запросов
ETS	Executable Test Suite	Выполнимая тестовая последовательность
FIFO	First In First Out	Первым пришел – первым обслужен
ICS	Implementation Conformance Statement	Заявление о соответствии реализации (ЗСРП)
IRV	International Reference Version	Международная справочная версия
IUT	Implementation Under Test	Тестируемая реализация
IXIT	Implementation eXtra Information for Testing	Дополнительная информация о реализации для тестирования
MTC	Main Test Component	Главный тестовый компонент
PTC	Parallel Test Component	Параллельный тестовый компонент
SUT	System Under Test	Тестируемая система (ТС)
TSI	Test System Interface	Интерфейс тестовой системы

4 Введение

4.0 Общие положения

TTCN-3 – гибкий и мощный язык, применимый для спецификации всех типов тестов реагирующих систем через различные интерфейсы связи. Типичными областями применения являются тестирование протоколов (включая протоколы систем подвижной связи и Интернет), тестирование служб (включая дополнительные услуги), тестирование модулей, тестирование платформ на базе CORBA, тестирование API и др. Применение TTCN-3 не ограничено аттестационным тестированием; этот язык может использоваться для многих других видов тестирования, в том числе для тестирования взаимодействия, устойчивости, ухудшений, систем и интеграции.

TTCN-3 имеет следующие основные характеристики:

- способность определять конфигурации динамического одновременного тестирования;
- работа с системами связи, основанными на процедурах и на сообщениях;
- способность определять информацию о кодировании и другие атрибуты (включая пользовательские расширения);
- способность определять шаблоны данных и сигнатур с эффективными механизмами сопоставления;
- параметризация значений;
- присвоение и обработка тестовых вердиктов;
- механизмы параметризации тестовой последовательности и выбора тестового примера;
- комбинированное использование TTCN-3 с другими языками;
- четко определенные синтаксис, формат обмена и статическая семантика;
- разные форматы представления (например, табличный и графический форматы представления);
- точный алгоритм выполнения (операционная семантика).

4.1 Базовый язык и форматы представления

Спецификация TTCN-3 разделена на несколько частей. Первой частью спецификации, определяемой в этой Рекомендации, является базовый язык. Второй частью, определяемой в Рекомендации МСЭ-Т Z.141 [1], является табличный формат представления. Третьей частью, определяемой в Рекомендации МСЭ-Т Z.142 [2], является графический формат представления. Четвертая часть, Рекомендация МСЭ-Т Z.143 [3], содержит операционную семантику языка. Пятая часть, Рекомендация МСЭ-Т Z.144 [4], определяет Интерфейс времени исполнения TTCN-3 (TRI), шестая часть, Рекомендация МСЭ-Т Z.145 [5], определяет Управляющий интерфейс TTCN-3 (TCI), и седьмая часть, Рекомендация МСЭ-Т Z.146 [6], определяет использование определений ASN.1 совместно с TTCN-3.

Базовый язык служит трем целям:

a) в качестве обобщенного языка тестов на базе текста, то есть в своей собственной сфере;

b) в качестве стандартного формата обмена тестовыми последовательностями TTCN-3 между инструментами TTCN-3;

c) в качестве семантического базиса (а где это уместно – и синтаксического базиса) для различных форматов представления.

Базовый язык может использоваться независимо от форматов представления. Однако ни табличный, ни графический форматы не могут использоваться без базового языка. Использование и реализация этих форматов представления должны осуществляться на основе базового языка.

Табличный и графический форматы – первые из ожидаемого набора различных форматов представления. Такими другими форматами могут быть стандартизованные форматы представления либо патентованные форматы представления, которые определяются самими пользователями TTCN-3. Эти дополнительные форматы не определяются в настоящей Рекомендации.

TTCN-3 может факультативно использоваться с другими нотациями тип-значение, в этом случае в качестве альтернативного синтаксиса типов и значений данных могут использоваться определения на других языках. Прочие части данного стандарта специфицируют использование других языков совместно с TTCN-3. Поддержка других языков не ограничивается спецификациями серии Рекомендаций МСЭ-Т Z.140, однако для поддержки тех языков, для которых определяется совместное использование с TTCN-3, должны применяться правила, которые приводятся в настоящей Рекомендации.

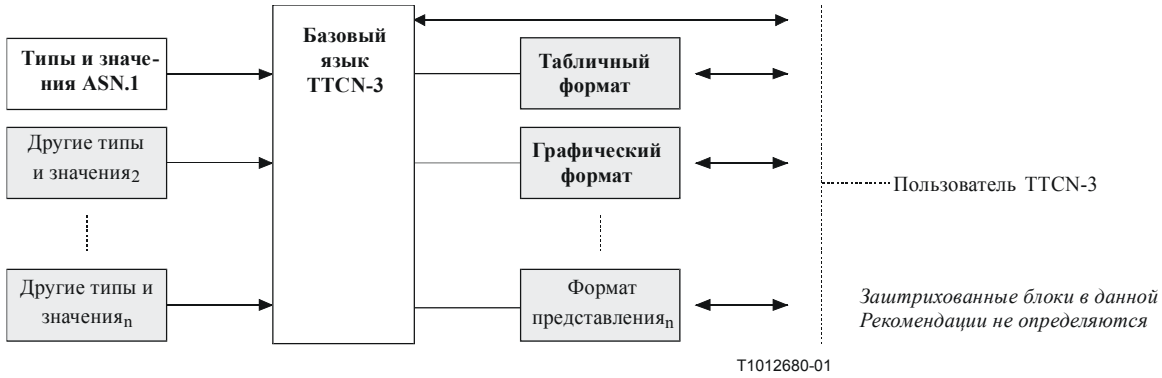


Рисунок 1/Z.140 – Точка зрения пользователя на базовый язык и различные форматы представления

Базовый язык определяется полным синтаксисом (см. Приложение A) и операционной семантикой (МСЭ-Т Z.143 [3]). Он содержит минимальную статическую семантику (определенную в основном тексте данной Рекомендации и в Приложении A), которая не ограничивает использование этого языка из-за некоторой нижележащей прикладной области или методологии.

4.2 Единство спецификации

Язык специфицируется синтаксически и семантически с помощью текстового описания в тексте настоящей Рекомендации (разделы с 5 по 28) и формализованным образом в Приложении A. В каждом из этих случаев, когда текстовое описание не является исчерпывающим, то оно дополняется формальным описанием. Если текстовое и формальное описание противоречат друг другу, то приоритет отдается последнему.

4.3 Соответствие

Для того чтобы реализация могла претендовать на соответствие настоящей версии языка, необходимо реализовать все функции, определенные в настоящей Рекомендации, в соответствии с требованиями, которые специфицируются в данной Рекомендации и в Рекомендации МСЭ-Т Z.143 [3].

5 Элементы базового языка

5.0 Общие положения

Единицей высшего уровня TTCN-3 является модуль. Модуль не может разделяться на субмодули. Он может импортировать определения из других модулей. Модули могут иметь параметры модуля для того, чтобы обеспечить параметризацию тестовой последовательности

Модуль состоит из определяющей части и управляющей части. Определяющая часть модуля определяет тестовые компоненты, порты связи, типы данных, константы, шаблоны тестовых данных, функции, сигнатуры для вызовов процедур в порты, тестовые примеры и др.

Управляющая часть модуля вызывает тестовые примеры и управляет их выполнением. Управляющая часть может также объявлять (местные) переменные и т. п. Программные команды (такие как if-else и do-while) могут использоваться для определения порядка выбора и выполнения конкретных тестовых примеров. Концепция глобальных переменных не поддерживается в TTCN-3.

В TTCN-3 имеется ряд предопределенных базовых типов данных, а также структурированных типов, например записей, наборов, объединений, перечислительных типов и массивов.

Особый вид структуры данных, называемый шаблоном, обеспечивает механизмы параметризации и сопоставления для определения тестовых данных, передаваемых или принимаемых через тестовые порты. Работа этих портов обеспечивает возможности связи, основанной как на сообщениях, так и на процедурах. Вызовы процедуры могут использоваться для тестирования реализаций, не применяющих сообщения.

Режим динамических тестов выражается тестовыми примерами. Программные команды TTCN-3 содержат эффективные механизмы описания поведения (режима), такие как случаи с альтернативным приемом связи и с таймером, перемежение и безусловное поведение (по умолчанию). Поддерживаются также механизмы присвоения и регистрации тестового вердикта.

Наконец, элементам языка TTCN-3 могут быть присвоены атрибуты, такие как информация кодирования и атрибуты визуального отображения. Могут также указываться (нестандартизованные) атрибуты, определяемые пользователем.

Таблица 1/Z.140 – Обзор элементов языка TTCN-3

Элемент 7 языка	Соответству-ющее ключевое слово	Указывается в определе-ниях модулей	Указывается в управлении модулей	Указывается в функциях/ altstep/ тестовых примерах	Указывается в типе тестового компонента
Определение модуля TTCN-3	**module**				
Импорт определений из другого модуля	**import**	Да			
Группирование определений	**group**	Да			
Определения типов данных	**type**	Да			
Определения портов связи	**port**	Да			
Определения тестовых компонентов	**component**	Да			
Определения сигнатур	**signature**	Да			
Определения внешних функций/констант	**external**	Да			
Определения констант	**const**	Да	Да	Да	Да
Определения шаблонов данных/сигнатур	**template**	Да	Да	Да	Да
Определения функций	**function**	Да			
Определения altstep	**altstep**	Да			
Определения тестовых примеров	**testcase**	Да			
Объявления значения переменной	**var**		Да	Да	Да
Объявления шаблона переменной	**var template**		Да	Да	Да
Определения таймера	**timer**		Да	Да	Да
ПРИМЕЧАНИЕ. – Понятия "определение" и "объявление" для переменных, констант, типов и других элементов языка в рамках данной Рекомендации используются, заменяя друг друга. Различие между этими двумя понятиями полезно только для целей реализации на таких языках программирования, как C и C++. На уровне TTCN-3 данные два понятия имеют эквивалентное значение.					

5.1 Порядок следования элементов языка

В общем случае порядок, в котором могут делаться объявления может быть произвольным. Внутри блока команд и объявлений, таких как тело функции или ветвь команды **if-else**, все объявления (если таковые имеются) делаются только в начале блока.

ПРИМЕР:

```
// Это – законное смешивание объявлений TTCN-3
   :
var MyVarType MyVar2 := 3;
const integer MyConst:= 1;
if (x > 10)
{
   var integer MyVar1:= 1;
   :
   MyVar1:= MyVar1 + 10;
   :
}
   :
```

Определения в определяющей части модуля могут даваться в любом порядке. Однако в пределах управляющей части модуля, определений тестовых примеров, функций altstep все необходимые определения должны быть сделаны заранее. Это означает, что никогда не должны использоваться местные переменные, местные таймеры и местные константы до того, как они будут определены. Единственным исключением из этого правила являются метки. Ссылки вперед на метку могут использоваться в командах **goto** до того места, где расположена данная метка (см. 19.5).

5.2 Параметризация

5.2.0 Статическая и динамическая параметризация

TTCN-3 поддерживает параметризацию *значения* со следующими ограничениями:

a) не могут быть параметризованы следующие элементы языка: **const, var, timer, control, group** и **import**;

b) элемент языка **module** позволяет *статическую* параметризацию значения для поддержки параметров тестовой последовательности, то есть эта параметризация может быть разрешена или не разрешена во время трансляции (компиляции), но должна быть разрешена при начале времени выполнения (то есть является *статической* во время выполнения). Это означает, что во время выполнения значения параметров модуля видны в глобальном масштабе, но являются неизменяемыми;

c) все определяемые пользователем определения типов (включая определения структурированных типов, таких, как **record, set** и т. п.) и специальный тип конфигурации **address** поддерживают параметризацию *статического* значения, то есть эта параметризация должна разрешаться во время трансляции;

d) элементы языка **template, signature, testcase** и **function** поддерживают параметризацию *динамических* значений (то есть эта параметризация должна быть разрешена во время выполнения);

В сводной таблице 2 показано, какие элементы языка могут быть параметризованы и что может быть перенесено в них в качестве параметров.

Таблица 2/Z.140 – Обзор параметризуемых элементов языка TTCN-3

Ключевое слово	Параметризация значения	Типы значений, которые могут появляться в списках формальных/реальных параметров
module	Статическое при запуске времени выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов и типа **address**.
type (Примечание 1)	Статическое во время трансляции	*Значения для:* всех базовых типов, всех определяемых пользователем типов и типа **address**.
template	Динамическое во время выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа **address**, и **template**.
function	Динамическое во время выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа address, типа **component**, типа **port**, **default template** и **timer**.
altstep	Динамическое во время выполнения	*Значения для:* всех бызовых типов, всех определенных пользователем типов, типа **address**, типа **component**, типа **port**, **default template** и **timer**.
testcase	Динамическое во время выполнения	*Значения для:* всех бызовых типов, всех определенных пользователем типов, типа **address** и **template**.
signature	Динамическое во время выполнения	*Значения для:* всех бызовых типов, всех определенных пользователем типов, типа **address** и типа **component**.
ПРИМЕЧАНИЕ 1. – **record of, set of enumerated, port, component** и **подтип sub-type definitions** не разрешают параметризацию. ПРИМЕЧАНИЕ 2. – Примеры синтаксиса и специфическое использование параметризации совместно с различными элементами языка приводятся в соответствующих разделах настоящей Рекомендации.		

5.2.1 Передача параметров по ссылке и по значению

5.2.1.0 Общие положения

Все действительные параметры базовых типов, базовых цепочечных типов, определяемых пользователем структурированных типов, типа address и типа component переносятся безусловно (по умолчанию) в значении. Это может быть факультативно отмечено ключевым словом **in**. Для передачи параметров перечисленных выше типов по ссылке должно использоваться ключевое слово **out** или **inout**.

Таймеры и порты всегда передаются по ссылке. Параметры таймера обозначаются ключевым словом **timer.** Параметры порта обозначаются их типом порта. Ключевое слово **inout** может использоваться факультативно для указания на передачу по ссылке.

5.2.1.1 Параметры, передаваемые по ссылке

Передача параметров по ссылке имеет следующие ограничения:

a) только список формальных параметров для элементов **altstep, functions**, **signatures** и **testcases** может содержать параметры, передаваемые по ссылке.

ПРИМЕЧАНИЕ. – Имеются дальнейшие ограничения на способ использования параметров, передаваемых по ссылке, для signatures (см. раздел 23) и altstep (см. 16.2.1 и 21.3.1).

b) реальные параметры должны быть только переменными значения или шаблона, портами или таймерами.

ПРИМЕР:

```
function MyFunction (inout boolean MyReferenceParameter) { … };
// MyReferenceParameter передается по ссылке. Реальный параметр может
// считываться и устанавливаться из этой функции

function MyFunction (out template boolean MyReferenceParameter) { … };
// MyReferenceParameter передается по ссылке. Реальный параметр может
// только устанавливаться из этой функции
```

5.2.1.2 Параметры, передаваемые по значению

Реальные параметры, которые передаются по значению, могут быть переменными, а также константами, шаблонами и т. д.

```
function MyFunction (in template MyTemplateType MyValueParameter) {  };
// MyValueParameter передается по значению, причем ключевое слово in необязательно
```

5.2.2 Списки формальных и реальных параметров

Число элементов и порядок их следования в списке реальных параметров должны быть такими же, как число элементов и порядок их следования в соответствующем списке формальных параметров. Кроме того, тип каждого реального параметра должен быть совместимым с типом каждого соответствующего формального параметра.

ПРИМЕР:

```
// Определение функции со списком формальных параметров
function MyFunction (integer FormalPar1, boolean FormalPar2, bitstring FormalPar3) { … }

// Вызов функции со списком реальных параметров
MyFunction (123, true, '1100'B);
```

5.2.3 Пустой список формальных параметров

Если пустым является список формальных параметров для элементов языка TTCN-3 **function**, **testcase**, **signature**, **altstep** или функции **external**, то пустые круглые скобки должны быть включены как в объявление, так и в вызов этого элемента. Во всех остальных случаях пустые круглые скобки опускаются.

ПРИМЕР:

```
// Определение функции с пустым списком параметров должно записываться как
function MyFunction ( ) { … }

// Определение записи с пустым списком параметров должно записываться как
type record MyRecord { … }
```

5.2.4 Вложенные списки параметров

Собственные параметры всех параметризованных объектов, указанных в виде реальных параметров, должны быть разрешены в списке реальных параметров верхнего уровня.

ПРИМЕР:

```
// Заданное определение сообщения
type record MyMessageType
{
  integer   field1,
  charstring   field2,
  boolean   field3
}

// Шаблоном сообщения может быть
template MyMessageType MyTemplate(integer MyValue) :=
```

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

// Тестовым примером, параметризованным с шаблоном, может быть
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{  :
   MyPCO.receive(RxMsg);
}

// Когда тестовый пример вызывается в управляющей части, а параметризованный
// шаблон используется в качестве реального параметра, должны
// обеспечиваться реальные параметры для шаблона
   control
{  :
   execute( TC001(MyTemplate(7)) );
   :
}
```

5.2.5 Формальные параметры типа шаблон

5.2.5.1 Параметризация с шаблонами и согласующими атрибутами

Для того чтобы разрешить передачу шаблонов или согласующих атрибутов в виде реальных параметров, перед полем типа соответствующего формального параметра необходимо добавить ключевое слово **template**. Благодаря этому параметр принимает тип шаблона и реально расширяет список разрешенных параметров для связанного типа, позволяя включить соответствующий набор согласующих атрибутов (см. Приложение B) вместе с нормальным набором значений. Не разрешается вызывать по ссылке поля шаблонов параметров.

ПРИМЕР:

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

// может использоваться следующим образом
pcol.receive(MyTemplate(?));
// или же так:
pcol.receive(MyTemplate(omit));
```

5.2.5.2 Элементы языка, использующие параметры типа шаблон

Только определения **function, testcase, altstep** и **template** могут иметь формальные параметры типа шаблон.

ПРИМЕР:

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

5.3 Контекстные правила

5.3.0 Общие положения

TTCN-3 обеспечивает семь базовых единиц контекста (областей применения):

a) определяющая часть модуля;

b) управляющая часть модуля;

c) типы компонента;

d) функции;

e) altstep;

f) тестовые примеры;

g) "блоки выражений и объявлений" в пределах сложных выражений.

ПРИМЕЧАНИЕ 1. – Дополнительные контекстные правила для групп приводятся в 7.3.1.

ПРИМЕЧАНИЕ 2. – Дополнительные контекстные правила для счетчиков приводятся в 19.7.

Каждая единица контекста состоит из (факультативных) объявлений Единицы контекста: управляющая часть модуля, функции, тестовые примеры, altstep и "блоки выражений и объявлений" внутри сложных выражений могут дополнительно специфицировать некоторую форму поведения с помощью использования программных команд и операторов TTCN-3 (см. раздел 18).

Определения, которые сделаны в части определений модуля, однако вне контекстных единиц, обладают глобальной видимостью, то есть могут использоваться во всем модуле, включая все функции, тестовые примеры и altstep, которые определены в пределах модуля и его управляющей части. Идентификаторы, импортированные из других модулей, также обладают глобальной видимостью во всем импортирующем модуле.

Определения, которые сделаны в управляющей части модуля, обладают локальной видимостью, то есть могут использоваться только в пределах управляющей части.

Определения, которые сделаны в типе тестового компонента, могут использоваться только в функциях, тестовых примерах и altstep, которые ссылаются на этот тип компонента или на совместимый (consistent) тип тестового компонента (см. 16.3) при помощи оператора **`runs on`**.

Тестовые примеры, altstep и функции являются индивидуальными контекстными единицами, между которыми не существует никаких иерархических отношений, то есть, объявления, которые сделаны в начале тела этой единицы, имеют локальную видимость и должны использоваться только в данном тестовом примере, altstep или функции (например, объявление, сделанное в тестовом примере, не является видимым в функции, которая вызывается данным тестовым примером или в altstep, который использует данный тестовый пример).

Сложные выражения, например **`if-else-`**, **`while-`**, **`do-while-`** или выражения **`alt`**, включают в себя "блоки выражений и объявлений". Они могут использоваться в управляющей части модуля, тестовых примерах, altstep, функциях, или же могут встраиваться в другие сложные выражения, напримепр, в выражение **`if-else`**, которое используется внутри цикла **`while`**.

Для сложных выражений и встроенных сложных выражений "блоки выражений и объявлений" имеют иерархическое отношение как относительно контекстной единицы, которая включает данный "блок выражений и объявлений", так и относительно любого встроенного "блока выражений и объявлений". Объявления, которые сделаны внутри "блока выражений и объявлений", имеют локальную видимость.

Иерархия контекстных единиц показана на Рисунке 2. Объявления контекстных единиц на более высоком иерархическом уровне являются видимыми для всех единиц на более низких уровней в данной ветви иерархии. Объявления контекстных единицу из нижних уровней иерархии являются невидимыми для тех единиц, которые относятся к более высокому иерархическому уровню.

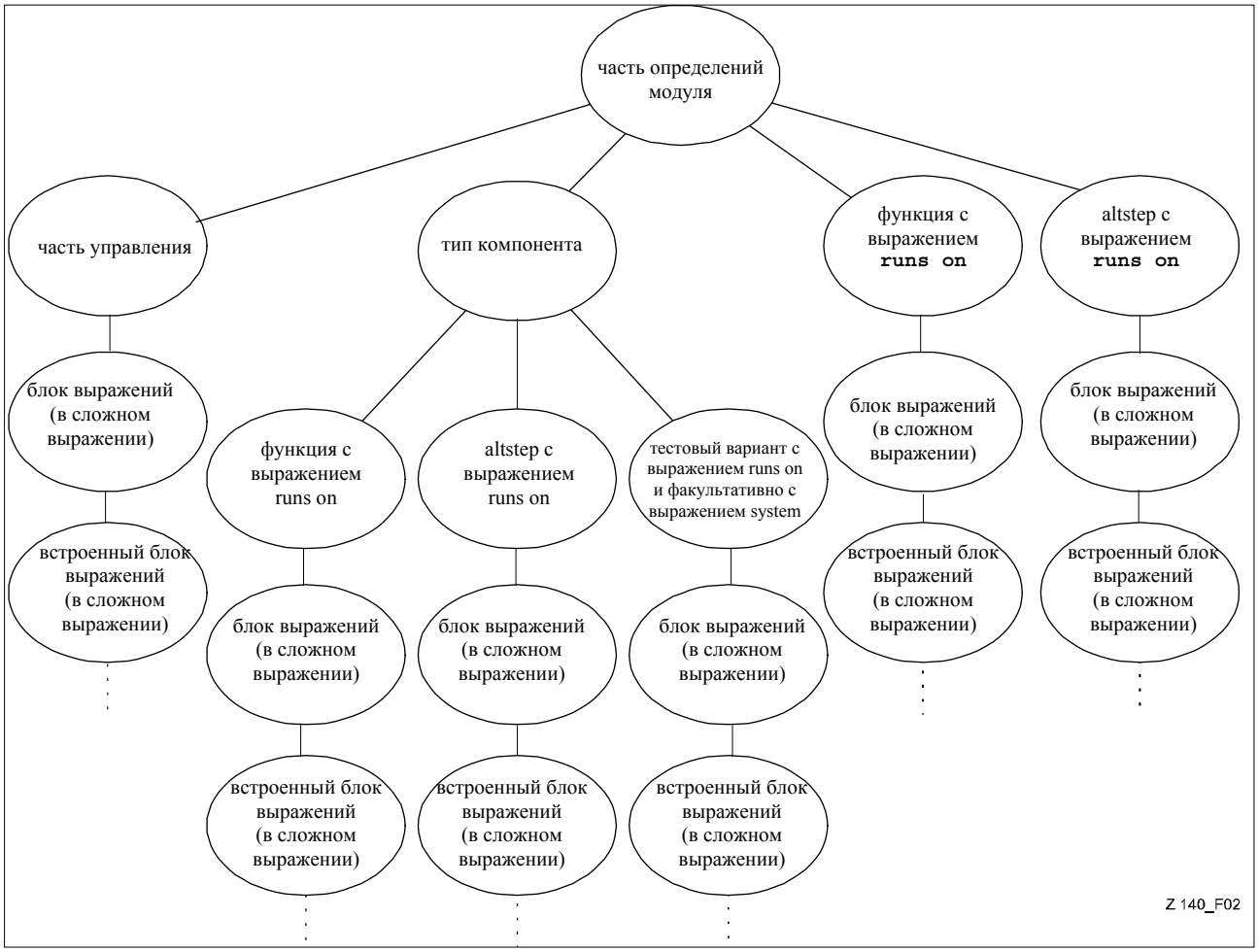


Рисунок 2/Z.140 – Иерархия контекстных единиц

ПРИМЕР:

```
module MyModule
{  :
   const integer MyConst := 0;//MyConst виден для MyBehaviourA и MyBehaviourB
   :
   function MyBehaviourA()
   {   :
       const integer A := 1;   // Константа A видна только для MyBehaviourA
       :
   }

   function MyBehaviourB()
   {   :
       const integer B := 1;   // Константа B видна только для MyBehaviourB
       :
   }
}
```

5.3.1 Контекст для формальных параметров

Контекст для формальных параметров в параметризованных элементах языка (например, в вызове функции) должен ограничиваться определением, в котором появляется параметр, и нижними уровнями контекста в той же иерархии контекста. Таким образом, они соответствуют обычным правилам контекста (см. 5.3.0).

5.3.2 Уникальность идентификаторов

TTCN-3 требует уникальности идентификаторов, то есть все идентификаторы в одной иерархии контекста должны быть различными. Это означает, что объявление на нижнем уровне контекста не должно повторно использовать идентификатор, который был использован в объявлении на более высоком уровне контекста в одной и той же ветви иерархии контекста. Идентификаторы для полей структурированного типа, перечисляемые значения и группы не обязаны иметь глобальную уникальность; однако в случае перечисляемых значений идентификаторы должны повторно использоваться только для перечисляемых значений внутри других типов перечисления. Правила уникальности идентификатора также должно применяться к идентификаторам формальных параметров.

ПРИМЕР:

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

   // Нижеследующее РАЗРЕШЕНО, поскольку константы не объявлены в той же
   // иерархии контекста (принимая, что в заголовке модуля нет объявления A)
function MyBehaviourA()
{  :
   const integer A := 1;
   :
}

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

5.4 Идентификаторы и ключевые слова

Идентификаторы TTCN-3 чувствительны к регистру клавиатуры, а ключевые слова TTCN-3 должны записываться только строчными буквами (см. Приложение A). Ключевые слова TTCN-3 не должны использоваться в качестве идентификаторов ни для объектов TTCN-3, ни в качестве идентификаторов для объектов, импортированных из модулей на других языках.

6 Типы и значения

6.0 Общие положения

TTCN-3 поддерживает ряд предопределенных базовых типов. В их число входят типы, которые обычно ассоциируются с языками программирования, например **integer, boolean** и **string**, а также некоторые специфические для TTCN-3 типы, например **verdicttype**. Структурированные типы, такие как **record**, **set** и **enumerated**, могут конструироваться из этих базовых типов.

Специальный тип данных **anytype** определяется как союз всех известных типов данных и типа адрес в пределах модуля.

Специальные типы, связанные с тестовыми конфигурациями, такие как **address, port** и **component**, могут использоваться для определения архитектуры тестовой системы (см. раздел 22).

Специальный тип **default** может использоваться для обработки по умолчанию (см. раздел 21).

Типы для TTCN-3 сведены в таблицу 3.

Таблица 3/Z.140 – Обзор типов для TTCN-3

Класс типов	Ключевое слово	Подтипы
Базовые типы	**integer**	range, list
	float	range, list
	boolean	list
	objid	list
	verdicttype	list
Базовые цепочечные типы	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	range, list, length, pattern
	universal charstring	range, list, length, pattern
Структурированные типы	**record**	list (см. Примечание)
	record of	list (см. Примечание), length
	set	list (см. Примечание)
	set of	list (см. Примечание), length
	enumerated	list (см. Примечание)
	union	list (см. Примечание)
Специальные типы данных	**anytype**	list (см. Примечание)
Специальные конфигурационные типы	**address**	
	port	
	component	
Специальные типы по умолчанию	**default**	
ПРИМЕЧАНИЕ. – Использование в качестве подтипа **list** для данных типов возможно тогда, когда определяется новый тип с ограничениями из существующего родительского типа, однако это происходит не непосредственно в декларации первого родительского типа.		

6.1 Базовые типы и значения

6.1.0 Простые базовые типы и значения

TTCN-3 поддерживает следующие базовые типы:

a) **integer** (целочисленный): тип, который различает значения, являющиеся положительными и отрицательными целыми числами, включая нуль.

Значения типа **integer** обозначаются одной или несколькими цифрами; первая цифра не должна быть нулем, за исключением случая, когда значение равно 0; значение НУЛЬ должно представляться одиночным нулем.

b) **float** (плавающий): тип для описания чисел с плавающей запятой ("с плавающей точкой" в английских текстах).

В общем случае, числа с плавающей запятой могут быть определены следующим образом: <*мантисса*>×<*основание*> $^{<показатель\ степени>}$, здесь <*мантисса*> является положительным или отрицательным целым числом, <*основание*> – положительным целым числом (обычно 2, 10 или 16), а <*показатель степени*> – положительным или отрицательным целым числом.

В TTCN-3 запись значения для чисел с плавающей запятой ограничена основанием со значением 10. Значения с плавающей запятой (точкой) могут выражаться разными способами:

– десятичным обозначением с точкой в последовательности цифр, как, например, 1.23 (что представляет 123×10^{-2}), 2.783 (то есть 2783×10^{-3}), –123.456789 (что представляет $-123\ 456\ 789\times10^{-6}$); либо

– двумя числами, разделенными буквой E, где первое число указывает мантиссу, а второе – показатель степени, например, 12.3E4 (что означает 123×10^{3}) или –12.3E-4 (что означает -123×10^{-5}).

ПРИМЕЧАНИЕ. – В отличие от общего определения значений с плавающей точкой, обозначение мантиссы в TTCN-3 может принимать не только целые, но и десятичные значения.

c) **boolean** (булев): тип, состоящий из двух различающихся значений.

Значения булева типа обозначаются символами **true** (ИСТИНА) и **false** (ЛОЖЬ).

d) Утратило значение.

e) **verdicttype** (тип "вердикт"): тип для использования с вердиктами тестов, имеющими 4 различающихся значения. Значения **verdicttype** выражаются с помощью **pass, fail, inconc, none** и **error**.

6.1.1 Базовые типы "строка" и их значения

TTCN-3 поддерживает следующие базовые типы "цепочка" (строка):

ПРИМЕЧАНИЕ 1. – Общий термин "цепочка" (строка) или "цепочечный тип" в TTCN-3 относится к **bitstring, hexstring, octetstring, charstring** и **universal charstring**.

a) **bitstring** (цепочка битов): тип, различаемыми значениями которого являются упорядоченные последовательности, содержащие нуль, один или более битов.

Значения типа **bitstring** выражаются произвольным числом (возможно, нулем) битовых цифр: 0 и 1, которым предшествует одинарная кавычка (') и за которыми следует пара знаков 'B.:

ПРИМЕР 1: `'01101'B.`

b) **hexstring** (шестнадцатеричная цепочка): тип, различаемыми значениями которого являются упорядоченные последовательности, содержащие нуль, одну или более шестнадцатеричных цифр, каждая из которых соответствует упорядоченной последовательности из четырех битов.

Значения типа **hexstring** выражаются произвольным числом (возможно, нулем) шестнадцатеричных цифр (в качестве которых могут использоваться символы как верхнего, так и нижнего регистра):

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

которым предшествует одинарная кавычка (') и за которыми следует пара знаков 'H; каждая шестнадцатеричная цифра применяется для выражения значения полуоктета, используя шестнадцатеричное представление.

ПРИМЕР 2:
```
'AB01D'H
'ab01d'H
'Ab01D'H
```

c) **octetstring** (цепочка октетов): тип, различаемыми значениями которого являются упорядоченные последовательности, содержащие нуль или положительное четное число шестнадцатеричных цифр (каждая пара цифр соответствует упорядоченной последовательности из восьми битов).

Значения типа **octetstring** выражаются произвольным, но четным числом (возможно, нулем) шестнадцатеричных цифр (в качестве которых могут использоваться символы как верхнего, так и нижнего регистра):

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

которым предшествует одинарная кавычка (') и за которыми следует пара знаков 'O; каждая шестнадцатеричная цифра применяется для выражения значения полуоктета, с использованием шестнадцатеричного представления.

ПРИМЕР 3:
```
'FF96'O
'ff96'O
'Ff96'O
```

d) **charstring** (цепочка знаков): это типы, различаемыми значениями которых являются нуль (знаков), один или более знаков из версии Рекомендации МСЭ-Т T.50 [9], соответствующим International Reference Version (IRV), как это специфицируется в 8.2/T.50 [9].

ПРИМЕЧАНИЕ 2. – Версия IRV Международного эталонного алфавита (International Reference Alphabet, ранее – International Alphabet No. 5 – IA5), которая описывается в Рекомендации МСЭ-Т T.50, эквивалентна IRV-версии МСЭ/МЭК 646.

Значения типа **charstring** выражаются произвольным числом (возможно, нулем) знаков из соответствующих наборов знаков; перед этими знаками и за ними располагаются двойные кавычки (") или же вычисленных с использованием предопределенной функции преобразования int2char, которая в качестве аргумента использует положительное целое число, соответствующее кодировке символа (см. C.1).

ПРИМЕЧАНИЕ 3. – Предопределенная функция преобразования может возвращать только значения с длиной, равной единственному символу.

В случаях, когда необходимо определять цепочки, содержащие знак "двойные кавычки" ("), этот знак представляется парой двойных кавычек на одной и той же строке без введения знаков пробела.

ПРИМЕР 4: `"abcd"` представляет буквенную цепочку `"abcd"`.

e) Тип цепочки символов, перед которым ставиться ключевое слово **universal**, обозначает типы, различаемыми значениями которых являются ноль, один или более знаков из стандарта ИСО/МЭК 10646-1 [10]. Значения типа **universal charstring** могут обозначаться произвольным количеством (возможно, равным нулю) знаков из соответствующего набора символов, которые заключены в двойные кавычки ("), которые вычисляются с помощью предопределенной функции преобразования (см. C.3) с положительным целочисленным значением ее кодирования в качестве аргумента или с помощью "тетрады".

ПРИМЕЧАНИЕ 4. – Предопределенная функция преобразования может возвращать только значения с длиной, равной единственному символу.

В тех случаях, когда необходимо определить строки, которые включают в себя символ двойной кавычки ("), то такой символ представляется в виде пары двойных кавычек, расположенных в одной строке и между ними не должно стоять пробелов.

"Тетрада" может обозначать только единственный символ, символ обозначается с помощью десятичных значений, указывающих группу, плоскость, ряд и ячейку согласно ИСО/МЭК 10646 [10], которые заключаются в круглые скобки, разделяются запятыми и перед ними располагается ключевое слово **char** (например, **char** (0, 0, 1, 113) обозначает используемый в венгерском языке символ `"ű"`). В том случае, когда в строке, для которой значение было присвоено по первому методу (внутри двойных кавычек), необходимо обозначить символ двойных кавычек ("), то этот символ обозначается как пара символов двойных кавычек, которые располагаются на одной строке и между которыми отсутствуют пробелы. Данные два метода могут использоваться в одном обозначении при задании значения строки с помощью применения оператора конкатенации.

ПРИМЕР 5: Выражение: "the Braille character" & **char** (0, 0, 40, 48) & "looks like this" представляет собой следующую строку символов: the Braille character ⣿ looks like this.

ПРИМЕЧАНИЕ 5. – Управляющие символы могут обозначаться при помощи предопределенной функции преобразования или же при помощи формы тетрады.

По умолчанию **universal charstring** должен соответствовать форме кодированного представления UCS-4, которая специфицируется в разделе 14.2 ИСО/МЭК 10646 [10].

ПРИМЕЧАНИЕ 6. – UCS-4 представляет собой формат кодирования, которые представляет символы UCS в виде фиксированных полей длиной 32 бита.

Данная кодировка по умолчанию могут быть изменена с помощью определенных атрибутов вариантов (см. 28.2.3). В Приложении E определяются следующие полезные типы строк символов, которые используют такие атрибуты: utf8string, bmpstring, utf16string и iso8859.

6.1.2 Доступ к отдельным элементам цепочки

Отдельные элементы в цепочечном типе могут быть доступны с помощью синтаксиса типа массива. Могут быть доступны только одиночные элементы цепочки.

Единицы длины для элементов разных цепочечных типов указаны в таблице 4.

Индексация должна начинаться со значения "нуль" (0).

ПРИМЕР:

```
// Дано
MyBitString : = '11110111'B;
// Затем делаем
MyBitString [4] : = '1'B;
// Получаем цепочку битов '11111111'B
```

6.2 Подтипы базовых типов

6.2.0 Общие положения

Определяемые пользователем типы обозначаются ключевым словом **type**. С помощью определяемых пользователем типов можно создавать подтипы (такие как списки, диапазоны и ограничения длины) базовых типов, структурированных типов и **anytype** в соответствии с Таблицей 3.

6.2.1 Списки значений

TTCN-3 позволяет определять список различаемых значений для базовых типов, структурированных типов и **anytype**, как это приводится в Таблице 3. Значения в таком списке должны быть корневого типа и должны представлять собой правильное подмножество значений, определенных для рассматриваемого корневого типа. Подтип, определенный этим списком, ограничивает разрешенные значения данного подтипа значениями из этого списка.

ПРИМЕР:

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

6.2.2 Диапазоны

6.2.2.0 Общие положения

TTCN-3 позволяет определять диапазон ограничений для типов **integer, charstring, universal charstring** и **float** (или производных от этих типов). Для **integer** и **float** подтип, определенный диапазоном, ограничивает разрешенные значения данного подтипа значениями этого диапазона, включая нижнюю и верхнюю границы. В случае типов **charstring** и **universal charstring** диапазон ограничивает разрешенные значения для каждого символа в данных строках. Границы должны оцениваться как действительные позиции символов в соответствии с таблицей (таблицами) кодирования символов для данного типа (например, данная позиция не должна являться пустой). Пустые позиции между нижней и верхней границами не считаются действительными значениями для указанного диапазона.

ПРИМЕР 1:

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

ПРИМЕР 2:

```
type charstring MyCharString ("a" .. "z");
// Определяет тип строки произвольной длины, в которой каждый символ
// относится к указанному диапазону
type universal charstring MyUCharString1 ("a" .. "z");
// Определяет тип строки произвольной длины, в которой каждый символ
// относится к диапазону от a до z (символы с кодами от 97 до 122), например
// "abxyz";
// строки, содержащие любой другой символ (включая упрапвляющие символы),
// например "abc2", запрещены.
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// Определяет тип строки произвольной длины, в которой каждый символ
// относится к диапазону, заданному с помощью обозначения в виде тетрады
```

6.2.2.1 Неограниченные диапазоны

Чтобы определить неограниченный диапазон целых чисел или чисел в формате с плавающей запятой, можно использовать ключевое слово **infinity** вместо значения, указывающего на отсутствие нижней или верхней границы. Верхняя граница должна быть больше нижней границы или равна ей.

ПРИМЕР:

```
type integer MyIntegerRange (-infinity .. –1); //  Все отрицательные целые  числа
```

ПРИМЕЧАНИЕ. – Значение для **infinity** зависит от реализации. Использование этого свойства может привести к проблемам совместимости.

6.2.2.2 Смешанные списки и диапазоны

Данный раздел не действует.

ПРИМЕЧАНИЕ. – Данный раздел заменен на раздел 6.2.5.

6.2.3 Ограничения длины цепочки

TTCN-3 позволяет определять ограничения длины для цепочечных типов. Границы длины основываются на различных единицах в зависимости от типа цепочки, с которым они используются. Во всех случаях эти границы определяются неотрицательными значениями **integer** (или производными значениями **integer**).

ПРИМЕР:

```
type bitstring MyByte length(8);                // Длина точно 8
type bitstring MyByte length(8 .. 8);           // Длина точно 8
type bitstring MyNibbleToByte length(4 .. 8);   // Минимальная длина 4, максимальная длина 8
```

В таблице 4 указаны единицы длины для разных цепочечных типов.

Таблица 4/Z.140 – Единицы длины, которые используются при спецификации длины поля

Тип	Единицы длины
bitstring	биты
hexstring	шестнадцатеричные цифры
octetstring	октеты
character strings	символы

Для верхней границы может также использоваться ключевое слово **infinity** для указания, что верхний предел длины отсутствует: верхняя граница должна быть больше нижней границы или равна ей.

6.2.4 Pattern-подтипы для типов символьных строк

TTCN-3 разрешает использование определенных в B.1.5 символьных образцов с целью ограничения разрешенных значений для типов **charstring** и **universal charstring**. Такое ограничения для типа использует ключевое слово **pattern**, за котором следует образец символов. Все указанные в образце знаечния должны относиться к правильному подмножеству типа, для которого определяется подтип.

ПРИМЕЧАНИЕ. – Определение подтипа с помощью образца может рассматриваться как специальная форма ограничения для списка, при которой члены списка определяются не с помощью перечисления специальных символьных строк, а с помощью механизма, который создает элементы для списка.

ПРИМЕР:

```
type charstring MyString (pattern "abc*xyz");
  // для MyString все разрешенные значения имеют префикс abc и постфикс xyz

type universal charstring MyUString (pattern "*\r\n")
  // для MyUString все разрешенные значения оканчиваются на CR/LF

type charstring MyString2 (pattern "abc?\q{0,0,1,113}");
   // приводит к ошибке, так как символ, обозначенный тетрадой (0,0,1,113}, не
   // являются разрешенными символами для типа charstring в TTCN-3

type MyString MyString3 (pattern "d*xyz");
   // приводит к ошибке, так как тип MyString не содержит значения,
   // которое начинается с символа d
```

6.2.5 Совместное использование механизмов создания подтипов

6.2.5.1 Совместное использование образцов, списков и диапазонов

В определениях подтипов для **integer** и **float** (или производными значениями для этих типов) разрешается совместно использовать списки и диапазоны. Пересечение различных ограничений не является ошибкой.

ПРИМЕР 1:

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

В определениях подтипов для **charstring** и **universal charstring** не разрешается совместно использовать ограничения с помощью образца, списка или диапазона.

ПРИМЕР 2:

```
type charstring MyCharStr0 ('gr', 'xyz');
  // содержит символьные строки gr и xyz;

type charstring MyCharStr1 ('a'..'z');
  // содержит символьные строки произвольной длины, содержащие символы от a до // z.

type charstring MyCharStr2 (pattern '[a-z]#(3,9)');
  // содержит символьные строки длиной от 3 до 9 символов, которые содержат
  // символы от a до z
```

6.2.5.2 Использование ограничения длины совместно с другими ограничениями

В пределах определений подтипа для **bitstring, hexstring, octetstring** разрешается совместное использование списков и ограничений длины в одном и том же определении подтипа.

В пределах подтипа для **bitstring, hexstring, octetstring** разрешается добавлять ограничения длины к ограничениям, которые содержат список, диапазон или образец в одном и том же определении подтипа.

При совместном использовании с другими ограничениями ограничение длины должно являться последним элементом в определении подтипа. Ограничение длины оказывает эффект совместно с другими механизмами создания подтипов (то есть набор значений для типа состоит из общего подмножества наборов значений, которые определяются ограничениями для подтипа с помощью списка, диапазона или образца, а также ограничением длины).

ПРИМЕР:

```
type charstring MyCharStr5 ('gr', 'xyz') length (1..9);
  // содержит символьные строки gr и xyz;

type charstring MyCharStr6 ( 'a'..'z') length (3..9);
  // содержит символьные строки длиной от 3 до 9 символов, содержащие
  // символы от a до z

type charstring MyCharStr7 (pattern '[a-z]#(3,9)') length (1..9);
  // содержит символьные строки длиной от 3 до 9 символов, содержащие
  // символы от a до z

type charstring MyCharStr8 (pattern '[a-z]#(3,9)') length (1..8);
  // содержит символьные строки длиной 3 до 8 символов, содержащие символы
  // от a до z

type charstring MyCharStr9 (pattern '[a-z]#(1,8)') length (1..9);
  // содержит любые символьные строки длиной от 1 до 8 символов, содержащие
  // символы от a до z

type charstring MyCharStr10 ('gr', 'xyz') length (4);
  // не содержит значений (тип empty).
```

6.3 Структурированные типы и значения

6.3.0 Общие положения

Ключевое слово **type** используется также для определения структурированных типов, таких, как **record, record of, set, set of, enumerated** и **union**.

Значения этих типов могут выражаться с помощью нотации явного присвоения или некоторогократкого обозначения списка значений.

ПРИМЕР 1:

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

// или
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
// обозначение списка значений
```

Когда при помощи обозначения присваивания указываются частичные значения (то есть присваивается значение только некоторому подмножеству полей, относящихся к структурированной переменной), то следует указывать только те поля, которым присваиваются значения. Поля, которые не указываются, автоматически остаются без изменений. Также можно в явном виде указать поля, которым не присваиваются значения, с помощью символа "-". При использовании обозначения списка значений все поля структуры должны быть указаны либо с присвоением значения, либо неиспользуемые поля указываются с помощью символа "-" или ключевого слова **omit**.

ПРИМЕР 2:

```
var MyRecordType MyVariable:=           // обозначение присваивания
{
  field1 := '11001'B,
  // поле field2 явным образом не указывается
  field3 := "A string"
}

// или
var MyRecordType MyVariable:=       // обозначение присваивания
{
  field1 := '11001'B,
  field2 := -, // поле field2 указывается явным образом
  field3 := "A string"
}

// или
var MyRecordType MyVariable:= {'11001'B, -, "A string"}
// обозначение с помощью списка значений
```

Не разрешается смешивать две нотации значений в одном (ближайшем) контексте.

ПРИМЕР 3:

```
// Так не разрешается
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "A string"}
```

Как в обозначении присваивания, так и в обозначении списка значений факультативные поля могут опускаться с помощью использования явного значения **omit** для соответствующего поля. Ключевое слово **omit** не должно использоваться для обязательных полей. При повторном присваивании для уже инициализированных значений использование неиспользуемого символа или пропуск поля в обозначении присваивания приведет к тому, что соответствующие поля останутся неизменными.

ПРИМЕР 4:

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

MyVariable := { '10111'B, -, - };
// после этого MyVariable содержит { '10111'B, false /* не изменилось */, <undefined> }

MyVariable :=
{
  field2 := true
}
// после этого MyVariable содержит { '10111'B, true, <undefined> }

MyVariable :=
{
  field1 := -,
  field2 := false,
  field3 := -
}
// после этого MyVariable содержит { '10111'B, false, <undefined> }
```

6.3.1 Тип "запись" и его значения

6.3.1.0 Общие положения

TTCN-3 поддерживает упорядоченные структурированные типы, известные как **record** (запись). Элементами типа "запись" могут быть любые базовые типы или определяемые пользователем типы данных (такие как другие записи, множества или массивы). Значения записи должны быть совместимы с типами полей записи. К записи добавляются местные идентификаторы элементов, которые должны быть уникальными в пределах записи (однако не должны быть уникальными глобально). Константа, являющаяся типом записи, не должна содержать переменных (или параметров модуля) в качестве значений поля ни прямо, ни косвенно.

ПРИМЕР 1:

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

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

Записи могут определяться без полей (то есть в качестве пустых записей).

ПРИМЕР 2:

```
type record MyEmptyRecord {}
```

Значение записи присваивается отдельно для каждого элемента. Порядок значений полей в обозначении списка значений должен быть таким же, как и порядок полей в соответствующем определении типа.

ПРИМЕР 3:

```
var integer MyIntegerValue:= 1;

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

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

То же значение, заданное с помощью списка значений.

ПРИМЕР 4:

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

6.3.1.1 Ссылки на поля для типа записи

На элементы записей ссылки делаются с помощью обозначения точкой *TypeOrValueId.ElementId*, где *TypeOrValueId* служит для определения имени структурированного типа или переменной. ElementId служит для определения имени поля для структурированного типа.

ПРИМЕР:

```
MyVar1 := MyRecord1.myElement1;
// Если запись является вложенной в другой тип, то ссылка может быть такой
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 Факультативные элементы в записи

Факультативные элементы в record указываются с помощью ключевого слова optional.

ПРИМЕР 1:

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

Факультативные поля должны быть опущены при помощи символа **omit**.

ПРИМЕР 2:

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

// Заметим, что это не эквивалентно записи
// MyRecordValue:= {MyIntegerValue, -, "A string"};
// которая означает, что поле field2 остается неизменным
```

6.3.1.3 Вложенные определения для типа поле

TTCN-3 поддерживает определения типов для полей записей, вложенных в определение **record**. Возможно использовать как определение новых структурированных типов (**record**, **set**, **enumerated**, **set of** и **record of**), так и спецификация ограничений для подтипов.

ПРИМЕР:

```
// тип записи с вложенными определениями структурированного типа
type record MyNestedRecordType
{
   record
   {
       integer nestedField1,
       float   nestedField2
} outerField1,
enumerated {
       nestedEnum1,
```

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

// тип записи с вложенными определениями подтипа
type record MyRecordTypeWithSubtypedFields
{
  integer    field1 (1 .. 100),
  charstring field2 length ( 2 .. 255 )
}
```

6.3.2 Тип "множество" и его значения

6.3.2.0 Общие положения

TTCN-3 поддерживает неупорядоченные структурированные типы, известные как **set** (множество). Типы и значения "множество" аналогичны записям, за исключением того, что порядок следования полей множества не имеет значения.

ПРИМЕР:

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

Идентификаторы полей являются локальными по отношению ко множеству и должны быть уникальными в пределах множества (однако не должна являться уникальными глобально)

Нотация списка значений не должна использоваться для значений типов **set**.

6.3.2.1 Ссылки на поля для типа set

Ссылки на элементы set осуществляются с помощью нотации точкой (см. 6.3.1.1).

ПРИМЕР:

```
MyVar3 := MySet1.myElement1;
// если множество является вложенным в другой тип, то ссылка
// может выглядеть следующим образом
MyVar4 := MyRecord1.myElement1.myElement2;
// заметим, что тип set, в котором мы ссылаемся на идентификатор 'myElement2',// вложен в тип record
```

6.3.2.1 Факультативные элементы в "множестве"

Факультативные элементы в **set** указываются с помощью ключевого слова **optional**.

6.3.2.3 Определение вложенного типа для типов поля

TTCN-3 поддерживает определение топов для полей множеств, вложенных в определение set, подобно механизу, который описывался в 6.3.1.3 для типа записи.

6.3.3 Записи и множества одних и тех же типов

6.3.3.0 Общие положения

TTCN-3 поддерживает определение записей и множеств, все элементы которых принадлежат к одному и тому же типу. Это указывается с помощью ключевого слова **of**. Такие записи и множества не имеют идентификаторов элементов и могут рассматриваться как упорядоченный массив и неупорядоченный массив соответственно.

Ключевое слово **length** используется для ограничения длин типов **record of** и **set of**.

ПРИМЕР 1:

```
type record length(10) integer MyRecordOfType; // Это запись максимум в
// точности из 10 целых чисел
type record length(0..10) of integer MyRecordOfType; // Это запись, состоящая
// максимум из 10 целых чисел
type set of boolean MySetOfType;    // Это неограниченное множество
// булевых значений
type record of length(0..10) of charstring StringArray length(12);
// Это запись максимум из 10 цепочек, каждая с максимальной длиной 12 знаков
```

Нотацией значений для **record of** и **set of** должна являться нотация списка значений или индексированная нотация для отдельного элемента (такая же нотация значений, которая используется для массивов, см. п. 6.5). Существует единственное исключение из этого общего правила: в случае определения модифицированных шаблонов, где также разрешается использование нотации присваивания (см. 14.6.0).

При использовании нотации списка значений первое значение в списке присваивается первому элементу, второе значение в списке присваивается второму элементу и так далее. Пустые присваивания не разрешаются (то есть не разрешается использовать две запятые, вторая из которых следует непосредственно за первой или же их разделяет только пробел). Элементы, которые не участвуют в присваивании, должны быть пропущены явным образом или же опущены в списке.

Нотации с индексированными значениями могут использоваться как в правой, так и в левой стороне выражения присваивания. Индекс первого элемента равняется нулю, значение индекса не должно превышать ограничения, наложенного на длину при определении подтипа. Если значение элемента, который определяется индексом, не определено в правой части приваивания, то это приводит к семантической ошибке или к ошибке времени выполнения. Если оператор индексирования в левой стороне выражения ссылается на несуществующий элемент, то значение в правой стороне присваивания присваивается элементу и всем другим элементам, у которых индекс имеет меньшее значение по сравнению с действительным индексом, а если значение не присваивается, то создается с неопределенным значением. Неопределенные элементы разрешаются только в переходных состояниях (когда значение является невидимым). Если отправить значение **record of** с неопределенными элементами, то это приведет к динамической ошибке для тестового варианта.

ПРИМЕР 2:

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

MyVar := MyRecordVar[0]; // значение первого элемента "record of" (integer 0)
                         // присваивается MyVar

// Индексированные значения могут использоваться также и в левой стороне присваивания:
MyRecordVar[1] := MyVar; // значение MyVar присваивается второму элементу
                         // значение MyRecordVar равно { 0, 0, 2, 3, 4 }

// Присаивание
MyRecordVar := { 0, 1, -, 2, omit };
// изменяет значение MyRecordVar на { 0, 1, 2 <unchanged>, 2};
// Заметим, что 3-йгэлемент будет неопределен если ему ранее не было присвоено // значение.

// Присваивание
  MyRecordVar[6] := 6;

// изменяет значение MyRecordVar на { 0, 1, 2 , 2, <undefined>,   <undefined>, 6 };
// Заметим, то элементы 5 и 6 (с индексами 4 и 5) перед последним
// присваиванием не имели присвоенных значений и являются неопределенными.
```

ПРИМЕЧАНИЕ. – Это позволяет копировать в цикле значения элементов **record of**. Например, показанная ниже функция изменяет порядок значений элементов **record of**:

```
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;
     }
```

Вложенные типы **record of** и **set of** приводят к созданию структур данных, которые подобны массивам с многими измерениями (см. 6.5).

ПРИМЕР 3:

```
// дано:
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType  MyRecordOfType;

// Тогда переменная myRecordOfArray будет иметь атрибуты, подобные двумерному // массиву:
var MyRecordOfType myRecordOfArray;
// а ссылка на отдельный элемент будет выглядеть следующим образом
// (значение второго элемента третьей конструкции 'MyBasicRecordOfType')
myRecordOfArray [2][1] := 1;
```

6.3.3.1 Вложенные определения типа

TTCN-3 поддерживает определение агрегированных типов, вложенных в определение **record of** или **set of**. Возможно определить как новые структурированные типы (**record**, **set**, **enumerated**, **set of** или **record of**)., так и спецификацию ограничений подтипа.

ПРИМЕР:

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

6.3.4 Перечислительный тип и его значения

TTCN-3 поддерживает типы **enumerated**. Перечислительные типы используются для построения типов, которые требуют только именованного множества значений, различающихся между собой. Такие различающиеся между собой значения носят название перечислений. Каждое перечисление должно иметь собственный идентификатор. В операциях с перечислительными типами должны использоваться только такие идентификаторы и только операторы присвоения, эквивалентности и упорядочения. Идентификаторы перечислений должны являться уникальными в пределах перечислительного типа (однако не должны являться глобально уникальными) и, следовательно, являются видимыми только в пределах контекста данного типа. Идентфкаторы перечисления должны повторно использоваться в пределах других определений структурированных типови не должны использоваться для идентификаторов, имеющих локальную или глобальную видимость на том же или более низком уровне той же ветви в контекстной иерархии (см. контекстная иерархия в 5.3.0).

ПРИМЕР 1:

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

type integer Monday;
// Данное определение не действительно, так как название типа имеет
// локальную или глобальную видимость

type enumerated MySecondEnumType {
  Saturday, Sunday, Monday
};
// Это определение является правильным – оно повторно использует идентификатор
// перечисления Monday в отличном структурированном типе в качестве
// заданного поля данного типа

type record MyRecordType {
  integer     Monday
};
// Данное определение является правильным – оно повторно использует
// идентификатор перечисления Monday в отличном структурированном типе в
// качестве идентфкатора для заданного поля данного типа

type record MyNewRecordType {
  MyFirstEnumType firstField,
  integer     secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// неявная ссылка на MyFirstEnumType с помощью первого элемента
// MyNewRecordType

const integer Monday := 7
// Данное определение не является правильным, так как оно использует
// идентификатор перечисления Monday для другого объекта TTCN-3 в пределах
// одной и той же единицы контекста
```

Каждому перечислению также может факультативно присвоено целое значение, которое определяется в скобках после названия перечисления. Каждое из присвоенных целых значений должно отличаться от других в пределах одного типа **enumerated**. Для каждого перечисления, которому не присвоено целое значение, система последовательно присваивает целые значения в порядке следования текстовых обозначений для перечисления, начиная с левой позиции, которой присваивается значение 0, увеличивая это значение на 1 для каждого следующего перечисления и пропуская те перечисления, которым целые значения были присвоены самостоятельно пользователем. Эти значения используются системой исключительно с целью разрешить применение операторов сравнения.

ПРИМЕЧАНИЕ 1. – Целое значение также может использоваться системой для кодирования/декодирования перечисляемых значений. Однако этот вопрос выходит за рамки данной Рекомендации (за исключением того, что TTCN-3 разрешает ассоциацию атрибутов кодирования с элементами TTCN-3).

Для каждого экземпляра ссылки на значение типа **enumerated** необходимо явная или неявная ссылка на данный тип.

ПРИМЕЧАНИЕ 2. – Если перечисляемый тип является элементом структурированного типа, определенного пользователем, то неявная ссылка на перечисляемый тип выполняется с помощью данного элемента (то есть с помощью идентификатора элемента или с помощью позиции значения в обозначении списка значений) при присваивании значений.

ПРИМЕР 2:

```
// Действительное создание экземпляров MyFirstEnumType и MySecondEnumType:
var MyFirstEnumType Today    := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// однако следующее выражение не разрешено, так как два перечисляемых типа не являются
совместимыми
Today := Tomorrow;
```

6.3.5 Объединения

6.3.5.0 Общие положения

TTCN-3 поддерживает тип **union**. Тип **union** представляет собой набор полей, которые идентифицируются с помощью единственного идентификатора**.** В реальном значении "объединение" всегда присутствует только одно из определенных полей. Типы **union** используются для моделирования структуры, которая может выбирать один тип из ограниченного числа известных типов.

ПРИМЕР:

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

// Действительным текущим значением MyUnionType может быть
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;  // обозначение значения при помощи ссылки на поле. Заметим,
                   // что такое обозначение также делает это поле выбранным
oneYearOlder := (number := age.number+1);
ageInMonths := age.number*12;
```

Для установки значений для значений типа **union** не должна использоваться нотация списка значений.

6.3.5.1 Обращение к поля для типа union

Обращение к полям для типа **union** выполняется с помощью обозначения точки (см. 6.3.1.1).

ПРИМЕР:

```
MyVar5 := MyUnion1.myChoice1;
// Если тип union является вложенным в другой тип, то обращение выглядит //следующим образом
MyVar6 := MyRecord1.myElement1.myChoice2;
// Заметим, что тип union, в котором мы обращаемся к полю с идентификатором
// 'myChoice2', вложен с тип record
```

6.3.5.2 Опциональность и объединение

Для типа **union** не разрешается использование факультативных полей, что означает, что ключевое слово **optional** не должно использоваться с типами **union**.

6.3.5.3 Вложенное определение типа для типов полей

TTCN-3 поддерживает определение типов для полей объединения, вложенных в определение объединения, аналогичный механизм для полей типа запись рассматривался в 6.3.1.3.

6.4 Тип anytype

Специальный тип **anytype** определен в качестве краткого условного обозначения для объединения всех известных типов данных и типа адреса в модуле TTCN-3. Определение понятия "известные типы" приводится в 3.1, то есть **anytype** объединяет все известные типы данных, но типы **port**, **component** и **default** сюда не относятся. Тип **address** должен быть включен в том случае, если он был явным образом определен в пределах данного модуля.

Названия полей для **anytype** должны уникальным образом идентифицироваться соответствующими названиями типов.

ПРИМЕЧАНИЕ 1. – В результате данного требования импортированные типы, названия которых приводят к конфликту (или конфликтуют с идентификатором определения в импортируемом модуле, или конфликтуют с идентификатором, импортированным из третьего модуля), не могут быть получены при помощи **anytype** для импортируемого модуля.

ПРИМЕР:

```
// допустимое использование anytype выглядит следующим образом
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

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

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

Тип **anytype** определяется локально для каждого модуля и (подобно другим предопределенным типам) не может непосредственно импортироваться другим модулем. Тем не менее, другой модуль может импортировать тип **anytype**, определенный пользователем. В результате этого импортируются все типы данного модуля.

ПРИМЕЧАНИЕ 2. – Тип **anytype**, определенный пользователем, "содержит" все типы, импортированные в модуль, где он был определен. Импортируя в модуль такой тип, определенный пользователем, может привести к сторонним эффектам и, следовательно, в таких случаях следует соблюдать необходимую осторожность.

6.5 Массивы

Как во многих языках программирования, массивы не считаются типами в TTCN-3. Вместо этого они могут определяться в момент объявления какой-либо переменной. Массивы могут определяться как одномерные или многомерные. Размерность массива должна быть указана при помощи выражения с константой, которая должна давать в результате положительное **integer**-значение.

ПРИМЕР 1:

```
var integer MyArray1[3];      // создает целый массив, содержащий 3 элемента,
                              // индекс которых изменяется от 0 до 2
var integer MyArray2[2][3];   // создает двумерный целый массив, содержащий
                              // 2 × 3 элемента, индексы которых изменяются
                              // от (0,0) до (1,2)
```

Доступ к элементам массива обеспечивается при помощи обозначения индекса ([ ]), при этом должно указываться действительное значение индекса в пределах диапазона значений для данного массива. Доступ к отдельным элементам многомерного массива обеспечивается при помощи повторного использования обозначения индекса. Если попытаться получить доступ к элементам массива, которые выходят за диапазон для данного массива, то это приводит к ошибке компиляции или ошибке тестового варианта.

ПРИМЕР 2:

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

MyArray1[4] := 12;     // ОШИБКА: индекс должен лежать в диапазоне от 0 до 2
MyArray2[3][2] := 15;  // ОШИБКА: первый индекс должен быть равен 0 или 1
```

Размерность массива также может быть определена при помощи указания диапазона. В этом случае нижнее и верхнее значение диапазона определяют минимальное и максимальное значение индекса.

ПРИМЕР 3:

```
var integer MyArray3[1 .. 5];  // создает целый массив из 5 элементов
                               // индекс принимает значения от 1 до 5
MyArray3[1] := 10;     // значение для минимального индекса
MyArray3[5] := 50;     // значение для максимального индекса

var integer MyArray4[1 .. 5][2 .. 3];   // создает двумерный целый массив,
                                        // состоящий из 5 × 2 элементов с индексами
                                        // от (1,2) до (5,3)
```

Значения элементов массива должны быть совместимы с соответствующим объявлением переменных. Значения могут присваиваться индивидуально при помощи обозначения списка значений или при помощи индексированного обозначения, или же могут присваиваться сразу несколько значений при помощи обозначения списка значений. При использовании обозначения списка значений первое значение в списке присваивается первому элементу массива (элемент, индекс которого равен 0), второе значение в списке присваивается второму элементу и так далее. Элементы, для которых присваивание не должно выполняться, должны быть явно пропущены или не указаны в списке. При присваивании значений многомерным массивам каждой размерности, для которой выполняется присваивание, должен соответствовать набор значений, заключенный в фигурные скобки. При указании значений для многомерных массивов размерность, которая является самой левой, соответствует наиболее удаленной структуре значения, а самая правая размерность соответствует самой внутренней структуре. Разрешается использование секций массивов для многомерных массивов, то есть когда количество индексов значений массивов меньше, чем количество размерностей в соответствующем определении массива. Индексы для секций массивов должны соответствовать размерностям массива в порядке слева направо (то есть индекс секции соответствует первой размерности в определении). Индексы секции должны соответствовать размерностям в определении соответствующего массива.

ПРИМЕР 4:

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

// or using a value list
MyArray1:= {10, 20, -, 30};

MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
   // значения массива определены полностью

var integer MyArray5[2][3][4] :=
{
   {
      {1, 2, 3, 4}, // присваивает значение секции [0][0] из MyArray5
      {5, 6, 7, 8}, // присваивает значение секции [0][1] из MyArray5
      {9, 10, 11, 12} // присваивает значения секции [0][2] из MyArray5
   }, // конец присваивания для секции [0] из MyArray5
   {
      {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
      } // присваивает значение секции [1] из MyArray5
};

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

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

var integer MyArrayInvalid[2][2];
MyArrayInvalid := { 1, 2, 3, 4 }
   // неправильно, так как размерность обозначения данных не соответствует
   // размерностям в определении
MyArrayInvalid[2] := { 1, 2 }
   // неправильно, так как индекс секции должен быть равен 0 или 1
```

ПРИМЕЧАНИЕ. – Альтернативным способом. Для использования многомерных данных является использование типов **record , record of, set** или **set of**.

ПРИМЕР 5:

```
// Задано
type record MyRecordType
{
   integer     field1,
   MyOtherStruct  field2,
   charstring     field3
}
// Массивом MyRecordType может быть
var MyRecordType myRecordArray[10];
// Ссылка на конкретный элемент может быть такой
myRecordArray[1].field1 := 1;
```

6.6 Рекурсивные типы

Там, где это применимо, определения типов TTCN-3 могут быть рекурсивными. Пользователь, однако, должен обеспечить, чтобы все рекурсии типов были разрешимыми и чтобы не появилась бесконечная рекурсия.

6.7 Совместимость типов

6.7.0 Общие положения

В общем случае в TTCN-3 требуется совместимость типов для значений при операциях присваивания, создания и сравнения.

Для целей данного раздела в тех случаях, когда действительное значение, которое должно быть присвоено, передается в качестве параметра или каким-то другим способом, то мы будем обозначать его как значение "b". Тип, который имеет значение "b", будет обозначаться как тип "B". Тип формального параметра, который должен получить действительное значение значения "b', будет называться типом "A".

6.7.1. Совместимость типов для неструктурированных типов

Для переменных, констант, шаблонов и т. п., относящихся к простым базовым типам, а также для типов **bitstring**, **hexstring** и **octetstring** значение "b" является совместимым с типом "A", если тип "B" разрешается к тому же корневому типу, что и тип "A" (например, к типу **integer**), и при этом он не нарушает правил для подтипов (то есть диапазон, ограничение по длине) для типа "A",

ПРИМЕР:

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

// Тогда
y := 5;    // Действительное присвоение

x := y;     // Действительное присвоение, так как y имеет тот же корневой тип, что и
// x, при этом не нарушаются правила для подтипов

x := 20;   // Действительное присвоение
y := x;
// Недействительное присвоение, так как значение x находится вне диапазона MyInteger

x := 5;    // действительное присвоение
y := x;
// Действительное присвоение, так как значение x теперь находится
// в пределах диапазона MyInteger

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

// тогда
myCharString := mySingleCharString;
// является дествительным присваиванием, так как charstring, длина которой
// ограничена 1, является совместимой с charstring.
myCharacter := mySingleCharString;
// действительное присваивание, так как две charstring с длиной в один символ
// являются совместимыми.

// дано
myCharString := "abcd";

// тогда
myCharacter := myCharString[1];
// является действительным, так как обозначение в правой части относится к
// индивидуальному элементу строки

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

// тогда
myCharString := myCharacterArray[1];
// является действительным и присваивает myCharString значение "B";
```

Для переменных, констант, шаблонов и т. п., относящихся к типу **charstring**, значение "b" является совместимым с **universal sharstring** типа "A", если только при этом не нарушаются заданные ограничения типа (диапазон, список или длина) для типа "A".

Для переменных, констант, шаблонов и т.п., относящихся к типу **universal charstring**, значение "b" является совместимым с "A" типа **charstring** в том случае, если все символы, которые используются в значении "b", имеют соответствующим им символы (то есть одинаковые управляющие или графические символы, которые используют тот же код символа) в типе **charstring**, при этом не нарушаются никакие заданные ограничения типа (диапазон, список или длина) для типа "A".

6.7.2. Совместимость типов для структурированных типов

6.7.2.0 Общие положения

Для структурированных типов (за исключением типа **enumerated**) значение "b" типа "B" является совместимым с типом "A" в том случае, если эффективные структуры значений типа "B" и "A" являются совместимыми, в этом случае разрешены операции присваивания, создания и сравнения.

6.7.2.1 Совместимость типа для перечисляемых типов

Перечисляемые типы никогда не являются совместимыми с другими базовыми или структурированными типами (то есть для перечисляемых типов необходимо строгое соответствие типов).

6.7.2.2 Совместимость типа для типов **record** и **record of**

Для типов **record** эффективные структуры значений являются совместимыми в том случае, если число, а также факультативный аспект поля в текстовом порядке определения являются идентичными, совместимыми являются типы каждого поля и значение каждого существующего поля для значения "b" является совместимым с типом соответствующего поля в типе "A". Значение каждого поля в значении "b" присваивается соответствующему полю в значении типа "A".

ПРИМЕР 1:

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

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

type record CType {     // тип с отличными названиями полей
  integer     d       optional,
  integer     e       optional,
  boolean     f

}

type record DType {     // тип, в котором поле c является факультативным
  integer     a       optional,
  integer     b       optional,
  boolean     c       optional
}

type record EType {     // тип с дополнительным полем d
  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};

  // тогда

  MyVarA := MyVarB;        // является действительным присваиванием,
                           // значение MyVarA равно ( a := <undefined>, b:= 2, c:= true)
  MyVarC := MyVarB;        // является действительным присваиванием
                           // значение MyVarC равно ( d := <undefined>, e:= 2, f:= true)
  MyVarA := MyVarD;        // не является действительным присваиванием по причине
                           // того, что факультативность полей не совпадает
  MyVarA := MyVarE;        // не является действительным присваиванием по причине
                           // того, что не совпадает количество полей

  MyVarC := { d:= 20 };    // действительное значение MyVarC равно { d:=20,e:=2,f:= true }
  MyVarA := MyVarC         // не является действительным присваиванием по причине
                           // того, что поле i 'd' из MyVarC нарушает правила
                           // подтипа для поля 'a' из AType
```

Для типов **record of** и для массивов эффективные структуры значений являются совместимыми в том случае, если типы их компонентов являются совместимыми и значение "b" типа "B" не нарушает правила длины для подтипа типа **record of** или размерность массива типа "A". Значения элементов значения "b" должны быть последовательно присвоены экземпляру типа "A", включая неопределенные элементы.

Типы **record of** и одномерные массивы являются совместимыми с типами **record** в том случае, если их эффективные структуры значений являются совместимыми и количество элементов значения "b" для типа "B", соответствующего **record of**, или же размерность массива "b" в точности равняется количеству элементов типа "A", соответствующего **record of**. При определении совместимости факультативность полей для типа **record** не имеет никакого значения, то есть она не учитывается при подсчете полей (что означает, что факультативные поля должны всегда учитываться при подсчете). Присваивание значений элементов типа **record** или массива экземпляру типа **record** должно выполняться в текстовом порядке, соответствующему определению типа **record**, включая неопределенные элементы. Если элемент с неопределенным значением присваивается факультативному элементу из **record**, то это приводит к тому, что факультативный элемент пропускается. Попытка присвоить элемент с неопределенным значением обязательному элементу из **record** приводит к ошибке.

ПРИМЕЧАНИЕ.– Если тип **record** не имеет ограничений по длине или ограничение длины превышает количество элементов сравниваемого типа **record**, а индекс любого определенного элемента значения **record of** меньше или равен количеству элементов типа **record** минус один, то условие совместимости всегда выполняется.

Значения типа **record** могут также присваиваться экземплярам типа **record of** или одномерному массиву при условии, что не нарушаются ограничения по длине для типа **record of** или размерность массива превышает или равна количеству элементов типа **record**. Факультативные элементы, которые отсутствуют в значении **record**, должны присваиваться как элементы с неопределенными значениями.

ПРИМЕР 2:

```
// дано
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];

// тогда

MyArrayVar := MyVarH;
// является действительным присваиванием, так как тип MyArrayVar и HType
// являются совместимыми

MyVarI := MyVarH;
// является действительным присваиванием, так как типы являются совместимыми
// и не нарушаются правила для подтипов

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// НЕ является действительным присваиванием, так как обязательное поле 'c'
// из Htype не получает значения
```

6.7.2.3 Совмесимость типа для типов set и set of

Типы **set** являются единственными типами, совместимыми с типами **set** и типами **set of**. Для типов **set** и для типов **set of** должны применяться те же правила совместимости, что и для типов **record** и **record of**.

ПРИМЕЧАНИЕ 1. – Это подразумевает, что хотя порядок элементов при передаче и при приеме неизвестен, решающее значение при определении совместимости для типов **set** имеет текстовый порядок полей в определении типа.

ПРИМЕЧАНИЕ 2. – В значениях **set** порядок полей может быть произвольным; тем не менее, это никак не влияет на совместимость типа, так как названия полей однозначно определяют, какие поля из связанного типа **set** соответствуют полям значений **set**.

ПРИМЕР:

```
// дано
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};

// тогда

MyVarF := MyVarG;    // является действительным присваиванием, так как типы
                     // FType и GType совместимы

MyVarF := MyVarA;    // не является действительным присваиванием, так как
                     // MyVarA относится к типу record
```

6.7.2.4 Совместимость для подструктур

Правила, определенные в данном разделе для совместимости структурированных типов, также относятся и к подструктурам таких типов.

ПРИМЕР:

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

var JType MyVarJ

// если рассматривать приведенную выше декларацию, то


MyVarJ.H := MyVarH;
// является действительным присваиванием, так как типы поля H
// для JType и HType являются совместимыми

MyVarI := MyVarJ.H;
// является действительным присваиванием, так как IType и тип поля H
// из JType являются совместимыми
```

6.7.3 Совместимость типов для типов компонентов

Совместимость типа для типов компонентов должна быть рассмотрена исходя из двух различных условий:

1) Совместимость между значением ссылки для компонента и типом компонента (например, при передаче ссылки на компонент в качестве действительного параметра в функцию или altstep, или же при присваивании значения ссылки для компонента переменной, которая имеет тип другого компонента): ссылка на компонент "b", которая имеет тип компонента "B", является совместимой с типом компонента "A" в том случае, если все определения для "A" имеют идентичные определения в "B".

2) Совместимость **runs on** (на ходу): функция или altstep, которые ссылаются на компонент типа "A", в своем выражении **runs on** может вызываться или запускаться с экземпляром компонента типа "B" в том случае, если все определения для "A" имеют идентичные определения в "B".

Идентичность определений в "A" и определений в "B" определяется на основании следующих правил:

- Для экземпляров портов идентичными должны являться как тип, так и идентификатор.
- Для экземпляров таймеров идентичными должны являться идентификаторы, а также либо оба таймера должны иметь одинаковую начальную продолжительность, либо оба не должны иметь начальной продолжительности.
- Для экземпляров переменных и определений констант идентичными должны являться идентификаторы, типы и значения, присваиваемые при инициализации (для переменных это означает, что они либо должны отсутствовать в обоих определениях, либо должны являться одинаковыми).
- Для локальных определений шаблонов идентичными должны являться идентификаторы, типы, списки формальных параметров и присвоенный шаблон или значения полей шаблона.

6.7.4 Совместимость типов для коммуникационных операций

Коммуникационные операции (см. Раздел 23) **send, receive, trigger, call, getcall, reply, getreply** и **raise** являются исключениями в слабых правилах совместимости типа и требуют строгого согласования типов. Типы значений или шаблонов, которые используются в качестве параметров для этих операций, также должны быть явным образом определены в определении типа для связанного порта. Сильная типизация также применяется при хранении полученного значения, адреса или ссылки на компонент при операциях **receive** или **trigger**.

6.7.5 Преобразование типа

Если необходимо преобразовать значения одного типа в значения другого, когда эти типы не получены из одного и того же корневого типа, то следует использовать либо одну из предопределенных функций преобразования, приведенных в Приложении C, либо функцию, определяемую пользователем.

ПРИМЕР:

```
// Чтобы преобразовать значение integer в значение hexstring, используйте
// предопределенную функцию int2hex
MyHstring := int2hex(123, 4);
```

7 Модули

7.0 Общие положения

Главными строительными блоками TTCN-3 являются модули. Например, модуль может определять полностью выполнимую тестовую последовательность или всего лишь библиотеку. Модуль состоит из определяющей части (факультативной) и управляющей части (факультативной).

ПРИМЕЧАНИЕ. – Термин "тестовая последовательность" является синонимом полного модуля TTCN-3, содержащего тестовые примеры и управляющую часть.

7.1 Именование модулей

Имена модулей являются формой идентификатора TTCN-3. В дополнение к этому, спецификация модуля может содержать дополнительный атрибут, который идентифицируется с помощью ключевого слова language, которое указывает на редакцию языка TTCN-3, которая используется в спецификации модуля. На данный момент для редакции языка поддерживаются следующие строковые значения: "TTCN-3: 2001" для спецификации модуля, совместимой с редакцией TTCN-3 2001, "TTCN-3: 2003" для редакции 2003, и "TTCN-3: 2005" для редакции 2006.

ПРИМЕЧАНИЕ. – Идентификатор модуля является неформальным текстовым именем модуля.

ПРИМЕР:

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

7.2 Параметры модуля

7.2.0 Общие положения

Список параметров **module** определяет набор значений, которые выдаются тестовой средой во время выполнения. В процессе выполнения теста эти значения рассматриваются как константы. Параметры модуля декларируются с помощью указания типа и перечисления идентификаторов, которое следует за ключевым словом modulepar. Параметры модуля не должны иметь тип порт, тип по умолчанию или тип компонента. Параметр модуля может иметь только тип адрес, если тип адрес явным образом определен в связанном модуле. Параметры модуля могут определяться только в пределах области определений модуля. Допускается неоднократное использование декларации параметров модуля, однако каждый параметр должен быть декларирован только один раз (это означает, что не допускается повторное определение параметра модуля).

ПРИМЕР:

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

7.2.1 Значения по умолчанию для параметров модуля

Для параметров модуля разрешается определять безусловные (по умолчанию) значения. Это делается путем присвоения в списке параметров модуля. Значение по умолчанию может являться исключительно символьной константой, которая присваивается только в месте декларирования параметра. Если тестовая система не предоставляет действительного значения во время выполнения для данного параметра, то при выполнении теста используется значение по умолчанию, в противном случае используется действительное значение, предоставленное тестовой системой.

ПРИМЕР:

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

7.3 Определяющая часть (область определений) модуля

7.3.0 Общие положения

Определяющая часть модуля дает определения верхнего уровня для модуля и может ипортировать идентификаторы из других модулей. Контекстные правила для деклараций, сделанных в определяющей части модуля, и для импортированных деклараций приводятся в 5.3. Те элементы языка, которые могут определяться в модуле TTCN-3, перечислены в таблице 1. Определения модуля могут импортироваться другими модулями.

ПРИМЕР:

```
module MyModule
{  // Этот модуль содержит только определения
   :
   const integer MyConstant := 1;
   type record MyMessageType { … }
   :
   function TestStep(){… }
   :
}
```

Объявления динамических элементов языка, таких как **var** или **timer**, делаются только в управляющей части, тестовых примерах, функциях, altstep или типах компонентов.

ПРИМЕЧАНИЕ. – TTCN-3 не поддерживает объявление переменных в определяющей части модуля. Это означает, что глобальные переменные не могут быть определены в TTCN-3. Тем не менее, переменные, которые определяются в тестовом компоненте, могут использоваться вместе с тестовыми примерами, функциями и так далее, которые выполняются с данным компонентом, а переменные, которые определены в управляющей части, обеспечивают возможность сохранять свои значения независимо от выполнения тестового примера.

7.3.1 Группы определений

В определяющей части модуля определения могут собираться в именованные группы. Может быть определена группа объявлений там, где разрешено одиночное объявление. Группы могут быть вложенными, то есть они могут содержать другие группы. Это позволяет спецификатору (описателю) тестовой последовательности структурировать, среди прочего, совокупности тестовых данных или функций, описывающих поведение теста.

Группирование производится для повышения удобства чтения и добавления логической структуры в модуль, если это требуется. Группы и вложенные группы не имеют ограничений, *за исключением* ограничения на контекст любых идентификаторов групп и атрибутов, приданных группе с помощью соответствующей команды **`with`**. Это означает:

- Идентификаторы группы не должны являться уникальными в пределах всего модуля. Тем не менее, все идентификаторы групп для подгрупп одной группы должны являться уникальными. При необходимости должно использоваться обозначение точки для уникальной идентификации подгрупп в иерархии группы, например, для импорта какой-то подгруппы.
- Правила, которые переопределяют правила для атрибутов, приводятся в 28.4.

  ПРИМЕР:

```
module MyModule {
    :
// Совокупность определений
group MyGroup {
    const integer MyConst:= 1;
    :
    type record MyMessageType { … };
    group MyGroup1 { // подгруппа с определениями
          type record AnotherMessageType { … };
          const boolean MyBoolean := false
     }
    }

    // группа altstep
    group MyStepLibrary {
     group MyGroup1 { // подгруппа с тем же названием, что и подгруппа
                      // с определениями
          altstep MyStep11() { … }
          altstep MyStep12() { … }
           :
          altstep MyStep1n() { … }
     }
     group MyGroup2 {
          altstep MyStep21() { … }
          altstep MyStep22() { … }
          :
          altstep MyStep2n() { … }
     }
    }
    :
}

// команда импортирования, которая импортирует MyGroup1 в MyStepLibrary
import from MyModule {
group MyStepLibrary.MyGroup1
}
```

7.4 Управляющая часть модуля

Управляющая часть модуля может содержать локальные определения и описывает порядок выполнения (возможно, повторяющихся) реальных тестовых примеров. Каждый тестовый пример должен определяться в определяющей части модуля и вызываться в управляющей части.

ПРИМЕР:

```
module MyTestSuite
{ // Этот модуль содержит определения
   :
   const integer MyConstant := 1;
   type record MyMessageType { … }
   template MyMessageType MyMessage := { … }
   :
   function MyFunction1() { … }
   function MyFunction2() { … }
   :
   testcase MyTestcase1() runs on MyMTCType { … }
   testcase MyTestcase2() runs on MyMTCType { … }
   :
   // ... и управляющую часть, так что имеется выполнимое
   control
   {
      var boolean MyVariable; // локальная управляющая переменная
      :
   execute(MyTestCase1());    // последовательное выполнение тестовых примеров
   execute(MyTestCase2());
      :
   }
}
```

7.5 Импортирование из модулей

7.5.0 Общие положения

Возможно повторное использование определений, сформулированных в различных модулях, при помощи команды **import**. В TTCN-3 нет явной конструкции экспорта; следовательно, по умолчанию, все определения модуля в определяющей части модуля могут быть импортированными. Команда **import** может использоваться где угодно в определяющей части модуля. Она не должна использоваться в управляющей части.

Если импортируемое определение имеет атрибуты (определяемые с помощью команды **with**), то эти атрибуты также должны импортироваться. Механизм, который позволяет изменить атрибуты импортированных определений, объясняется в 28.6.

ПРИМЕЧАНИЕ. – Если модуль имеет глобальные атрибуты, то они связываются с определениями без этих атрибутов.

ПРИМЕР:

```
module MyModuleA
{ // Этот модуль содержит определения и импортированные определения
   :
     const integer MyConstant := 1;
     import from MyModuleB all; // Контекст импортированных определений
                                // является глобальным для MyModuleA
import from MyModuleC {
           type MyType1, MyType2;
           template all
}
   type record MyMessageType { … }
   :
   function MyBehaviourC()
   {
      const integer MyConstant := 2;
      // здесь не может использоваться импорт
      :
   }
   :
   control
   {   // Здесь не может использоваться импорт
      :
   }
}
```

7.5.1 Структура импортируемых определений

TTCN-3 поддерживает импорт следующих определений: параметров модуля, типов, определенных пользователем, сигнатур, консиани, внешних констант, шаблонов данных, шаблонов сигнатур, функций, внешних функций, altstep и тестовых примеров. Каждое определение имеет *имя* (определяет идентификатор определения, например, имя функции), *спецификацию* (например, спецификация типа или сигнатура функции), а в случае функции, altstep или тестового примера – также связанное *описание поведения*.

ПРИМЕР:

	Имя	Спецификация	Описание поведения

function	MyFunction	(**inout** MyType1 MyPar) **return** MyType2
		runs on MyCompType
			{
			const MyType3 MyConst := …;
			: // дальнейшее поведение
			}

	Спецификация	Имя	Спецификация

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

	Спецификация	Имя	Спецификация

template	MyType5	MyTemplate	:= {
			field1 := 1,
			field2 := MyConst, // MyConst является константой модуля
			field3 := ModulePar // ModulePar является параметром модуля
			}

Описание поведения не оказывает никакого влияния на механизм импорта, так как считается, что их внутренняя часть являются невидимыми для импортера при импорте соответствующих функций, altstep или тестовых примеров. Таким образом, они в последующих определениях не рассматриваются.

Часть спецификации импортируемого определения содержит *локальные определения* (например, названия полей для определений структурированных типов или значения для перечисляемых типов) и *ссылочные определения* (например, ссылки на определения типа, шаблоны, константы или параметры модуля). Для приведенного выше примера это означает:

	Имя	Локальные определения	Ссылочные определения
function	MyFunction	MyPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType4, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

ПРИМЕЧАНИЕ 1. – Столбец с локальными определениями относится исключительно к новым идентификаторам, которые определяются в импортируемом определении. Значения, присвоенные индивидуальным полям импортируемых определений, например, определениям шаблонов, также могут рассматриваться как локальные определения, однако это не имеет важного значения при объяснении механизма импорта.

ПРИМЕЧАНИЕ 2. – Ссылочные определения field1, field2 и filed3 шаблона MyTemplate являются именами полей MyType5, то есть ссылка на них осуществляется при помощи MyType5.

Ссылочные определения также являются импортируемыми определениями, то есть источник ссылочного определения может вновь быть структурирован на имя и часть спецификации, а часть спецификации также содержит локальные и ссылочные определения. Другими словами, импортируемое определение может быть рекурсивно построено из других импортируемых определений.

Механизм импорта TTCN-3 связан с локальными и ссылочными определениями, которые используются в части спецификации импортируемых определений. Следовательно, Таблица 5 специфицирует возможные локальные и ссылочные определения для импортируемых определений.

Таблица 5/Z.140 – Возможные локальные и ссылочные определения для импортируемых определений

Импортируемое определение	Возможные локальные определения	Возможные ссылочные определения
Параметр модуля		Тип параметра модуля
Тип, определенный пользователем (для всех)	Имена параметров	Тип параметра
• перечисляемый тип	Конкретные значения	
• структурированный тип	Названия полей	Типы полей
• тип порт (port)		Типы сообщений, сигнатуры
• тип компонент	Названия констант, названия переменных, названия таймеров и названия портов	Типы констант, типы переменных, типы портов
Сигнатура	Названия параметров	Типы параметров, возвращаемые типы, типы исключений
Константа		Тип константы
Внешняя константа		Тип константы
Шаблон данных	Названия параметров	Тип шаблона, типы параметров, константы, параметры модуля, функции
Шаблон сигнатуры		Определение сигнатуры, константы, функции параметров модуля
Функция	Названия параметров	Типы параметров, возвращаемые типы, типы компонентов (выражение **runs on**)
Внешняя функция	Названия параметров	Типы параметров, возвращаемый тип
Altstep	Названия параметров	Типы параметров, тип компонента (выражение **runs on**)
Текстовый пример	Названия параметров	Типы параметров, типы компонентов (выражения **runs on** и **system**)

Механизм импорта TTCN-3 делает различие между *идентификатором ссылочного определения* и *информацией, необходимой для использования ссылочного определения* в пределах импортированного определения. Для использования идентификатор ссылочного определения не нужен, следовательно, он не будет импортироваться автоматически.

7.5.2 Правила использования импорта

При использовании импорта должны применяться следующие правила:

a) импортироваться могут лишь определения самого верхнего уровня в модуле. Определения, находящиеся на более низком уровне контекста (например, локальные константы, определенные в какой-либо функции), не должны импортироваться.

b) разрешается только прямое импортирование из исходного модуля определения (то есть модуля, где находится действительное определение идентификатора, на который ссылается выражение **`import`**).

c) определение импортируется вместе со своим названием и всеми локальными определениями.

ПРИМЕЧАНИЕ 1. – Локальное определение, например, название поля для определенного пользователем типа записи, имеет смысл исключительно в контексте определений, в которых оно определено, например, название поля для типа запись может использоваться только для доступа к полю этого типа записи и не может использоваться за пределами этого контекста.

d) Определение импортируется вместе со всей информацией ссылочных определений, которая необходимо для использования ссылочного определения.

ПРИМЕЧАНИЕ 2. – Команды импортирования являются транзитивными, то есть если модуль A импортирует определение из модуля B, который в свою очередь использует ссылку на тип, который определен в модуле C, то соответствующая информация, необходимая для использования этого типа, автоматически импортируется в модуль A.

e) Идентификаторы ссылочных определений не импортируются автоматически.

ПРИМЕЧАНИЕ 3. – Если желательно использовать в импортирующем модуле ссылочные определения, то они должны быть в явном виде импортированы из исходного модуля.

f) При импорте функции, altstep или тестового примера, соответствующая спецификация поведения и все определения, которые используются внутри спецификации поведения, остаются невидимыми для импортирующего модуля.

g) Циклическое импортирование запрещено.

ПРИМЕР:

```
module ModuleONE {

    modulepar integer ModPar1, ModPar2 := 7

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

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

      const integer MyConst := 13;

      template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= {
// Параметризованный шаблон
       Field1_T2 := TempPar_T2,          // ссылка на параметр шаблона
       Field2_T2 := {MyConst, true},     // ссылка на константу модуля
       Field3_T2 := ModPar1              // ссылка на параметр модуля
      }

  } // конец модуля ModuleONE

  module ModuleTWO {

      import from ModuleONE {
       template Template_T2
      }

// В модуле ModuleTWO будут видны только названия Template_T2 и TempPar_T2
// Заметьте, что идентификатор TempPar_T2 может использоваться только в
// контексте Template_T2, например, когда предоставляет значение
// действительного параметра. Вся информация, необходимая для использования
  // Template_T2, например, для целей проверки типа, импортируется для ссылочных
  // определений RecordType_T2, RecordType_T1, Field1_T2, Field2_T2,
  // Field3_T3, MyConst и ModPar1, однако их идентфкаторы являются невидимыми в
  // ModuleTWO. Это означает, что невозможно использовать константу MyConst или
  // объявлять переменную типа RecordType_T1 или RecordType_T2 в ModuleTWO
  // без импортирования этих типов в явном виде.

      import from ModuleONE {
       modulepar ModPar2
      }

      // Параметр модуля ModPar2 из ModuleONE импортируется ModuleONE и
      // может использоваться как целая константа

  } // конец модуля ModuleTWO

  module ModuleTHREE {

      import from ModuleONE all; // импортирует все определения из ModuleONE

      type port MyPortType {
       inout RecordType_T2
      }

      type component MyCompType {
       var integer MyComponentVar := ModPar2; // ссылка на параметрр модуля из
                                              // ModuleONE
       port MyPortType MyPort
      }

      function MyFunction () return integer {
       return MyConst   // возвращает константу модуля, которая определена в
                        // ModuleONE
      }

      testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

       var integer MyTCVar := ModPar2; //ссылка на параметр модуля из ModuleONE

       MyPort.send(Template_T2); // отправка шаблона, определенного в ModuleONE
       MyPort.receive(RecordType_T2 : ?) -> value MyPar; // Присваивание
                              // полученного значения out-параметру MyPar.

      } // конец тестового примера MyTestCase

  } // конец ModuleTHREE

  module ModuleFOUR {

      import from ModuleTHREE {
       testcase MyTestCase
      }
```

```
// только имена MyTestCase и MyPar будут видимы и могут использоваться в
// ModuleFOUR. Информация о типе для RecordType_T2 импортируется с помощью
// ModuleTHREE из ModuleONE, а информация о типе для MyCompType импортируется
// из ModuleTHREE. Все определения, которые используются в области поведения
// MyTestCase остаются скрытыми для пользователя ModuleFOUR.

  } // конец ModuleFOUR
```

7.5.3 Действие отменено

7.5.4 Импортирование одиночных операций

Могут быть импортированы одиночные определения.

ПРИМЕР:

```
import from MyModuleA {
    type MyType1          // импортирует одно определение типа из MyModuleA
}

import from MyModuleB {
    type MyType2, Mytype3, MyType4;     // импортирует три типа
    template MyTemplate1;               // импортирует один шаблон
    const MyConst1, MyConst2            // испортирует две константы
}
```

7.5.5 Импортирование всех определений из модуля

Все определения, имеющиеся в части определений модуля, могут импортироваться с помощью ключевого слова **all**, расположенного вслед за названием модуля. Если из модуля импортируются все определения с помощью ключевого слова **all**, то никакая другая форма импортирования (импорт единичного определения, импорт для одного типа и так далее) не должна использоваться для того же выражения **import**.

ПРИМЕР 1:

```
import from MyModule all;
```

Если нет необходимости импортировать некоторые определения, то их виды и идентификаторы должны быть перечислены в списке исключений, который заключен в фигурные скобки, которые следуют за ключевым словом **except**.

ПРИМЕР 2:

```
from MyModule all except {
      type MyType3, MyType5
      // исключает из выражения импорта объявления типов MyType3 и MyType5
      // и импортирует все другие определения из MyModule
   }
```

Ключевое слово **all** также может использоваться в списке исключений; это позволяет исключить объявления того же вида из выражения импорта.

ПРИМЕР 3:

```
import from MyModule all except {
type MyType3, MyType5;   // исключает два данных типа из выражения импорта
    template all         // исключает все шаблоны, которые определены в MyModule,
                         // из выражения импорта
}
```

7.5.6 Импорт групп

Могут импортироваться группы определений.

ПРИМЕР 1:

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

Эффект от импортирования группы идентичен выражению import, которое перечисляет все импортируемые определения (включая подгруппы) для данной группы.

В TTCN-3 группы используются исключительно для целей структурирования и не являются единицами контекста. Следовательно, разрешается непосредственно импортировать подгруппы (то есть группы, определенные внутри другой группы), то есть не обязательно указывать группу, к которой относится подгруппа. Если имя подгруппы, которую следует импортировать, идентично имени другой подгруппы в том же модуле (см. 7.3.1), то для того, чтобы однозначно определить импортируемую подгруппу следует использование обозначение с помощью точки.

Если некоторые из определений группы не следует импортировать, то их виды и идентификаторы должны быть перечислены в списке исключений, который заключается в пару фигурных скобок, расположенных вслед за ключевым словом except.

ПРИМЕР 2:

```
import from MyModule {
    group MyGroup except {
     type MyType3, MyType5
     // исключает из выражения импорта определения типов MyType3 и MyType5
     // однако импортирует все другие определения MyGroup
    }
}
```

Ключевое слово all также может использоваться в списке исключений; это позволяет исключить объявления того же вида из выражения импорта.

ПРИМЕР 3:

```
    import from MyModule {
        group MyGroup except {
            type MyType3, MyType5;   // исключает два типа из выражения импорта и
            template all             // исключает из выражения импорта все шаблоны, которые
                                     // определены в MyGroup
        }
    }
```

7.5.7 Импорт определений одного вида

Ключевое слово all может использоваться для импорта из модуля всех определений одного вида. Ключевое слово all, которое используется совместно с ключевым словом constant, идентифицирует все константы, а также внешние константы, объявленные в части определений модуля, к которым относится выражение импорта. Подобным образом, ключевое слово all, которое используется совместно с ключевым словом function, идентифицирует все функции и все внешние функции, определенные в модуле, который обозначен в выражении импорта.

ПРИМЕР 1:

```
    import from MyModule {
        type all;                        // импортирует все типы из MyModule
        template all                     // импортирует все шаблоны из MyModule
    }
```

Если объявления какого-то типа необходимо исключить из данного выражения импорта, то необходимо перечислить их идентификаторы вслед за ключевым словом except.

ПРИМЕР 2:

```
    import from MyModule {
        type all except MyType3, MyType5;   // импортирует все типы за исключением
                                            // MyType3 и MyType5
        template all                        // импортирует все шаблоны, определенные в
                                            // Mymodule
    }
```

7.5.8 Обработка столкновений имен при импорте

Все модули TTCN-3 располагают собственным пространством имен, в котором все определения имеют уникальные идентификаторы. Столкновение (совпадение) имен может возникнуть из-за импорта, например импорта из других модулей. Столкновение имен должно преодолеваться путем добавления к импортируемому определению (которое вызывает столкновение имен) префикса в виде идентификатора того модуля, из которого оно импортировано. Такой префикс и идентификатор разделяются точкой (.).

В случаях, когда неоднозначностей не возникает, при использовании импортированных определений не обязательно должен (но может) присутствовать префикс. Когда дается ссылка на определение в том же модуле, в котором оно определено, то в качестве префикса идентификатора определения также может использоваться идентификатор модуля для этого модуля (текущего модуля).

ПРИМЕР:

```
module MyModuleA
{  :
   type bitstring MyTypeA;
   import from SomeModuleC {
       typeMyTypeA,     // здесь MyTypeA – символьная строка
            MyTypeB     // здесь MyTypeB – символьная строка
       }
   :
   control
   {   :
       var  SomeModuleC.MyTypeA MyVar1 := "Test String";  // Должен быть использован префикс
       var MyTypeA MyVar2 := '10110011'B;    // Это – исходный MyTypeA
       :
```

```
        var MyTypeB MyVar3 := "Test String";  // Префикс не требуется …
        var SomeModuleC.MyTypeB MyVar3 := "Test String";
                                              // … но при желании он может быть
// использован
        :
    }
}
```

ПРИМЕЧАНИЕ. – Определения с одним и тем же именем, определенные в разных модулях, всегда считаются разными, даже если реальные определения в разных модулях идентичны. Например, импортирование какого-либо типа, который уже определен на местном уровне, даже с тем же именем, приведет к двум разным типам, доступным в рассматриваемом модуле.

7.5.9 Обработка нескольких ссылок на одно и то же определение

Применение операции **import** к одиночным определениям, группам определений, определениям одного и того же вида и т. д. может привести к ситуациям, когда на некоторое определение дается более одной ссылки. Такие случаи должны разрешаться системой и определения импортируются только один раз.

ПРИМЕЧАНИЕ. – Механизмы устранения такой неоднозначности, например перезапись и передача предупреждений пользователю, выходят за рамки данной Рекомендации и должны обеспечиваться инструментами TTCN-3.

Все выражения **import** и определения, которые находятся внутри выражений импорта, должны обрабатываться независимо друг от друга в порядке их появления. Важно указать, что выражение **except** не исключает перечисленные определения от импортирования в общем случае; все определения выражений импорта могут рассматриваться как сокращенное обозначение эквивалентного списка идентификаторов одиночных определений. Выражение **except** исключает определения только из этого отдельного списка.

ПРИМЕР:

```
import from MyModule {
     type all except MyType3;    // импортирует все типы из MyModule за
                                 // исключением MyType3
     type MyType3                // явным образом импортирует MyType3 explicitly
}
```

7.5.10 Импорт определений из модулей не TTCN-3

В тех случаях, когда определения импортируются из других источников, не являющихся модулями TTCN-3, то необходимо использовать спецификацию языка для обозначения языка (возможно, вместе с указанием его версии) источника (например, модуль, пакет, библиотека или даже файл), из которого импортируются определения. Для этого используется ключевое слово **language**, за которым следует текстовое определение для обозначенного языка. При импортировании из TTCN-3 модуля той же редакции, что и импортирующий модуль, использование спецификации языка является необязательным. Когда обнаруживается несовместимость между идентификацией языка (включая неявную идентификацию, когда спецификация языка не используется) и синтаксиса модуля, из которого импортируются определения, инструменты должны обеспечить разумные усилия для разрешения конфликта.

В TTCN-3 определены следующие идентификаторы языка:

- 'TTCN-3:2001' – используется для модулей, которые соответствуют версии 2001 данной Рекомендации (см. список литературы)
- 'TTCN-3:2003' – используется для модулей, которые соответствуют версии 2003 данной Рекомендации (см. список литературы)
- 'TTCN-3:2005' – используется для модулей, которые соответствуют данной Рекомендации.

ПРИМЕР:

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

ПРИМЕЧАНИЕ. – Механизм импортирования применяется для того, чтобы разрешить использование определений из других модулей TTCN-3 или модулей на других языках. Правила для импорта определений из спецификаций, написанных на других языках, например, из пакетов SDL, могут соответствовать правилам TTCN-3 или же их необходимо будет определять отдельно.

8 Тестовые конфигурации

8.0 Общие положения

TTCN-3 обеспечивает возможность (динамической) спецификации конфигураций параллельных тестов (или, для краткости, конфигураций). Конфигурация состоит из набора взаимосоединенных тестовых компонентов с четко определенными портами связи и явным интерфейсом тестовой системы, который определяет границы тестовой системы.

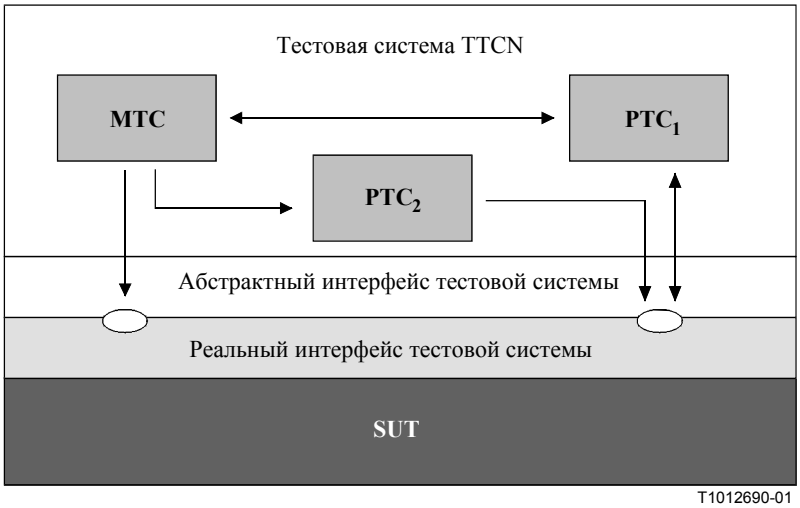


Рисунок 3/Z.140 – Схематическое представление типичной тестовой конфигурации TTCN-3

Внутри каждой конфигурации должен быть один (и только один) Главный тестовый компонент (MTC). Тестовые компоненты, не являющиеся MTC, называются параллельными тестовыми компонентами (PTC). MTC создается системой автоматически при запуске каждого выполнения тестового примера. Поведение, определенное в теле тестового примера, должно выполняться в этом компоненте. Во время выполнения тестового примера могут динамически создаваться другие компоненты путем явного применения операции **`create`**.

Выполнение тестового примера заканчивается, когда останавливается MTC. Все остальные PTC равноправны, то есть отсутствуют явные иерархические взаимоотношения между ними, а остановка одного PTC не останавливает другие компоненты или MTC. При остановке MTC тестовая система должна остановить все PTC, которые еще не были остановлены на момент окончания выполнения тестового примера.

Связь между тестовыми компонентами и между этими компонентами и интерфейсом тестовой системы обеспечивается через порты связи (см. 8.1).

Типы тестовых компонентов и типы портов, обозначаемые ключевыми словами **`component`** и **`port`**, определяются в определяющей части модуля. Реальная конфигурация компонентов и соединения между ними обеспечиваются путем выполнения операций **`create`** и **`connect`** в рамках поведения тестового примера. Порты компонентов соединяются с портами интерфейса тестовой системы с помощью операции **`map`** (см. п. 22.2).

8.1 Модель связи портов

Тестовые компоненты соединяются через свои порты, то есть соединения между компонентами, а также между компонентом и интерфейсом тестовой системы ориентированы на порты. Каждый порт моделируется в виде бесконечной очереди FIFO, которая накапливает входящие сообщения или вызовы процедур до их обработки компонентом, к которому принадлежит этот порт.

ПРИМЕЧАНИЕ. – Несмотря на то что порты TTCN-3 являются в принципе бесконечными, в реальной тестовой системе они могут быть перегружены. Это должно рассматриваться как ошибка тестового примера (см. п. 25.2.1).

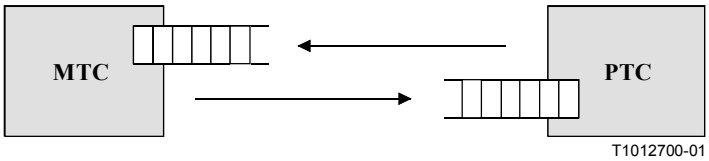


Рисунок 4/Z.140 – Модель портов связи TTCN-3

8.2 Ограничения на соединения

Соединения TTCN-3 представляют собой соединения порт-порт или порт-интерфейс тестовой системы (см. Рисунок 5). Не существует ограничений на количество соединений, которое может поддерживать один компонент. Также разрешены соединения один-многие (см. Рисунок 5{g} или Рисунок 5(h)).

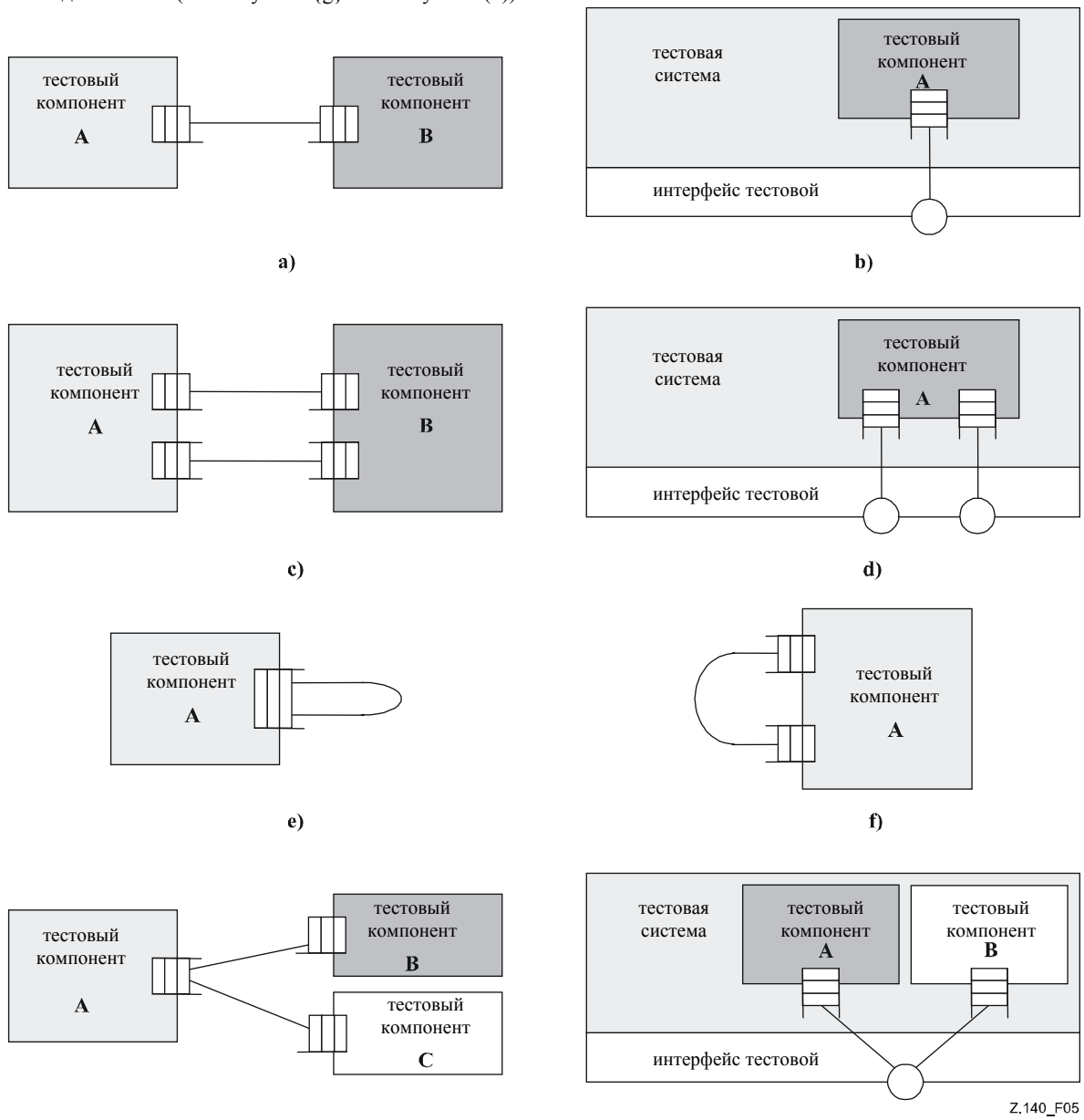


Рисунок 5/Z.140 – Разрешенные соединения

Не разрешаются следующие соединения:

Порт, который принадлежит компоненту A, не должен соединяться с одним или более портами, принадлежащими тому же компоненту (Рисунок 6(a) и Рисунок 6(e)).

Порт, принадлежащий компоненту A, не должен соединяться с одним или более портами, принадлежащими компоненту B (Рисунок 6(c)).

Порт, принадлежащий компоненту A, может иметь только соединения типа один-с-одним для интерфейса тестовой системы. Это означает, что не разрешены соединения, показанные на Рисунках 6(b) и 6(d).

Не разрешены соединения внутри интерфейса тестовой системы (см. Рисунок 6(f)).

Порт, который имеет соединение, не должен отображаться, а порт, который уже был отображен, не должен соединяться (см. Рисунок 6(g)).

Так как TTCN-3 разрешает динамические конфигурации и адреса, то ограничения на соединения не могут быть всегда проверены во время компилирования. Такие проверки производятся во время выполнения, и при неудаче приводят к ошибке тестового примера.

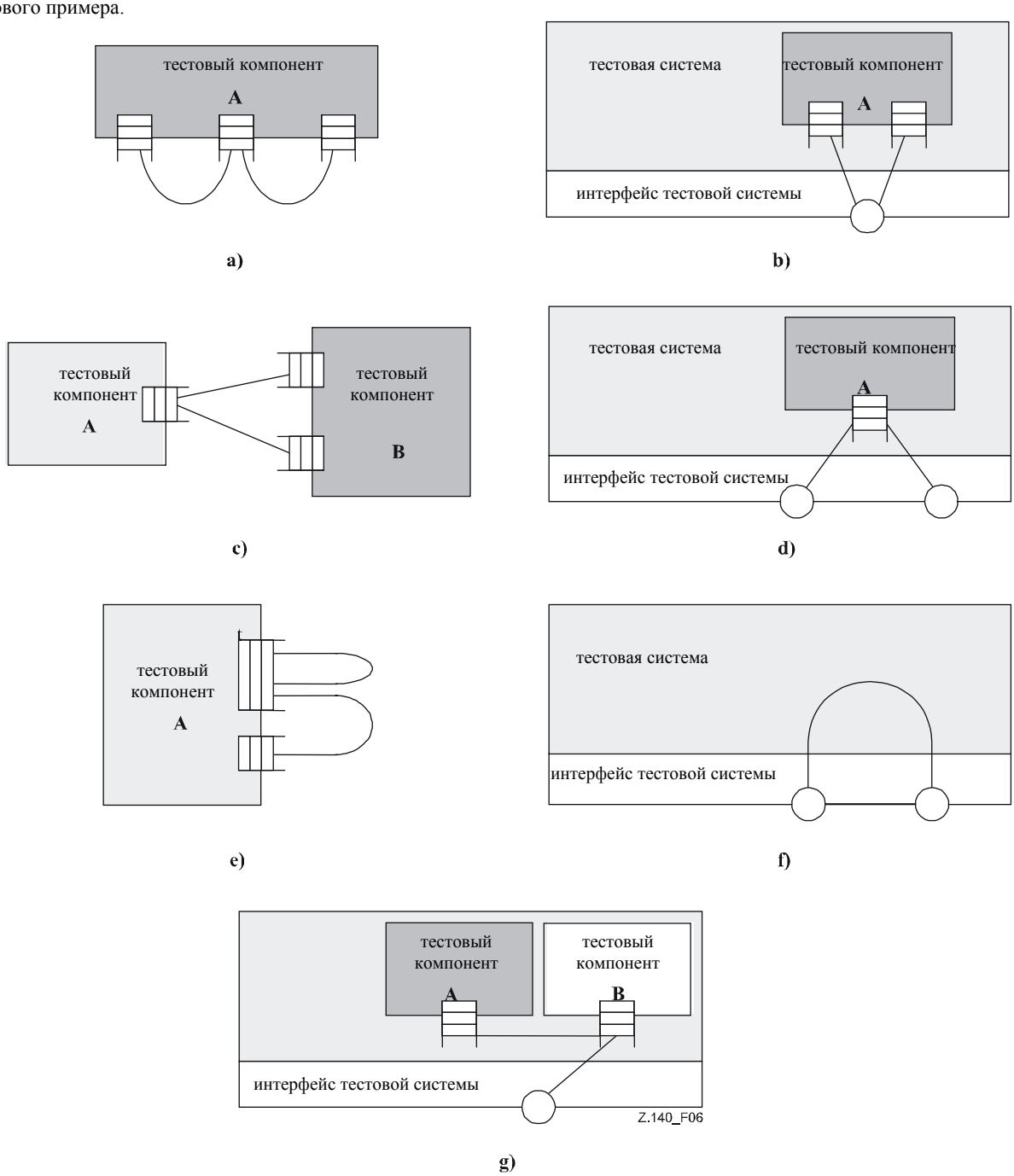


Рисунок 6/Z.140 – Запрещенные соединения

8.3 Абстрактный интерфейс тестовой системы

TTCN-3 используется в тестовых реализациях. Объект, который тестируется, называется тестируемой реализацией, или IUT. IUT может предоставлять прямые интерфейсы для тестирования либо быть частью системы; в последнем случае тестируемый объект называется тестируемой системой, или SUT. В минимальном случае IUT и SUT будут эквивалентны. В данной Рекомендации термин SUT используется повсеместно, обозначая либо SUT, либо IUT.

В реальной тестовой среде тестовым примерам необходимо связываться с SUT. Однако спецификация реального физического соединения выходит за рамки TTCN-3. Вместо этого с каждым тестовым примером связывается четко определенный (но абстрактный) интерфейс тестовой системы. Определение интерфейса тестовой системы идентично определению компонента, то есть оно представляет собой список всех возможных портов связи, через которые тестовый пример соединяется с SUT.

Интерфейс тестовой системы статическим образом определяет во время выполнения теста количество и тип соединений между портом и SUT. Однако соединения между интерфейсом тестовой системы м тестовыми компонентами TTCN-3 являются по своей природе динамическими и могут изменяться во время выполнения теста при помощи операций **map** и **unmap**.

8.4 Определение типов портов связи

8.4.0 Общие положения

Порты содействуют связи между тестовыми компонентами, а также между тестовыми компонентами и интерфейсом тестовой системы.

TTCN-3 обеспечивает порты на базе сообщений и на базе процедур. Каждый порт должен быть определен как порт на базе сообщений или на базе процедур (или одновременно могут реализованы оба варианта, см. 8.4.1). Порт на базе сообщений должен указываться ключевым словом **message** и порт на базе процедур должен указываться ключевым словом **procedure** внутри определения типа связанного порта.

Порты являются двунаправленными. Направления определяются ключевыми словами **in** (для входящего направления), **out** (для исходящего направления) и **inout** (для обоих направлений). Каждое определение типа порта должно иметь один или несколько списков, указывающих разрешенную совокупность типов (сообщений) и/или процедур наряду с разрешенным направлением связи.

Когда для "out"-направления порта на базе процедур определяется сигнатура (см. также раздел 13), типы всех ее параметров **inout** и **out**, возвращаемый тип и типы исключений автоматически являются частью направления "in" для данного порта. Когда для "in"-направления порта на базе процедур определяется сигнатура, типы всех ее параметров **inout** и **out**, возвращаемый тип и типы исключений автоматически являются частью направления "out" для данного порта.

ПРИМЕР:

```
// Порт на базе сообщений, который разрешает типы MsgType1 и MsgType2
// для приема, MsgType3 для передачи и любое целочисленное значение,
// которое будет передаваться и приниматься через этот порт
type port MyMessagePortType message
{
  in      MsgType1, MsgType2;
  out MsgType3;
  inout    integer
  }

// Порт на базе процедур, который разрешает удаленный
// вызов процедур Proc1, Proc2 и Proc3. Отметим, что
// Proc1, Proc2 и Proc3 определены в виде сигнатур
type port MyProcedurePortType procedure
{
  out Proc1, Proc2, Proc3
}
```

ПРИМЕЧАНИЕ. – Термин "сообщение" используется для обозначения как сообщений, определенных в шаблонах, так и реальных значений, выведенных из выражений. Поэтому ограничивающий список, указывающий, что можно использовать через порт на базе сообщений, является просто списком имен типов.

8.4.1 Смешанные порты

Возможно определить такой порт, который позволяет оба вида связи. Это обозначается ключевым словом **mixed**. Это означает, что списки смешанных портов также будут смешанными и будут содержать как сигнатуры, так и типы. В определении не проводится различие.

```
// Смешанный порт, определяющий порт на базе сообщений и порт на базе
// процедур под одним именем. Списки in, out и inout также являются смешанными:
// MsgType1, MsgType2, MsgType3 и integer относятся к порту на базе сообщений
// в смешанном порту, а Proc1, Proc2, Proc3, Proc4 и Proc5 относятся к порту
// на базе процедур
type port MyMixedPortType mixed
{
  in      MsgType1, MsgType2, Proc1, Proc2;
  out MsgType3, Proc3, Proc4;
  inout    integer, Proc5;
}
```

Смешанный порт в TTCN-3 определяется как краткая нотация для двух портов, то есть порта на базе сообщений и порта на базе процедур под одним именем. Во время выполнения различие между такими двумя портами проводится с помощью операций связи.

Операции, используемые для управления портами (см. раздел 23.5), то есть **start, stop** и **clear**, должны выполнять операции в обеих очередях (в произвольном порядке), если они вызваны идентификатором смешанного порта.

8.5 Определение типов компонентов

8.5.0 Общие положения

Тип **component** определяет, какие порты связаны с компонентом. Эти определения даются в определяющей части модуля. Имена портов в определении компонента являются местными для этого компонента, то есть другой компонент может иметь порты с теми же именами. Все порты одного компонента должны иметь уникальные имена. Определение компонента само по себе не означает, что между компонентами имеется какое-либо соединение через эти порты.

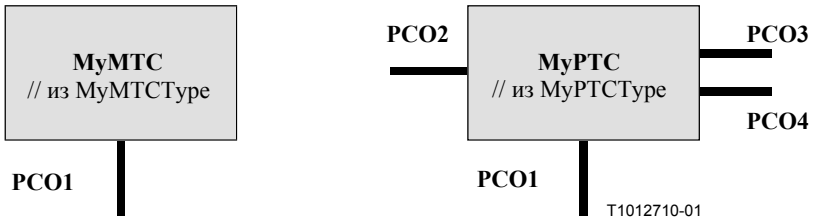


Рисунок 7/Z.140 – Типичные компоненты

ПРИМЕР:

```
type component MyMTCType
{
  port MyMessagePortType       PCO1
}

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

8.5.1 Объявление местных переменных, констант и таймеров в компоненте

Возможно объявлять константы, переменные и таймеры, местные для конкретного компонента.

ПРИМЕР:

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

Эти объявления видны для всех тестовых примеров, функций и altstep, которые выполняются в данном компоненте. Что явно устанавливается с помощью ключевого слова **runs** (см. раздел 16).

Переменные и таймеры компонента связаны с экземпляром компонента и следуют контекстным правилам, определенным в п. 5.3. Поэтому каждый новый экземпляр компонента будет иметь свой собственный набор переменных и таймеров, указанный в определении компонента (включая любые начальные значения, если они устанавливаются).

ПРИМЕЧАНИЕ. – При использовании в качестве интерфейсов тестовой системы (см. 8.8), компоненты не могут использовать любые константы, переменные и таймеры, объявленные в этом компоненте.

8.5.2 Определение компонентов с множеством портов

Возможно определять множества портов в определениях типов компонентов (см. также п. 21.12).

ПРИМЕР:

```
type component My3pcoCompType
{
  port MyMessageInterfaceType PCO[3]
  port MyProcedureInterfaceType PCOm[3][3]
  // Определяет тип компонента, который имеет массив из 3 портов на базе
   // сообщений и двумерный массив из портов на базе процедур.
}
```

8.5.3 Расширение типов компонентов

Возможно определить типы компонентов как расширения других типов компонентов с помощью ключевого слова **extends**.

ПРИМЕР 1:

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

В таком определении определение нового типа обозначается как *расширенный тип*, а тип определения, который следует за ключевым словом **extends**, обозначается как *родительский тип*.

В результате этого определения расширенный тип будет неявно содержать все определения родительского типа. Таким образом, приведенное выше определение эквивалентно записи (и следовательно носит название *определение эффективного типа*).

ПРИМЕР 2:

```
// в результате определение из Примера 1 эквивалентно следующему:
type component MyExtendedMTCType
{
  /* определения из MyMTCType */
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType PCO1

  /* дополнительные определения */
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

Разрешается расширять типы компонентов, которые определяются с помощью расширения, если только не возникает циклической цепи определений.

ПРИМЕР 3:

```
type component MTCTypeA extends MTCTypeB { /* … */ };
type component MTCTypeB extends MTCTypeC { /* … */ };
type component MTCTypeC extends MTCTypeA { /* … */ };  // ОШИБКА – циклическое
                                                       // расширение
type component MTCTypeD extends MTCTypeD { /* … */ };  // ОШИБКА – циклическое
                                                       // расширение
```

При определении типов компонентов при помощи расширений, не должно существовать конфликтов имен между определениями, взятыми из родительского типа и определений, которые были добавлены в расширенном типе; то есть, не должен существовать идентификатор порта, переменной, константы, таймера или шаблона, который объявлен как в родительском типе (непосредственно или с помощью расширения) и в расширенном типе.

ПРИМЕР 4:

```
type component MyExtendedMTCType extends MyMTCType
{
  var integer MyLocalInteger;    // ОШИБКА – уже определено в MyMTCType (см.
                                 // Пример 2)
  var float MyLocalTimer;        // ОШИБКА – таймер с таким именем уже
                                 // существует в MyMTCType
  port MyOtherMessagePortType PCO1; // ОШИБКА – порт с таким именем существует
                                 // в MyMTCType
}

type component MyBaseComponent { timer MyLocalTimer };
type component MyInterimComponent extends MyBaseComponent { timer MyOtherTimer };
type component MyExtendedComponent extends MyInterimComponent
{
  timer MyLocalTimer; // ОШИБКА – уже определен в MyInterimComponent с помощью
                      // расширения
}
```

Разрешается иметь один компонентный, расширяющий в одном определении несколько родительских типов, которые в определении список типов, которые разделяются между собой запятой.

ПРИМЕР 5:

```
type component MyCompA extends MyCompB, MyCompC, MyCompD {
  /* дополнительные определения для MyCompA */
}
```

Каждый из родительских типов также может быть определен с помощью расширения.

Эффективное определение типа компонента расширенного типа получается как коллекция всех определений констант, переменных, таймеров, портов и шаблонов, которые предоставляются родительскими типами (определяется рекурсивно, если родительский тип определяется с помощью расширения) и определений, которые непосредственно объявляются в расширенном типе. Определение эффективного типа компонента не должны иметь конфликты имен. Для выполнения этого условия, в пределах набора родительских типов , которые используются в определении расширенного типа, все определения должны иметь уникальные имена и эти имена должны отличаться от любого из имен в определениях, которые объявляются непосредственно в расширенном типе.

ПРИМЕЧАНИЕ 1. – Считается, что объявление является различным и не вызывает ошибки, если одно и то же определение вносится в расширенный тип различными родительскими типами (при помощи различных путей расширения).

ПРИМЕР 6:

```
type component MyCompB { timer T };
type component MyCompC { var integer T };
type component MyCompD extends MyCompB, MyCompC {}
  // ОШИБКА – столкновение имен между MyCompB и MyCompC

// MyCompB определен выше
type component MyCompE extends MyCompB {
  var integer MyVar1 := 10;
}

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

type component MyCompG extends MyCompB, MyCompE, MyCompF {
  // Нет столкновения имен.
  // Все родительские типы MyCompG имеют таймер T, либо непосредственно, либо
// с помощью расширения MyCompB; так как все они происходят
// (непосредственно или с помощью расширения) от таймера T, объявленного
// в MyCompB, что делает данную форму столкновения легальной
  /* здесь находятся дополнительные определения */
}
```

Семантики типов компонентов с расширениями определяются с помощью простой замены каждого определения типа компонента на его определение эффективного типа компонента в качестве этапа предварительной обработки перед его использованием.

ПРИМЕЧАНИЕ 2. – Относительно совместимости компонентных типов это означает, что компонентная ссылка с типа CT1, которая расширяет CT2, является совместимой с CT2, и те тестовые примеры, функции и altstep, которые указывают CT2 в своих выражениях **runs on**, могут выполняться с компонентной ссылкой с (см. 6.7.3).

8.6 Адресация объектов внутри SUT

SUT может состоять из нескольких объектов, имеющих индивидуальную адресацию. Тип данных адреса является типом для использования с операциями порта при адресации элементов SUT. При использовании **to**, **from** и **sender** тип данных адреса должен использоваться в операциях передачи и приема для портов, отобюраженных на интерфейс тестовой системы. Представление реальных данных в **address** определяется либо явным определением типа в тестовой последовательности, либо извне тестовой системой (то есть тип **address** остается открытым типом в спецификации TTCN-3). Это позволяет определять абстрактные тестовые примеры независимо от любых реальных адресных механизмов, специфичных для конкретной SUT.

Явные адреса SUT должны генерироваться только внутри модуля TTCN-3, если тип определяется внутри этого модуля. Если тип не определяется внутри модуля, то явные адреса SUT должны передаваться только в виде параметров либо приниматься в полях сообщений или в виде параметров удаленных вызовов процедуры.

Кроме того, можно использовать специальное значение **null** для указания на неопределенный адрес, например, для инициализации переменных адресного типа.

ПРИМЕР:

```
// Связывает целое число типа с адресом открытого типа
type integer address;
  :
// Переменная нового адреса, инициализированная с нулем
var address MySUTentity : =null;
 :
// Прием адресного значения и присвоение его переменной MySUTentity
PCO.receive (address :*) -> value MySUTentity;
 :
// Использование принятого адреса для передаваемого шаблона MyResult
PCO.send (MyResult) to MySUTentity;
 :
// Использование принятого адреса для приема подтверждающего шаблона
PCO.receive (MyConfirmation) from MySUTentity;
```

8.7 Компонентные ссылки

Компонентные ссылки – это однозначные ссылки на тестовые компоненты, которые создаются во время выполнения тестового примера. Такая уникальная компонентная ссылка генерируется тестовой системой во время создания компонента, то есть компонентная ссылка является результатом операции **create** (см. п. 22.1). Дополнительно компонентные ссылки выдаются стандартными операциями **system** (выдает компонентную ссылку для указания портов в интерфейсе тестовой системы), **mtc** (выдает компонентную ссылку на MTC) и **self** (выдает компонентную ссылку на компонент, в котором вызывается **self**).

Компонентные ссылки используются в конфигурирующих операциях **connect, map** и **start** (см. раздел 22) для установления тестовых конфигураций, а также в частях **from, to** и **sender** операций связи портов, соединенных с тестовыми компонентами, отличных от **interface** тестовой системы для целей адресации (см. раздел 23 и Рисунок 5).

Кроме того, можно использовать специальное значение **null** для указания на неопределенную компонентную ссылку, например, для инициализации переменных при обработке компонентных ссылок.

Представление реальных данных компонентных ссылок должно определяться извне тестовой системой. Это позволяет определять абстрактные тестовые примеры независимо от любой реальной обстановки во время выполнения TTCN-3; другими словами, TTCN-3 не ограничивает реализацию тестовой системы в части обработки и идентификации тестовых компонентов.

ПРИМЕЧАНИЕ. – Компонентная ссылка содержит информацию о типе компонента. Это означает, например, что в переменной для обработки компонентных ссылок при ее объявлении должно использоваться имя соответствующего типа компонента.

ПРИМЕР:

```
// Определение типа компонента
type component MyCompType {
  port PortTypeOne PCO1;
  port PortTypeTwo PCO2
}
// Объявление переменной для обработки ссылок на компоненты типа
// MyCompType и создания компонента этого типа
var MyCompType MyCompInst := MyCompType.create;

// Использование компонентных ссылок в конфигурирующих операциях,
// постоянно имеющих ссылку на компонент, созданный выше
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self));     // self передается в виде параметра к
                                        // MyBehavior

// Использование компонентных ссылок в разделах from и to
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:?) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MPCO2.send(integer:5) to MyCompInst;

// Следующий пример поясняет случай соединения типа "один ко многим" в
// порту PCO1, где значения типа M1 могут приниматься от отдельных компонентов
// различных типов CompType1, CompType2 и CompType3 и где должен быть
// выбран передатчик. В этом случае может использоваться следующая схема:
 :
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); // Из функции выбирается некоторый
                                          // результат
 :
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 Определение интерфейса тестовой системы

Определение типа компонента используется для определения интерфейса тестовой системы, так как определения типов компонентов и определения интерфейсов тестовых систем имеют в принципе одинаковую форму (оба являются совокупностями портов, определяющих возможные точки соединения).

ПРИМЕЧАНИЕ – Переменные, таймеры и константы, объявленные в типах компонентов, которые используются в интерфейсах тестовой системы, не являются действующими.

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

Как правило, ссылка на тип компонента, определяющая интерфейс тестовой системы, связана с каждым тестовым примером, использующим более одного тестового компонента. Порты интерфейса тестовой системы автоматически реализуются системой вместе с MTC при запуске выполнения тестового примера.

Операцией, возвращающей ссылку на компонент интерфейса тестовой системы, является **system**. Она может использоваться для адресации портов тестовой системы.

ПРИМЕР:

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

В случае, когда MTC является единственным компонентом, реализуемым во время выполнения теста, не требуется связывать интерфейс тестовой системы с тестовым примером. В этом случае определение типа компонента, связанное с MTC, неявно определяет соответствующий интерфейс тестовой системы.

9 Объявление констант

Константы можно объявлять и использовать в определяющей части модулях, в определениях типа компонента, в управляющей части модуля, тестовых примерах и в функциях и altstep. Определения констант обозначаются ключевым словом **const**. Константы не должны иметь тип порта. Значение константы присваивается в момент объявления.

ПРИМЕЧАНИЕ. – Единственное значение, которое может присваиваться константам с типом по умолчанию или компонентного типа, является специальное значение **null**.

ПРИМЕР 1:

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

Присвоение значения для константы может производиться внутри модуля либо извне. Последний случай представляет собой внешнее объявление константы, обозначаемое ключевым словом **external**.

ПРИМЕР 2:

```
external const integer MyExternalConst; // объявление внешней константы
```

Внешняя константа может иметь произвольный тип, за исключением типа порт, типа по умолчанию или компонентного типа, но этот тип должен быть известен в модуле, то есть он должен быть корневым типом или типом, который определен пользователем, определенным в модуле или импортированным из какого-либо другого модуля. Отображение этого типа во внешнее представление какой-либо внешней константы и механизм введения внешней константы в модуль выходят за рамки данной Рекомендации.

10 Объявление переменных

10.0 Общие положения

Переменные могут иметь простой базовый тип, базовый строковый тип, структурированные типы, специальные типы данных (включая подтипы, производные от других типов) а также типы адрес, компонентный тип или тип по умолчанию.

ПРИМЕЧАНИЕ. – Переменные компонентного или структурированного типа могут объявляться только на базе типов, определенных пользователем.

Переменные можно объявлять и использовать в управляющей части модуля, в тестовых примерах, в функциях и altstep. Кроме этого, переменные могут объявляться в определениях компонентного типа. Такие переменные могут использоваться в тестовых примерах, altstep и функциях, выполняемых для данного компонентного типа. Переменные не объявляются и не используются в определяющей части модуля (то есть глобальные переменные не поддерживаются в TTCN-3).

Использование неинициализированных или не полностью инициализированных переменных в позициях, отличных от левой стороны в выражениях присваивания или в качестве действительных параметров, передаваемых формальным out-параметром, приводит к ошибке.

10.1 Переменные со значениями

Переменные со значениями объявляются при помощи ключевого слова **var**, за которым следует идентификатор типа и идентификатор переменной. В объявлении переменной может присваиваться начальное значение. Переменные со значениями могут хранить только значения и могут использоваться как в левой, так и в правой стороне выражения присваивания, в выражениях, вслед за ключевым словом **return** в теле функций, в заголовках которых используется выражение возврата, а также могут передаваться формальным параметрам со значениями или с типом шаблон.

ПРИМЕР 1:

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

10.2 Шаблонные переменные

Шаблонные переменные объявляются при помощи ключевого слова **var template**, за которым следуют идентификатор типа и идентификатор переменной. В объявлении также может быть присвоен начальный контент. В отличие от значений, шаблонные переменные также могут хранить механизмы сопоставления (см. 14.3). Они могут использоваться как в правой, так и в левой стороне выражения присваивания, вслед за ключевым словом **return** в теле функций, в заголовках которых определяется возвращаемое значение типа шаблон, а также могут передаваться формальным параметрам с типом шаблон. При использовании в правой стороне присваивания они не должны являться операндами операторов TTCN-3 (см. раздел 15), а переменные в левой стороне также должны являться шаблонными переменными. Также разрешается присваивать экземпляр шаблона шаблонной переменной или полю шаблонной переменной.

ПРИМЕЧАНИЕ. – Шаблонные переменные, подобно глобальным или локальным шаблонам, должны быть полностью специфицированы перед использованием в операциях приема и передачи.

ПРИМЕР:

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

Хотя не разрешается непосредственно применять операции TTCN-3 к шаблонным переменным, допускается использовать обозначение точки и обозначение индекса для проверки и изменения полей шаблонной переменной. Правила, которые определяют применение данных обозначений при попытке доступа к полям, находящимся вне механизма сопоставления, приводятся в 14.3.1.

11 Объявление таймеров

11.0 Общие положения

Таймеры можно объявлять и использовать в управляющей части модуля в тестовых примерах, в функциях и altstep. Кроме этого, таймеры могут объявляться в определениях компонентного типа. Такие таймеры также могут использоваться в тестовых примерах, функциях и altstep, которые выполняются для данного компонентного типа. Объявление таймера может иметь присвоенное ему факультативное безусловное (по умолчанию) значение выдержки. Таймер должен запускаться с этим значением, если не указано другое значение. Это значение должно иметь неотрицательное значение типа **float** (то есть более или равно 0.0), причем основной единицей является секунда.

ПРИМЕР 1:

```
timer MyTimer1 := 5E-3;// Объявление таймера MyTimer1 с безусловным
                       // значением 5 мс

timer MyTimer2;         // Объявление таймера MyTimer2 без безусловного
                       // значения, то есть значение должно быть присвоено
                       // при запуске таймера
```

В дополнение к единичным экземплярам таймеров, также могут объявляться массивы таймеров. Продолжительность(-и) по умолчанию для элементов массива таймеров присваиваются при помощи массива значений. Присваивание продолжительности(-ей) по умолчанию производится с помощью обозначения массива значений согласно разделу 6.5. Если для некоторых элементов массива таймров присваивание продолжительности по умолчанию следует пропустить, то это необходимо явно объявить при помощи неиспользуемого символа ("-").

ПРИМЕР 2:

```
timer t_Mytimer1[5] := { 1.0, 2.0, 3.0, 4.0, 5.0 }
    // все элементы массива таймеров получают продолжительность по умолчанию.

timer t_Mytimer2[5] := { 1.0, -, 3.0, 4.0, 5.0 }
    // второй таймер (t_Mytimer2[1]) не получает продолжительность по
    // умолчанию.
```

11.1 Таймеры в качестве параметров

Таймеры можно переслать с помощью ссылок только в функции и altstep. Таймеры, пересланные в функцию или altstep, известны внутри определения поведения функции или altstep.

Таймеры, пересланные в качестве параметра с помощью ссылки, могут использоваться как любой другой таймер, то есть они не нуждаются в объявлении. Запущенный таймер также можно переслать в функцию или altstep. Таймер продолжает свою работу, то есть его не останавливают неявно. Таким образом, возможные события, связанные с тайм-аутом, могут быть обработаны внутри функции или altstep, к которой таймер был переслан.

ПРИМЕР:

```
// Определение функции с таймером в списке формальных параметров
function MyBehaviour (timer MyTimer)
{  :
   MyTimer.start;
   :
}
```

12 Объявление сообщений

Одним из ключевых элементов TTCN-3 является способность передавать и принимать сложные сообщения через порты связи, определенные тестовой конфигурацией. Такими сообщениями могут быть сообщения, явно связанные с тестированием SUT или с внутренней координацией, а также управляющие сообщения, специфичные для соответствующей тестовой конфигурации.

ПРИМЕЧАНИЕ. – В TTCN-2 такими сообщениями являются примитивы абстрактной службы (ASP), протокольные блоки данных (PDU) и координирующие сообщения. Базовый язык TTCN-3 является общим в том смысле, что он не делает каких-либо синтаксических или семантических различий подобного рода.

13 Объявление процедурных сигнатур

13.0 Общие положения

Процедурные сигнатуры (или сигнатуры, для краткости) нужны для связи на базе процедур. Связь на основе процедур может использоваться для связи в пределах тестовой системы, то есть между тестовыми компонентами, или же для связи между тестовой системой и SUT. В последнем случае процедура может быть вызвана либо в SUT (то есть этот запрос выполняет тестовая система), либо в тестовой системе (то есть запрос выполняет SUT). Для всех используемых процедур, то есть процедур, используемых для связи между тестовыми компонентами, процедур, вызываемых из SUT, и процедур, вызываемых из тестовой системы, в модуле TTCN-3 должны быть определена полная процедура **signature**.

13.1 Сигнатуры для блокирующей и неблокирующей связи

В TTCN-3 поддерживается *блокирующая* и *неблокирующпая* связь на базе процедур. Определения сигнатур для неблокирующей связи должно использовать ключевое слово **nonblock**, может иметь только **in**-параметры (см. 13.2) и не должно содержать возвращаемого значения (см. 13.3.), однако может возбуждать исключения (см. 13.4). По умолчанию предполагается, что определения сигнатур, которые не содержат ключевого слова **nonblock**, используются для блокирующей связи на базе процедур.

ПРИМЕР:

```
signature MyRemoteProcOne (); // MyRemoteProcOne используется для блокирующей
                              // связи на базе процедур. Она не имеет
                              // параметров и не возвращает значение.

signature MyRemoteProcTwo () noblock;   // MyRemoteProcTwo используется для
                                        // неблокирующей связи на базе процедур.
                                        // Она не имеет параметров и не
                                        // возвращает значение.
```

13.2 Параметры сигнатур процедур

Определения сигнатур могут иметь параметры. В определении **signature** список параметров может содержать идентификаторы параметров, типы параметров и их направления, например, **in**, **out** или **inout**. Направления **inout** или **out** указывают, что данные параметры используются для получения информации от удаленной процедуры. Заметим, что направление параметров определяется с точки зрения вызываемой стороны, а не вызывающей стороны.

ПРИМЕР:

```
signature MyRemoteProcThree (in integer Par1, out float Par2, inout integer Par3)
  // MyRemoteProcThree используется для блокирующей связи на базе процедур.
  // Процедура имеет три параметра: Par1 – параметр с типом integer, Par2 - out
  // параметр типа float и Par3 - inout параметр типа integer.
```

13.3 Удаленные процедуры, возвращающие значение

Удаленная процедура после своего завершения может возвращать значение. Тип возвращаемого значения специфицируется при помощи выражения **return** в соответствующем определении сигнатуры.

ПРИМЕР:

```
  signature MyRemoteProcFour (in integer Par1) return integer;
// MyRemoteProcFour используется для блокирующей связи на базе процедур.
// Процедура имеет параметр Par1 целого типа и возвращает после своего завершения
// значение целого типа
```

13.4 Определение особых состояний (исключений)

Особые состояния (исключения), которые могут возбуждаться удаленными процедурами, представляются в TTCN-3 в виде значений специфического типа. Следовательно, для спецификации или проверки возвращаемых удаленными процедурами значений могут использоваться шаблоны и механизмы сопоставления.

ПРИМЕЧАНИЕ. – Преобразование особых состояний (исключений), генерируемых в или переданных SUT, в соответствующий тип TTCN-3 или представление для SUT, зависит от применяемых инструментов и от системы и, следовательно, в данной Рекомендации оно не рассматривается.

Особые состояния (исключения) определяются в виде списка особых состояний (исключений), включенного в определение сигнатуры. Этот список определяет все возможные различные типы, связанные с набором возможных особых состояний (смысл самих особых состояний будет обычно различаться только с помощью специфических значений этих типов).

ПРИМЕР:

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

  // MyRemoteProcFive используется для блокирующей связи на основе процедур. Она
  // возвращает вещественное значение в inout-параметре Par1 и целое значение,
  // или же может возбуждать исключение типа ExceptionType1 или ExceptionType2


  signature MyRemoteProcSix (in integer Par1) noblock
      exception (integer, float);
// MyRemoteProcSix используется для неблокирующей связи на основе процедур. В
  // случае неудачного завершения MyRemoteProcSix возбуждает исключения с типом
  // integer или float.
```

14 Объявление шаблонов

14.0 Общие положения

Шаблоны используются либо для передачи набора отдельных значений, либо для проверки, согласуются ли принятые значения со спецификацией шаблона. Шиблоны могут определяться глобально в определяющей части модуля, локально в тестовом примере, функции, altstep или блоке выражений или же in-line в качестве аргументов операции связи или действительного параметра при вызове тестового примера, функции или altstep.

Шаблоны обеспечивают следующие возможности:

a) они являются способом организации и повторного использования данных, включая простую форму наследования;

b) они могут быть параметризованы;

c) они позволяют применять механизмы сопоставления;

d) их можно использовать как при связи на базе сообщений, так и при связи на базе процедур.

Внутри шаблона можно определять значения, диапазоны и атрибуты сопоставления, а затем использовать их при связи на базе сообщений и на базе процедур. Шаблоны могут быть определены для любого типа или любой процедурной сигнатуры TTCN-3. Шаблоны на базе типа используются для связи на базе сообщений, а шаблоны с сигнатурой – при связи на базе процедур.

Определение шаблона должно специфицировать набор базовых значений или символов сопоставления для всех до единого полей, которые определяются в определении соответствующего типа или сигнатуры, то есть, он является полностью специфицированным. Объявление модифицированного шаблона (см. 14.6) специфицирует только поля, которые должны быть изменены в отличие от базового шаблона, то есть, здесь используется частичная спецификация. Символ NotUsedSymbol должен использоваться в сигнатурах шаблонов только для параметров, которые не имеют отношения, а в объявлениях модифицированных шаблонов, а также в модифицированных inline-шаблонах – для указания того, что для данного поля или элемента нет никаких изменений.

Существует определенное количество ограничений на функции, которые используются в выражениях при спецификации шаблонов или полей шаблонов; они описываются в разделе 16.1.4.

14.1 Объявление шаблонов для сообщений

14.1.0 Общие положения

Экземпляры сообщений с реальными значениями могут определяться с помощью шаблонов. Шаблон можно считать состоящим из набора инструкций, позволяющих сформировать сообщение для передачи или сопоставить принятое сообщение.

Шаблоны могут определяться для любого типа TTCN-3, указанного в таблице 3, за исключением типов **port** и **default**.

ПРИМЕР:

```
// Этот шаблон при использовании в принимающей операции будет сопоставлять любое //
целочисленное значение
template integer Mytemplate ::=*;
// Этот шаблон будет сопоставлять только целочисленные значения 1, 2 или 3
template integer Mytemplate ::=(1, 2, 3);
```

14.1.1 Шаблоны для передаваемых сообщений

Шаблон, применяемый для операции send, определяет полный набор значений полей, которые входят в состав сообщения, предназначенного для передачи через тестовый порт. Во время операции send шаблон должен быть полностью определен, то есть все поля должны быть превращены в реальные значения, а механизмы сопоставления не должны использоваться в полях шаблона ни прямо, ни косвенно.

ПРИМЕЧАНИЕ. – Для передаваемых шаблонов пропуск необязательного поля рассматривается как обозначение значения, а не механизм сопоставления.

ПРИМЕР:

```
// Задано определение сообщения
type record MyMessageType
{
  integer field1 optional,
  charstring  field2,
  boolean field3
}
// Шаблоном сообщения может быть
template MyMessageType MyTemplate:=
{
  field1 := omit,
  field2 := "My string",
  field3 := true
}
// а соответствующей операцией передачи может быть
MyPCO.send(MyTemplate);
```

14.1.2 Шаблоны для принимаемых сообщений

Шаблон, применяемый в операции **receive**, **trigger** или **check**, определяет некоторый шаблон данных, с которым должно сопоставляться входящее сообщение. В принимаемых шаблонах могут использоваться механизмы сопоставления, определенные в Приложении B. Никакого связывания входящих значений с шаблоном не происходит.

ПРИМЕР:

```
// Задано определение сообщения
type record MyMessageType
{
  integer  field1 optional,
  charstring  field2,
  boolean  field3
}

// Шаблоном сообщения может быть
template MyMessageType MyTemplate:=
```

```
{
  field1 := 1,
  field2 := pattern "abc*xyz",
  field3 := true
}

// а соответствующей операцией приема может быть
MyPCO.receive(MyTemplate);
```

14.2 Объявление шаблонов для сигнатур

14.2.0 Общие положения

Экземпляры списков параметров процедур с реальными значениями могут определяться при помощи шаблонов. Шаблоны могут быть описаны для какой-либо процедуры путем ссылки на определение связанной сигнатуры. Шаблон сигнатуры определяет значения и механизмы сопоставления исключительно для параметров процедур, без учета возвращаемого значения. Значения или механизм сопоставления для возвращения должны определяться внутри операции **reply** или **getreply** (см. соответственно 23.3.3 и 23.3.4).

ПРИМЕР:

```
// определение сигнатуры для удаленной процедуры
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3) return integer;

// примеры шаблонов, связанных с определенной процедурной сигнатурой
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 Шаблоны для вызова процедур

Шаблон, применяемый в операции **call** или **reply**, определяет полный набор значений полей для всех параметров **in** и **inout**. Во время операции **call** все параметры **in** и **in inout** в шаблоне должны быть превращены в реальные значения, а механизмы сопоставления не должны использоваться в этих полях ни прямо, ни косвенно. Любое определение шаблона для параметров **inout** просто игнорируется, так как он разрешен, чтобы определять механизмы сопоставления для этих полей или исключать их (см. Приложение B).

ПРИМЕР:

```
// дан пример из раздела 14.2.0

// Действительный вызов, так как все параметры in и inout имеют различающиеся значения
MyPCO.call(RemoteProc:Template1);

// Действительный вызов, так как все параметры in и inout имеют различающиеся значения
MyPCO.call(RemoteProc:Template2);

// Недействительный вызов, так как inout-параметр Par3 имеет атрибут сопоставления,
// а не какое-либо значение
MyPCO.call(RemoteProc:Template3);

// Шаблоны никогда не возвращают значений. В случае Par2 и Par3 значения, возвращаемые
// при вызове операции, должны быть найдены с использованием раздела присвоений в конце
// команды вызова
```

14.2.2 Шаблоны для приема вызовов процедуры

Шаблон, применяемый в операции **getcall**, определяет некоторый шаблон данных, с которым должны сопоставляться входящие поля параметров. В любых шаблонах, применяемых этой операцией, могут использоваться механизмы сопоставления, определенные в Приложении B. Никакого связывания входящих значений с шаблоном не происходит. Любые параметры **out** в процессе сопоставления игнорируются.

ПРИМЕР:

```
// дан пример из раздела 14.2.0

// Действительный getcall, он будет соответствовать, если Par1 == 1 и Par 3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Действительный getcall, он будет соответствовать, если Par1 == 1 иь Par 3 == 3
MyPCO.getcall(RemoteProc:Template2);

// Действительный getcall, он будет соответствовать при Par1 == 1 и любых значениях Par3
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Механизмы сопоставления шаблона

14.3.0 Общие положения

Как правило, механизмы сопоставления используются для замены значений отдельных полей шаблона или даже для замены всего содержимого шаблона. Некоторые механизмы могут использоваться в комбинации.

Механизмы сопоставления и символы групповой операции могут также использоваться "инлайн (в процессе)", но только в принимаемых событиях (то есть в операциях **receive, trigger, getcall, getreply** и **catch**). Они могут появляться в явных значениях.

ПРИМЕР 1:

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

Идентификатор типа может опускаться в том случае, когда значение однозначно указывает на тип.

ПРИМЕР 2:

```
MyPCO.receive ("AAAA"0);
```

ПРИМЕЧАНИЕ. – Могут опускаться следующие типы: **integer, float, Boolean, bitstring, hexstring и octetxtring**.

Однако тип шаблона "инлайн" должен быть в списке порта, через который принят данный шаблон. В случае, когда имеется неоднозначность между указанным в списке типом и типом предоставленного значения (например, из-за наличия подтипа), имя типа должно включаться в команду на прием.

Механизмы сопоставления разделяются на четыре группы:

a) специфичные значения
   - выражение, представляющее конкретное значение;
   - **omit**: значение опускается;

b) специальные символы, которые могут быть использованы *вместо* значений:
   - (…) список значений;
   - **complement** (…): дополнение к списку значений;
   - ? : универсальный символ для любого значения;
   - * : универсальный символ для любого значения или для отсутствия значения вообще (то есть пропущенного значения);
   - (*lowerBound* .. *upperBound*): диапазон целочисленных или вещественных значений от нижней границы до верхней границы включительно;
   - **superset**: по крайней мере все перечисленные элементы, то есть могут существовать и другие;
   - **subset**: в лучшем случае все перечисленные элементы, то есть может существовать меньше;

c) специальные символы, которые могут быть использованы *внутри* значений:
   - ? : универсальный символ для любого отдельного элемента в цепочке, массиве, типе **record of** или **set of**;
   - * : универсальный символ для любого числа последовательных элементов в цепочке, массиве, типе **record of** или **set of** либо для отсутствия элемента вообще (то есть пропущенного элемента);
   - **permutation**: по крайней мере все перечисленные элементы, но в произвольном порядке (заметим, что в качестве элементов списка перестановки также могут использоваться ? и *);

d) специальные символы, которые описывают *атрибуты* значений:
   - **length**: ограничения длины строк для типов строк и количества элементов для **record of**, **set of** и массивов;
   - **ifpresent**: для сопоставления значений факультативных полей (если они не пропущены).

Поддерживаемые механизмы сопоставления и связанные с ними символы (если таковые имеются), а также контекст их применения показаны в таблице 6. В левом столбце данной таблицы перечислены все эквивалентные типы TTCN-3, к которым применяются эти механизмы сопоставления. Полное описание каждого механизма сопоставления можно найти в Приложении B.

Таблица 6/Z.140 – Механизмы сопоставления TTCN-3

Используется для значений типа	Значение	Вместо значений						Внутри значений		Атрибуты	
	Специфичное значение	Список значений	Дополнительный список	Пропущенное значение	Любое значение (?)	Любое значение или его отсутствие (*)	Диапазон	Любой элемент (?)	Любой элемент или его отсутствие (*)	Ограничение длины	ifpresent
boolean	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
integer	Да	Да	Да	Да	Да	Да[a)]	Да				Да[b)]
char	Да	Да	Да	Да	Да	Да[a)]	Да				Да[b)]
universal char	Да	Да	Да	Да	Да	Да[a)]	Да				Да[b)]
float	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
bitstring	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
octetstring	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
hexstring	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
character strings	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
record	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
record of	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
array	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
set	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
set of	Да	Да	Да	Да	Да	Да[a)]		Да	Да	Да	Да[b)]
enumerated	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
union	Да	Да	Да	Да	Да	Да[a)]					Да[b)]
anytype	Да	Да	Да	Да	Да	Да[a)]					Да[b)]

[a)] – При использовании должно применяться только к необязательным полям типов record и set (без ограничений на тип данного поля).

[b)] – При использовании должно применяться только к тем типам record и set (без ограничений на тип данного поля).

14.3.1 Ссылки на элементы шаблонов или поля шаблонов

14.3.1.1 Ссылки на отдельные элементы строк

В шаблонах или полях шаблонов не разрешается ссылаться на индивидуальные элементы строки.

ПРИМЕР:

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

t_Char2 := t_Char1[1];
// приводит к ошибке, так как не разрешается обращаться к отдельным элементам
// строки;
```

14.3.1.2 Ссылка на поля `record` и `set`

Как шаблоны, так и шаблонные переменные позволяют ссылаться на субполя в пределах определения шаблона с помощью обозначения точка. Однако поле, на которое указывает ссылка, может являться субполем структурированного поля, которому присвоен механизм сопоставления. Данный раздел содержит правила для подобных случаев.

- Omit, AnyValueOrNone, списки значений и дополняющие списки: ссылка на субполе в структурированном поле, которому присваивается значение **omit**, AnyValuerOf (*), список значений или дополняющий список, приводит к ошибке.

ПРИМЕР 1:

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

:
var template R1 t_R1 := {
    f1 := 5,
    f2 := omit
}
var template R2 t_R2 := t_R1.f2.g2;
    // приводит к ошибке, так как omit присваивается t_R1.f2
t_R1. f2 := *
t_R2 := t_R1.f2.g2;
    // приводит к ошибке, так как * присваивается t_R1.f2

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

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
// все эти присваивания приводят к ошибке, так как список значений
// присваивается t_R1

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

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
// все эти присваивания приводят к ошибке, так как дополняющий список
// присваивается t_R1
```

- AnyValue: при ссылке на субполе в структурированном поле, которому присвоено значение AnyValue (?), когда оно находится в правой части выражения присваивания, для обязательных суб-полей возвращается AnyValue (?), а для необязательных полей возвращается AnyValueOrNone.

  При ссылке на суб-поля в структурированном поле, которому присвоено AnyValue (?), когда она находится в левой части выражения, структурированное поле рекурсивно расширяется до глубины субполя, на которое указывает ссылка. В течение этого процесса обязательным суб-полям присваивается AnyValue (?), а необязательным полям присваивается AnyValueOrNone. После такого расширения значение или механизм согласования, который стоит в правой части выражения присваивания, присваивается субполю, на которое указывает ссылка.

  ПРИМЕР 2:

```
t_R1 := {f1:=0, f2:=?}
t_R2 := t_R1.f2.g2;
// после присваивания t_R2 будет равно {g1:=?, g2:=*}
t_R1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
// во-первых, поле t_R1.f2 предполагается расширить до
// {g1:=?,g2:={g1:=?,g2:=*}}
// после чего присваивание t_R1 будет иметь следующий вид:
//   {f1:=0, f2:={g1:=?,g2:={g1:=?,g2:=({g1:=1, g2:=omit},{g1:=2, g2:=omit})}}}
```

- Атрибут ifpresent: ссылка на субполе в структурированном поле, которому присвоен атрибут **ifpresent**, приводит в ошибке (независимо от значения или механизма сопоставления, к которому добавлен **ifpresent**)

14.3.1.3 Ссылка на элементы `record of` и `set of`

Как шаблоны, так и шаблонные переменные позволяют ссылаться на элементы шаблонов или полей **`record of`** или **`set of`** при помощи индексной нотации. Однако шаблону или полю, в котором находится элемент, на который указывает ссылка, может быть присвоен механизм сопоставления. Данный раздел содержит правила, которые рассматривают такой случай.

- Omit, AnyValueOf, списки значений, дополняющие списки, subset и superset: ссылка на элемент поля record of или set of, которому присвоено **omit**, AnyValueOrNone (*) с или без атрибута длины, список значений, дополняющий список, subset или superset, приводит к ошибке.

ПРИМЕР 1:

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

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoRoI := ({},{0},{0,0},{0,0,0});
t_RoI := t_RoRoI[0];
// приводит к ошибке, так как t_RoRoI присвоен список значений;
```

- AnyValue: при ссылке на элемент шаблона или поля **record of** или **set of**, которому присвоено AnyValue(?) (без атрибута длины), в правой части присваивания, должно возвращаться AnyValue(?). Если к AnyValue(?) прикреплен атрибут длины, то индекс ссылки не должен нарушать атрибут длины.

  При ссылке на элемент шаблона или поля **record of** или **set of**, которому присвоено AnyValue(?) (без атрибута длины), в левой части присваивания, элементу, на который указывает ссылка, должно быть присвоено значение или механизм сопоставления, который находится в правой части присваивания. Всем элементам, которые находятся перед элементом, на который указывает ссылка (если такой имеется), должно быть присвоено AnyElement(?), а в конце должен быть добавлен единственный AnyElementsOrNone(*). Когда к AnyValue(?) прикреплен атрибут длины, то этот атрибут прозрачно передается новому шаблону или полю. Во всех указанных выше случаях индекс не должен нарушать ограничения для типа.

  ПРИМЕР 2:

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

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoI := ?;
t_Int := t_RoI[5];
    // после присваивания t_Int будет равно AnyValue(?);

t_RoRoI := ?;
t_RoI := t_RoRoI[5];
    // после присваивания t_RoI будет равно AnyValue(?);
t_Int := t_RoRoI[5].[3];
    // после присваивания t_Int будет равно AnyValue(?);

t_RoI := ? length (2..5);
t_Int := t_RoI[3];
    // после присваивания t_Int будет равно AnyValue(?);
t_Int := t_RoI[5];
    // приведет к ошибке так как индекс ссылки находится вне атрибута длины
    // (заметим, что индекс 5 указывает на 6-ой элемент);

t_RoRoI[2] := {0,0};
    // после присваивания t_RoRoI будет равно {?,?,{0,0},*};
t_RoRoI[4] := {1,1};
    // после присваивания t_RoRoI будет равно {?,?,{0,0},?,{1,1},*};
t_RoI[0] := -5;
    // после присваивания t_RoI будет равно {-5,*}length(2..5);
t_RoI := ? length (2..5);
t_RoI[1] := 1;
    // после присваивания t_RoI будет равно {?,1,*}length(2..5);
t_RoI[3] := ?
    // после присваивания t_RoI будет равно {?,1,?,?,*}length(2..5);
t_RoI[5] := 5
    // после присваивания t_RoI будет равно {?,1,?,?,?,5,*}length(2..5);
    // заметим, что t_RoI становится пустым набором, однако это не
    // приводит к ошибке;
```

- Перестановка: при ссылке на шаблон или поле поля **record of** или **set of**, которое находится внутри перестановки (основываясь на его индексе), то это приводит к ошибке. Индексы элементов, над которыми производится перестановка, должны определяться на основании количества элементов перестановки. Если в качестве элемента перестановки используется AnyValueOrNone, то перестановка будет производиться над всеми индексами элементов **record of**.

ПРИМЕР 3:

```
t_RoI := {permutation(0,1,3,?),2,?}
t_Int := t_RoI[5];
    // после присваивания t_Int будет равно AnyValue(?)

t_RoI := {permutation(0,1,3,?),2,*}
t_Int := t_RoI[5];
    // после присваивания t_Int будет равно * (AnyValueOrNone)
t_Int := t_RoI[2];
    // приводит к ошибке, так как третий элемент (с индексом 2) находится
    // внутри перестановки

t_RoI := {permutation(0,1,3,*),2,?}
t_Int := t_RoI[5];
    // приводит к ошибке, так как перестановка содержит AnyValueOrNone(*),
    // которое может относиться к любым индексам record of
```

- Атрибут **Ifpresent**: если ссылке указывает на шаблон или поле поля **record of** или **set of**, которому прикреплен атрибут **ifpresent**, то это приводит к ошибке (независимо от значения или механизма сопоставления, к которому добавлен **ifpresent**).

14.4 Параметризация шаблонов

14.4.0 Общие положения

Шаблоны для операций как передачи, так и приема могут быть параметризованы. Действительные параметры шаблона могут содержать значения и шаблоны, функции и специальные символы сопоставления. Должны соблюдаться правила для списков формальных и реальных параметров, определенные в п. 5.2.

ПРИМЕР:

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

// может использоваться следующим образом
pco1.send(MyTemplate(123));
```

14.5 Утратило значение

14.6 Модифицированные шаблоны

14.6.0 Общие положения

Обычно шаблон определяет набор базовых, или безусловных (по умолчанию), значений либо символов сопоставления для каждого поля, указанного в соответствующем определении типа или сигнатуры. В тех случаях, когда для определения нового шаблона требуются небольшие изменения, можно определить модифицированный шаблон. Модифицированный шаблон указывает изменения для конкретных полей оригинального шаблона либо прямо, либо косвенно.

Ключевое слово **modifies** обозначает родительский шаблон, из которого будет образован новый, то есть модифицированный, шаблон. Этот родительский шаблон может быть либо оригинальным, либо модифицированным шаблоном.

Изменения, появляющиеся при некотором связанном способе, в конечном счете, ведут обратно к оригинальному шаблону. Если поле шаблона и его соответствующее значение или символ сопоставления указаны в модифицированном шаблоне, то указанное значение или символ сопоставления заменяет значение или символ, указанный в родительском шаблоне. Если поле шаблона и его соответствующее значение или символ сопоставления не указаны в модифицированном шаблоне, то должно использоваться значение или символ сопоставления из родительского шаблона. Если поле, которое должно модифицироваться, является вложенным в поле шаблона, которое само является структурированным полем, то не будут изменяться никакие поля структурированного поля за исключением того поля(-ей), которое обозначено явным образом.

Модифицированный шаблон не должен ссылаться на себя ни прямо, ни косвенно, то есть рекурсивное заимствование не разрешается.

ПРИМЕР 1:

```
// задано
type record MyRecordType
{
    integer field,
    charstring field2,
    boolean field3
}
```

```
template MyRecordType MyTemplate1 :=
{
  field1 := 123,
  field2 := "A string",
  field3 := true
}

// затем записывается
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
      field1 := omit,   // field1 является факультативным, однако присутствует
                        // в MyTemplate1
      field2 := "A modified string"
                        // field3 не изменяется
  }

// это то же, что и запись
template MyRecordType MyTemplate2 :=
{
  field1 := omit,
  field2 := "A modified string",
  field3 := true
}
```

Когда необходимо изменить индивидуальные значения модифицированного шаблона или поля модифицированного шаблона типа **record of**, и только в таких случаях, может также использоваться обозначение присваивания значения, где в левой части присваивания стоит индекс того элемента, который необходимо изменить.

ПРИМЕР 2:

```
template MyRecordOfType MyBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
template MyRecordOfType MyModifTemplate modifies MyBaseTemplate := { [2] := 3, [3] := 2 };
// MyModifTemplate должно соответствовать последовательности значений
// { 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }
```

14.6.1 Параметризация модифицированных шаблонов

Если базовый шаблон имеет список формальных параметров, то применяются следующие правила для всех модифицированных шаблонов, выведенных из этого базового шаблона, независимо от того, выведены они за один шаг модификации или за несколько:

a) в выведенном шаблоне не должны пропускаться параметры, определенные на любом из шагов модификации между базовым шаблоном и действительным модифицированным шаблоном;

b) производный шаблон может при желании иметь дополнительные (добавленные) параметры;

c) список формальных параметров для каждого модифицированного шаблона должен следовать за именем шаблона.

ПРИМЕР:

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

// затем может быть внесено изменение
  template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{ // field1 параметризовано в Template1 и остается таким в Template2
  field2 := "A modified string",
}
```

14.6.2 Шаблоны, модифицированные "инлайн"

Помимо создания явно именованных модифицированных шаблонов TTCN-3 позволяет определять "инлайновые" модифицированные шаблоны.

ПРИМЕР:

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

// Может использоваться для определения "инлайнового" модифицированного шаблона для Setup
pco1.send (modifies Setup := {field1:= 76});
```

14.7 Изменения полей шаблона

В операциях связи (например, **send**, **receive**, **call**, **getcall** и так далее) разрешается изменять поля шаблона только с помощью параметризации или с помощью "инлайновых" производных (выведенных) шаблонов. Действие этих изменений в значении поля шаблона не распространяется на последовательность шаблонов для соответствующего сеанса связи.

Нотация-точка *MyTemplateId.Fieldid* не должна использоваться для установки или запроса значений в шаблонах при событиях связи. Для этой цели используется символ "–>" (см. раздел 23).

14.8 Операция сопоставления

Операция **match** позволяет сравнивать значение какой-либо переменной или параметра с шаблоном. Эта операция выдает булево значение. Если тип шаблона и переменная не совместимы (см. 6.7), то операция выдает **false** (ЛОЖЬ). Если типы совместимы, то выдаваемое операцией значение указывает, соответствует ли значение переменной определенному шаблону.

ПРИМЕР:

```
template integer LessThan10 := (infinity..10);

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

   if(match(RxValue, LessThan10)) { … }
   // true если действительное значение Rxvalue менее 10 и false в противном
   // случае
}
```

14.9 Операция Valueof

Операция **valueof** позволяет присваивать значение, указанное в шаблоне, полям переменной. Эта переменная и шаблон должны быть совместимы по типу (см. п. 6.7), а каждое поле шаблона должно сводиться к одиночному значению.

ПРИМЕР:

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

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

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

15 Операторы

15.0 Общие положения

TTCN-3 поддерживает ряд предопределенных операторов, которые могут использоваться в элементах выражений TTCN-3. Предопределенные операторы делятся на семь категорий:

a) арифметические операторы;

b) операторы цепочки;

c) операторы отношения;

d) логические операторы;

e) побитовые операторы;

f) операторы сдвига;

g) операторы циклического сдвига.

Эти операторы перечислены в таблице 7.

Таблица 7/Z.140 – Список операторов TTCN-3

Категория	Оператор	Символ или ключевое слово
Арифметические операторы	сложение	**+**
	вычитание	**–**
	умножение	*****
	деление	**/**
	модуль	**mod**
	остаток	**rem**
Операторы цепочки	сцепление	**&**
Операторы отношения	равно	**= =**
	меньше чем	**<**
	больше чем	**>**
	не равно	**! =**
	больше чем или равно	**> =**
	меньше чем или равно	**< =**
Логические операторы	логическое "нет"	**not**
	логическое "и"	**and**
	логическое "или"	**or**
	логическое "исключающее или"	**xor**
Побитовые операторы	побитовое "нет"	**not4b**
	побитовое "и"	**and4b**
	побитовое "или"	**or4b**
	побитовое "исключающее или"	**xor4b**
Операторы сдвига	сдвиг влево	**< <**
	сдвиг вправо	**> >**
Операторы циклического сдвига	циклический сдвиг влево	**< @**
	циклический сдвиг вправо	**@ >**

Старшинство этих операторов показано в таблице 8. Перечисленные в одной строке этой таблицы операторы имеют одинаковое старшинство. Если в каком-либо выражении появляется более одного оператора одинакового старшинства, то операторы вычисляются слева направо. Для группирования операндов (компонентов операции) в выражениях могут использоваться круглые скобки, в таком случае выражение в круглых скобках имеет высший приоритет при вычислении.

Таблица 8/Z.140 – Старшинство операторов

Приоритет	Тип оператора	Оператор
Высший		**(…)**
	Унарный (одинарный)	**+, –**
	Бинарный (двоичный)	***, /, mod, rem**
	Бинарный	**+, –, &**
	Унарный	**not4b**
	Бинарный	**and4b**
	Бинарный	**xor4b**
	Бинарный	**or4b**
	Бинарный	**< <, > >, <@, @>**
	Бинарный	**<, >, <=, >=**
	Бинарный	**= =, !=**
	Унарный	**not**
	Бинарный	**and**
	Бинарный	**xor**
Низший	Бинарный	**or**

15.1 Арифметические операторы

Арифметические операторы представляют операции сложения, вычитания, умножения, деления, определения модуля и определения остатка. Операнды этих операторов должны быть типа **integer** (включая производные **от integer**) или **float** (включая производные от **float**), за исключением **mod** и **rem**, который должен использоваться только с типом **integer** (включая производные от **integer**).

При арифметической операции с типом **integer** результат будет иметь тип **integer**. При арифметической операции с типом **float** результат будет иметь тип **float**.

В случае, когда используется в качестве унарного оператора плюс (+) или минус (–), применяются, кроме того, правила для операндов. Результатом использования оператора "минус" будет отрицательное значение операнда, если он был положительным, и наоборот.

Результат выполнения операции деления (/) может быть двояким:

a) значения **integer** дают целую часть значения **integer**, получаемое делением первого **integer** на второе (то есть дробные части отбрасываются);

b) значения **float** дают значение **float**, получаемое делением первого **float** на второе (то есть дробные части не отбрасываются).

Операторы **rem** и **mod** производят вычисления над операндами типа **integer** и дают результат типа **integer**. Операции `x` **rem** `y` и `x` **mod** `y` вычисляют остаток, который остается от целочисленного деления `x` на `y`. Следовательно, они определены только для ненулевых операндов `y`. Для положительных `x` и `y` как `x` **rem** `y`, так и `x` **mod** `y` дают один и тот же результат, но для отрицательных аргументов они различаются.

Формально **mod** и **rem** определяются следующим образом:

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

Таблица 9 иллюстрирует различие между операторами **mod** и **rem**:

Таблица 9/Z.140 – Действие операторов mod и rem

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 Строковые операторы

Предопределенные строковые операторы выполняют сцепление (конкатенацию) значений, совместимых с типом "цепочка" (строка). Операцией является простое сцепление слева направо. Никакие формы арифметического сложения не подразумеваются. Типом результата является корневой тип операндов.

ПРИМЕР:

```
'1111'B & '0000'B & '1111'B дает '111100001111'B
```

15.3 Операторы отношения

Предопределенные операторы отношения представляют отношения равенства (==), "меньше чем" (<), "больше чем" (>), "не равно" (!=), "больше чем или равно" (>=), а также "меньше чем или равно" (<=). Операнды равенства и неравенства могут иметь произвольные типы, совместимые между собой, за исключением типа **enumerated** – в этом случае операнды должны являться экземплярами одного типа. Все остальные операторы отношения должны иметь операнды только типа **integer** (включая производные от **integer**) или **float** (включая производные от **float**) или же являться экземплярами одного и того же типа **enumerated**. Типом результата этих операций является **boolean**.

Два значения **charstring** или **universal charstring** равны в том и только в том случае, если для всех позиций располагающиеся в них символы одинаковы. Для значений типов **bitstring**, **hexstring** или **octetstring** применимо такое же правило равенства с тем исключением, что элементами, для которых должно выполняться равенство для каждой позиции, являются соответственно биты, шестнадцатеричные цифры или пары шестнадцатеричных цифр.

Два значения **record**, значения **set**, значения **record of** или **set of** являются равными в то и только в том случае, если их эффективные структуры значений являются совместимыми и значения всех соответствующих полей равны. Значения записи также могут сравниваться с значениями **record of,** а значения множества - со значениями **set of**. Во всех этих случаях такие же правила применяются при сравнении двух значений **record** или **set**.

ПРИМЕЧАНИЕ. – "Все поля" означает, что факультативные поля не присутствуют в действительном значении типа **record** должно рассматриваться как неопределенное значение. Такое поле может быть равно только пропущенному факультативному полю (которое также рассматривается как неопределенное значение) при сравнении со значением другого типа **record** или с элементом с неопределенным значением при сравнении со значением типа **record of**. Данный принцип также применяется тогда, когда сравниваются значения двух типов **set** или типа **set** и типа **set of**.

Два значения типов **union** равны в том и только в том случае, если типы значений выбранных полей являются совместимыми и действительные значения выбранных полей равны.

ПРИМЕР:

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

typeset SetB{
        integer b1  optional,
        integer b2  optional,
        integer b3  optional
        };

typeset SetC{
        integer c1  optional,
        integer c2  optional,
        };

typeset of integer  SetOf;

typeunion UniD  {
        integer d1,
        integer d2,
        };

typeunion UniE  {
        integer e1,
        integer e2,
        };

typeunion UniF  {
        integer f1,
        integer f2,
        boolean f3,
        };

// и
const       Set A   conSetA1 :=  { a1 := 0, a2 := omit, a3 := 2 };
// заметим, что порядок определения значений полей не имеет значения
const       SetB    conSetB1 :=  { b1 := 0, b3 := 2, b2 := omit };
const       SetB    conSetB2 :=  { b2 := 0, b3 := 2, b1 := omit };
const       SetC    conSetC1 :=  { c1 := 0, c2 :=2 };
const       SetOf    conSetOf1   :=  { 0, omit, 2 };
const       SetOf    conSetOf2   :=  { 0, 2 };
const       UniD    conUniD1 :=  { d1:= 0 };
const       UniE    conUniE1 :=  { e1:= 0 };
const       UniE    conUniE2;    :=  { e2:= 0 };
const       UniF    conUniF1;    :=  { f1:= 0 };

// тогда
conSetA1 == conSetB1;
// возвращает true
conSetA1 == conSetB2;
// возвращает false, так как ни a1, ни a2 не равны своим значениям
// ( соответствующий элемент не пропускается )
conSetA1 == conSetC1;
// возвращает false, так как эффективное значения структур SetA и SetC
// не являются совместимыми
conSetA1 == conSetOf1;
// возвращает true
conSetA1 == conSetOf2;
// возвращает false, так как значение пропущенного a2 равно 2,
// однако значние a3 является неопределенным
conSetC1 == conSetOf2;
// возвращает true
conUniD1 == conUniE1;
// возвращает true
```

```
conUniD1 == conUniE2;
// возвращает false, так как выбранное поле e2 не является аналогом поля d1 из
// UniD1
conUniD1 == conUniF1;
// возвращает false, так как эффективное значение структур UniD1 и UniF
// не являются совместимыми
```

15.4 Логические операторы

Предопределенные булевы (логические) операторы выполняют операции отрицания, логического "и", логического "или" или логического "исключающего или". Их операнды должны иметь тип **boolean**. Типом результата логических операций является **boolean**.

Логическое "нет" является унарным оператором, который выдает значение **true** (ИСТИНА), если его операнд имел значение **false** (ЛОЖЬ), и выдает значение **false**, если операнд имел значение **true**.

Логическое "и" выдает значение **true**, если оба операнда имеют значение **true**; в остальных случаях оно выдает значение **false**.

Логическое "или" выдает значение **true**, если по меньшей мере один из его операндов имеет значение true; оно выдает значение **false**, только если оба операнда имеют значение **false**.

Логическое "исключающее или" выдает значение **true**, если один из его операндов имеет значение **true**; оно выдает значение **false**, если оба операнда имеют значение **false** либо **true**.

Для булевых выражений используется короткая оценка схемы, то есть оценка операндов логических операций прекращается, если общий результат известен: если в операторе **and** для левого операнда получаем значение **false**, то правый аргумент не рассматривается и значение выражения принимается равным **false**. В случае оператора or если для левого аргумента получаем значение **true**, то значение правого аргумента не оценивается и результат всего выражения принимается равным **true**.

15.5 Побитовые операторы

Предопределенные побитовые операторы выполняют операции побитовое "нет", побитовое "и", побитовое "или" и побитовое "исключающее или". Эти операции называются **not4b, and4b, or4b** и **xor4b** соответственно.

ПРИМЕЧАНИЕ. – Следует читать "not for bit", "and for bit" и т. д.

Их операнды должны иметь тип **bitstring**, **hexstring** или **octetstring**. В случае операторов **and4b**, **or4b** и **xor4b** операнды должны иметь совместимый тип. Тип результата побитовых операторов должен иметь корневой тип операндов.

Побитовый унарный оператор **not4b** инвертирует значения индивидуальных битов его операнда. Для каждого бита такого операнда бит 1 устанавливается в 0, а бит 0 – в 1. То есть:

not4b '1'B дает '0'B
not4b '0'B дает '1'B

ПРИМЕР 1:

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

Побитовый оператор **and4b** принимает два операнда равной длины. Для каждой соответствующей позиции бита значением результата будет 1, если оба бита установлены в 1; в остальных случаях значением бита результата будет 0. То есть:

```
'1'B and4b '1'B дает  '1'B
'1'B and4b '0'B дает  '0'B
'0'B and4b '1'B дает  '0'B
'0'B and4b '0'B дает  '0'B
```

ПРИМЕР 2:

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

Побитовый оператор **or4b** принимает два операнда равной длины. Для каждой соответствующей позиции бита значение результата будет 0, если оба бита установлены в 0; в остальных случаях значением бита результата будет 1. То есть:

```
'1'B or4b '1'B дает  '1'B
'1'B or4b '0'B дает  '1'B
'0'B or4b '1'B дает  '1'B
'0'B or4b '0'B дает  '0'B
```

ПРИМЕР 3:

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

Побитовый оператор **xor4b** принимает два операнда равной длины. Для каждой соответствующей позиции бита значением результата будет 0, если оба бита установлены в 0 или в 1; в остальных случаях значением бита результата будет 1. То есть:

```
'1'B xor4b '1'B дает  '0'B
'0'B xor4b '0'B дает  '0'B
'0'B xor4b '1'B дает  '1'B
'1'B xor4b '0'B дает  '1'B
```

ПРИМЕР 4:

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

15.6 Операторы сдвига

Предопределенные операторы сдвига выполняют операции сдвига влево (<<) и сдвига вправо (>>). Их левый операнд должен иметь тип **bitstream**, **hexstring** или **octetstring**. Их правый операнд должен иметь тип **integer**. Тип результата этих операторов должен быть таким же, как у левого операнда.

Операторы сдвига действуют по-разному в зависимости от типа их левого операнда. Если типом левого операнда является:

a) **bitstring**, то применяемой единицей сдвига является 1 бит;

b) **hexstring**, то применяемой единицей сдвига является 1 шестнадцатеричная цифра;

c) **octetstring**, то применяемой единицей сдвига является 1 октет.

Оператор сдвига влево (<<) принимает два операнда. Он сдвигает левый операнд влево на число единиц сдвига, указанное правым операндом. Лишние единицы сдвига (биты, шестнадцатеричные цифры или октеты) отбрасываются. Для каждой единицы сдвига, сдвинутой влево, вводится нуль ('0'B, '0'H или '00'O, определяемый согласно типу левого операнда) с правой стороны левого операнда.

ПРИМЕР:

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

Оператор сдвига вправо (>>) принимает два операнда. Он сдвигает левый операнд вправо на число единиц сдвига, указанное правым операндом. Лишние единицы сдвига (биты, шестнадцатеричные цифры или октеты) отбрасываются. Для каждой единицы сдвига, сдвинутой вправо, вводится нуль ('0'B, '0'H или '00'O, определяемый согласно типу левого операнда) с левой стороны левого операнда.

ПРИМЕР:

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

15.7 Операторы циклического сдвига

Предопределенные операторы циклического сдвига выполняют операции циклического сдвига влево (<@) и циклического сдвига вправо (@>). Их левый операнд должен иметь тип **bitstream, hexstring, octetstring, charstring** или **universal charstring**. Их правый операнд должен иметь тип **integer**. Тип результата этих операторов должен быть таким же, как у левого операнда.

Операторы циклического сдвига действуют по-разному в зависимости от типа их левого операнда. Если типом левого операнда является:

a) **bitstring**, то применяемой единицей циклического сдвига является 1 бит;

b) **hexstring**, то применяемой единицей циклического сдвига является 1 шестнадцатеричная цифра;

c) **octetstring**, то применяемой единицей циклического сдвига является 1 октет;

d) **charstring** или **universal charstring**, то применяемой единицей циклического сдвига является один знак.

Оператор циклического сдвига влево (<@) принимает два операнда. Он циклически сдвигает левый операнд влево на число единиц сдвига, указанное правым операндом. Лишние единицы сдвига (биты, шестнадцатеричные цифры, октеты или знаки) снова вводятся в левый операнд с его правой стороны.

ПРИМЕР1:

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

Оператор циклического сдвига вправо (@>) принимает два операнда. Он циклически сдвигает левый операнд вправо на число единиц сдвига, указанное правым операндом. Лишние единицы сдвига (биты, шестнадцатеричные цифры, октеты или знаки) снова вводятся в левый операнд с его левой стороны.

ПРИМЕР 2:

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

16 Функции и altstep

В TTCN-3 функции и altstep используются для спецификации и структуризации тестового поведения, определения поведения по умолчанию, для структурирования поведения в модуле и так далее, как это описывается в последующих разделах.

16.1 Функции

16.1.0 Общие положения

Функции используются в TTCN-3 для выражения поведения теста, для организации выполнения тестов или для структурного вычисления в модуле, например, для подсчета одиночного значения, для инициализации набора переменных или для проверки некоторых условий. Функции могут выдавать (возвращать) некоторое значение. Возвращаемое значение обозначается ключевым словом **return**, за которым следует идентификатор типа. Возвращение шаблона обозначается ключевым словом **return template**, за которым следует идентификатор типа.

Ключевое слово **return**, когда оно использовано в теле функции и возвращаемое значение определено в ее заголовке, то оно должно всегда следовать за выражением, представляющим возвращаемое значение. Тип возвращаемого значения должен быть совместим с возвращаемым типом. Ключевое слово **return**, когда оно используется в теле функции и в заголовке функции определен возвращаемый шаблон, то за ним всегда должен следовать выражение или экземпляр шаблона, который представляет возвращаемый шаблон. Тип возвращаемого шаблона должен быть совместим с типом возвращаемого шаблона.

Выражение **return** в теле функции побуждает эту функцию завершить свою работу и возвратить возвращаемое значение в то место, где осуществлялся вызов функции.

ПРИМЕР 1:

```
// Определение функции MyFunction, не имеющей параметров
function MyFunction() return integer
{

  return 7; // Когда функция заканчивается, возвращается целочисленное значение 7
}

// Определение функций, которые могут возвращать символы сопоставления или
// шаблоны
  function MyFunction2() return template integer
  {
  :
      return ?;   // возвращает механизм сопоставления AnyValue
  }
  function MyFunction3() return template octetstring
  {
  :
      return "FF??FF"O;     // возвращает octetstring с AnyElement внутри
  }
```

Функция может быть определена внутри модуля или объявлена как определяемая извне (то есть внешняя). Для внешней функции в модуле TTCN-3 должен быть предусмотрен только интерфейс функции. Реализация внешней функции в данной Рекомендации не рассматривается. Внешние функции не должны содержать портовые операции. Внешним функциям не разрешается возвращать шаблоны.

```
external function MyFunction4() return integer;// Внешняя функция без
                                               // параметров, которая выдает
                                               // целочисленное значение

external function InitTestDevices();  // Внешняя функция, которая действует только
                                      // вне модуля TTCN-3
```

Поведение функции может быть определено в модуле с помощью программных команд и операций, описанных в разделе 18. Если функция использует переменные, константы, таймеры и порты, которые объявлены в определении типа компонента, то тип компонента указывается с помощью ключевого слова **runs on** в заголовке функции. Единственным исключением из этого правила будет случай, когда вся информация, доступная в пределах компонента, передается функции в виде параметров.

ПРИМЕР 2:

```
function MyFunction3() runs MyPTCType {
                      // MyFunction3 не возвращает значение, однако
var integer MyVar := 5;  // использует операцию порта send и следовательно
PCO1.send(MyVar);        // требует использования runs on для разрешения
                      // идентификатора порта с помощью ссылки на тип
                      // компонента
```

Функция без **runs on** никогда не должна вызывать функцию или altstep или активировать altstep с локальным выражением **runs on**.

Функции, которые запускаются при помощи операции над тестовыми компонентами **start** должны всегда иметь выражение **runs on** (см. 22.5) и считается, что они должны вызываться в том компоненте, в котором должны запускаться, то есть не локально. Тем не менее, операция тестового компонента **start** может вызываться в тех функциях, которые не имеют выражения **runs on**.

ПРИМЕЧАНИЕ. – Ограничения, которые касаются выражения **runs on**, относятся только к функциям и altstep, но не к тестовым примерам.

Функции, которые используются в управляющей части модуля TTCN-3, не должны содержать выражения **runs on**. Тем не менее, они могут выполнять тестовые примеры.

16.1.1 Параметризация функций

Функции могут быть параметризованы. Должны выполняться правила для списков формальных параметров, определенные в п. 5.2.

ПРИМЕР:

```
function MyFunction2(inout integer MyPar1)
{
                      // MyFunction 2 не выдает никакого значения,
  MyPar1 := 10 * MyPar1;   // но изменяет значение MyPar1, которое
}                     // передается по ссылке
```

16.1.2 Вызов функций

Функция вызывается путем указания ее имени, а также с помощью указания действительного списка параметров. Функции, которые не возвращают значений, могут вызываться прямо. Функции, которые возвращают значения, могут вызываться прямо или внутри выражений. Для списков реальных параметров должны выполняться правила, определенные в п. 5.2.

ПРИМЕР:

```
MyVar := MyFunction4();    // Значение, выданное функцией MyFunction4, присвоено
                        // MyVar. Типы выданного значения и MyVar
                        // должны быть совместимыми
```

```
MyFunction2(MyVar2);       // MyFunction2, которая не выдает значения, вызвана
                           // с реальным параметром MyVar2, который может быть
                           // передан по ссылке

MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3);// Функции, используемые в выражениях
```

К функциям применяются специальные ограничения, определяющие пределы для тестовых компонентов с помощью операции тестовых компонентов **start**. Эти ограничения описываются в п. 22.5.

16.1.3 Предопределенные функции

TTCN-3 содержит ряд предопределенных (встроенных) функций, которые не требуется объявлять перед использованием.

Таблица 10/Z.140 – Список предопределенных функций TTCN-3

Категория	Функция	Ключевое слово
Функции преобразования	Преобразовать значение **integer** в значение **charstring**	**int2char**
	Преобразовать значение **integer** в значение **universal charstring**	**int2unichar**
	Преобразовать значение **integer** в значение **bitistring**	**int2bit**
	Преобразовать значение **integer** в значение **hexstring**	**int2hex**
	Преобразовать значение **integer** в значение **octetstring**	**int2oct**
	Преобразовать значение **integer** в значение **charstring**	**int2str**
	Преобразовать значение **integer** в значение **float**	**int2float**
	Преобразовать значение **float** в значение **integer**	**float2int**
	Преобразовать значение **charstring** в значение **integer**	**char2int**
	Преобразовать значение **charstring** в значение **octetstring**	**char2oct**
	Преобразовать значение **universal charstring** в значение **integer**	**unichar2int**
	Преобразовать значение **bitstring** в значение **integer**	**bit2int**
	Преобразовать значение **bitstring** в значение **hexstring**	**bit2hex**
	Преобразовать значение **bitstring** в значение **octetstring**	**bit2oct**
	Преобразовать значение **bitstring** в значение **charstring**	**bit2str**
	Преобразовать значение **hexstring** в значение **integer**	**hex2int**
	Преобразовать значение **hexstring** в значение **bitstring**	**hex2bit**
	Преобразовать значение **hexstring** в значение **octetstring**	**hex2oct**
	Преобразовать значение **hexstring** в значение **charstring**	**hex2str**
	Преобразовать значение **octetstring** в значение **integer**	**oct2int**
	Преобразовать значение **octetstring** в значение **bitstring**	**oct2bit**
	Преобразовать значение **octetstring** в значение **hexstring**	**oct2hex**
	Преобразовать значение **octetstring** в значение **charstring**	**oct2str**
	Преобразовать значение **octetstring** в значение **charstring**	**oct2char**
	Преобразовать значение **charstring** в значение **integer**	**str2int**
	Преобразовать значение **charstring** в значение **octetstring**	**str2oct**

Таблица 10/Z.140 – Список предопределенных функций TTCN-3

Категория	Функция	Ключевое слово
	Преобразовать значение **charstring** в значение **float**	**str2float**
Функции длины/ размера	Выдать длину значения какого-либо типа **string**	**lengthof**
	Выдать число элементов в **record, record of, template, set, set** of или **array**	**sizeof**
	Возвращает количество элементов структурированного типа	**sizeoftype**
Функции присут-ствия/ выбора	Определить, присутствует ли факультативное поле в **record, record of, template, set** или **set of**	**Ispresent**
	Определить, какой выбор был сделан в типе **union**	**ischosen**
Функции присут-ствия/ выбора	Возвращает ту часть входной строки, которая совпадает с описанием заданного образца	**regexp**
	Возвращает заданную часть входной строки	**substr**
	Замещает фрагмент (подстроку) строки на сходную строку или вставляет входную строку	**replace**
Другие функции	Создать случайное число типа float	**rnd**

Когда вызывается предопределенная функция:

1) число реальных параметров должно быть таким же, как и число формальных параметров;

2) каждый реальный параметр должен определять элемент типа соответствующего формального параметра; а также

3) все переменные, появляющиеся в списке действительных параметров, должны быть ограниченными.

Полное описание предопределенных функций дается в Приложении C.

16.1.4 Ограничения на функции, вызываемые из специфических областей

Функции, которые возвращают значения, могут вызываться во время операций связи (в шаблонах, полях или in-line шаблонах) или во время мгновенной (snapshot) оценки (в Булевых ограничениях для **alt**-выражений или для altstep (см. 20.1.1) и в инициализации локальных определений altstep (см. 16.2.2)). Для того чтобы избежать побочных эффектов, которые приводят к изменению состояния компонента или действительного мгновенного снимка и для того, чтобы предотвратить появление различных результатов при последовательных оценках неизменяющегося мгновенного снимка, то не следует использовать следующие операции в функциях, вызываемых в описываемых выше случаях:

Все операции с компонентами, то есть **create**, **start** (для компонента), **stop** (для компонента), **kill**, **running** (для компонента), **alive**, **done** и **killed** (см. Примечания 1, 3, 4 и 6).

Все операции с портами, то есть **start** (для порта), **stop** (для порта), **halt, clear, send, receive, trigger, call, get**call, **reply, getreply, raise, catch, check, connect, map** (см. Примечания 1, 2, 3 и 6).

Операция action (см. Примечания 2 и 6).

Все операции с тайерами, то есть **start** (для таймера), **stop** (для таймера), **running** (для таймера), **read, timeout** (см. Примечания 4 и 6).

Вызов внешних функций (см. Примечания 4 и 6).

Вызов предопределенной функции rnd (см. Примечания 4 и 6).

Изменение компонентных переменных, то есть использование компонентных переменных в правой части присваиваний, а также в экземплярах параметров **out** и **inout** (см. Примечания 4 и 6).

Вызов операции **setverdict** (см. Примечания 4 и 6).

Активирование и деактивирование значений по умолчанию, то есть выражения **activate** и **deactivate** (см. Примечания 5 и 6).

Вызов функций с параметрами **out** или **inout** (см. Примечания 7 и 8).

ПРИМЕЧАНИЕ 1.– Элементы операций **start, stop, done, killed, halt, clear, receive, trigger, getcall, getreply, catch** и **check** могут вызывать изменения в текущем мгновенном снимке.

ПРИМЕЧАНИЕ 2.– Следует избегать использовать операции **send, call, reply, raise** и **action** по причине читаемости: то есть все связи должны быть явными, а не появляться в качестве побочного эффекта другой операции связи или оценки мгновенного снимка

ПРИМЕЧАНИЕ 3. – Следует избегать использовать операции **map, unmap, connect, disconnect, create** по причине читаемости: то есть все связи должны быть явными, а не появляться в качестве побочного эффекта другой операции связи или оценки мгновенного снимка.

ПРИМЕЧАНИЕ 4. – Следует избегать вызова внешних функций, **rnd, running, alive, read, setverdict** и избегать записи в компонентные переменные, так как это может привести к различным результатам при последовательной оценке одного и того же мгновенного снимка, например, это делает невозможным обнаружение блокировки (deadlock).

ПРИМЕЧАНИЕ 5. – Следует избегать использовать операции **activate** и **deactivate** по той причине, что они изменяют набор значений по умолчанию, который рассматривается при оценке текущего мгновенного снимка.

ПРИМЕЧАНИЕ 6. – Ограничения, за исключением ограничений на использование параметризации **out** или **inout**, должны применяться рекурсивно, то есть их запрещается использовать непосредственно или с помощью произвольно длинной цепочки вызовов функций.

ПРИМЕЧАНИЕ 7. – Ограничения на вызов функций с параметрами **out** или **inout** не применяются рекурсивно, то есть вызов функций, которые сами вызывают функции с параметрами out или inout, является разрешенным.

ПРИМЕЧАНИЕ 8. – Следует избегать использования параметров **out** или **inout,** так как это также может приводить к различным результатам при последовательной оценке одного и того же мгновенного снимка.

16.2 Altstep

16.2.0 Общие положения

TTCN-3 использует altstep для определения поведения по умолчанию или же для структурирования различных вариантов выражения **alt**. Altstep являются единицами контекста, схожими с функциями. Тело altstep определяет необязательный набор локальных определений и набор альтернативных вариантов, так называемый *top alternatives*, которые образуют тело altstep. Синтаксические правила для top alternatives идентичны синтаксическим правилам, которые применяются к альтернативным вариантам выражения **alt**.

Поведение altstep может быть определено при помощи выражений программирования и операций, которые приводятся в разделе 18. Если altstep содержит операции с портами или же использует компонентные переменные, константы или таймеры, то на соответствующий компонентный тип необходимо ссылаться с помощью ключевого слова **runs on** в заголовке altstep. Единственным исключением из этого правила является случай, когда все порты, переменные, константы и таймера, которые используются в altstep, передаются как параметры.

ПРИМЕР:

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

// определение Altstep при помощи PCO1, PCO2, MyIntVar и MyTimer для
// MyComponentType

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

```
    [] PCO2.receive {
         repeat
       }
    [] MyTimer.timeout {
    setverdict(fail);
         stop
       }
}
```

Altstep могут вызывать функции и altstep или активировать altstep по умолчанию. Если altstep не содержит выражения **runs on**, то она никогда не должна вызывать функцию или altstep по умолчанию с локальным выражением **runs on**.

16.2.1 Параметризация altstep

Altstep могут быть параметризованы. Altstep, которая активирована по умолчанию, должна иметь только параметры **in**, параметры портов, а также параметры таймеров. Если altstep вызывается только как альтернативный вариант в выражении **alt** или же как отдельное выражением в описании поведения TTCN-3, то может иметь параметры **in**, **out** и **inout**. Следует соблюдать правила для списка формальных параметров, которые определены в 5.2.

16.2.2 Локальные определения в altstep

16.2.2.0 Общие положения

Altstep могут определять локальные определения констант, переменных и таймеров. Локальные определения должны быть определены до набора альтернативных вариантов.

ПРИМЕР:

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

16.2.2.1 Ограничения на инициализацию локальных определений в altstep

Инициализация локальных определений при помощи вызова функций, возвращающих значение, может иметь побочные эффекты. Для избежания подобных побочных эффектов, которые приводят к несогласованности между действующим мгновенным снимком и состоянием компонента, а также для того, чтобы предотвратить получение различных результатов при последовательной оценке не изменяющегося мгновенного снимка, при инициализации локальных определений действуют ограничения, которые приводятся в 16.1.4.

16.2.3 Вызов altstep

Вызов altstep всегда связан с выражением **alt**. Вызов может осуществляться либо неявным образом с помощью механизма по умолчанию (см. раздел 21), или же явным образом с помощью непосредственного вызова в выражении **alt** (см. 20.1.6). Вызов altstep не приводит к новому мгновенному снимку, а оценка верхних альтернативных вариантов (top alternatives) для altstep производится с помощью действующего моментального снимка выражения **alt**, из которого вызывалась altstep.

ПРИМЕЧАНИЕ. – Новый мгновенный снимок в altstep будет сделан, если в выбранных верхних альтернативных вариантах определить и войти в новое выражение **alt**.

Для неявного вызова altstep при помощи механизма по умолчанию, altstep должна быть активирована по умолчанию с помощью выражения **activate** до того момента, когда будет достигнуто место вызова.

ПРИМЕР 1:

```
 :
var default MyDefVarTwo := activate(MySecondAltStep());   // активация altstep
                                                          // по умолчанию
 :
```

Явный вызов altstep из выражения alt выглядит как вызов функции в качестве альтернативного варианта.

ПРИМЕР 2:

```
 :
alt {
    [] PCO3.receive {
     …
     }
    [] AnotherAltStep(); // явный вызов altstep AnotherAltStep в качестве
                         // альтернативного варианта в выражении alt
    [] MyTimer.timeout {}
}
```

Когда altstep вызывается явным образом из выражения **alt**, то следующим альтернативным вариантом, который следует проверить, является первый альтернативный вариант **altstep**. Альтернативные варианты **altstep** проверяются и выполняются таким же образом, как и альтернативные варианты выражения **alt** (см. 20.1) за тем исключением, что при входе в **altstep** ене выполняется новый мгновенный снимок. Неудачное завершение altstep (то есть были проверены все верхние альтернативные варианты altstep и не было найдено соответствующего ответвления) приводит к оценке следующего альтернативного варианта или вызову механизма по умолчанию (если явный вызов является последним альтернативным вариантом в выражении **alt**). Удачное завершение может приводить либо к завершению тестового компонента, то есть **altstep** завершается при помощи выражения **stop**, или же получению нового мгновенного снимка и повторной оценке выражения **alt**, то есть **altstep** завершается **repeat** (см. 20.2) или продолжается непосредственно после выражения **alt**, то есть выбранный верхний альтернативный вариант **altstep** завершается без явного вызова **repeat** или **stop**.

Также в описании поведения в TTCN-3 **altstep** может вызываться как отдельное выражение. В этом случае вызов **altstep** может интерпретироваться как сокращенная запись выражения **alt**, в котором имеется только один альтернативный вариант, описывающий явный вызов **altstep**.

ПРИМЕР 3:

```
// выражение
AnotherAltStep();    // предполагается, что AnotherAltStep является корректно
                  // определенной altstep

// является сокращением для

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

16.3 Функции и altstep для различных компонентных типов

См. 6.7.3.

17 Тестовые примеры

17.0 Общие положения

Тестовые примеры являются специальным видом функций. В управляющей части модуля команда **execute** используется для запуска тестовых примеров (см. п. 27.1). Результатом выполненного тестового примера всегда является значение типа **verdicttype**. Каждый тестовый пример содержит один и только один MTC, тип которого указывается в заголовке определения этого тестового примера. Поведение, определенное в теле тестового примера, является поведением MTC.

Когда вызывается тестовый пример, создается MTC, конкретизируются порты MTC и интерфейс тестовой системы, и в этом MTC запускается поведение, указанное в определении тестового примера. Все эти действия должны выполняться неявно, то есть без явных операций **create** и **start**.

Чтобы обеспечить информацию, позволяющую выполнить эти неявные операции, в заголовке тестового примера предусматриваются две части:

a) часть "интерфейс" (обязательная): обозначается ключевым словом **runs on**, указывает необходимый тип компонента для МТС и делает связанные имена портов видимыми в пределах поведения МТС; а также

b) часть "тестовая система" (факультативная): обозначается ключевым словом **system** и указывает тип компонента, который определяет необходимые порты для интерфейса тестовой системы. Часть "тестовая система" опускается лишь в том случае, когда во время выполнения теста конкретизируется только МТС. В этом случае тип МТС неявно определяет порты в интерфейсе тестовой системы.

ПРИМЕР:

```
      testcase MyTestCaseOne()
      runs on MyMtcType1      // определяет тип МТС
system  MyTestSystemType // делает имена портов в интерфейсе тестовой системы
                              // видимыми для МТС
      {
         :   // Поведение, определенное здесь, выполняется в МТС при вызове
             // этого тестового примера
      }

      // либо тестовый пример, в котором конкретизируется только МТС
      testcase MyTestCaseTwo() runs on MyMtcType2
      {
         :   // Поведение, определенное здесь, выполняется в МТС при вызове
             // этого тестового примера
      }
```

17.1 Параметризация тестовых примеров

Тестовые примеры могут быть параметризованы. Необходимо соблюдать правила для списков яормальных параметров, которые определяются в 5.2.

18 Обзор программных команд и операций

Основными программными элементами тестовых примеров, функций, altstep и управляющей части модулей TTCN-3 являются выражения, базовые программные команды, такие как присвоения, циклические конструкции и др., команды поведения, такие как последовательное поведение, альтернативное поведение, перемежение, по умолчанию и др., а также операции, такие как **send, receive, create** и др.

Команды могут быть либо одиночными (которые не содержат других программных команд), либо составными (которые могут включать другие команды, блоки команда и объявления).

Команды должны выполняться в порядке их появления, то есть последовательно, как это показано на Рисунке 8.
.

Рисунок 8/Z.140 – Иллюстрация последовательного поведения

Индивидуальные команды в последовательности должны разделяться с помощью разделителя ";".

ПРИМЕР:

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

Разрешается определение пустого блока команд и объявлений, то есть { }, может использоваться в сложных командах, то есть ветвь в выражении **alt**, и это означает, что никакие действия не выполняются.

Таблица 11/Z.140 – Обзор команд, выражений и операций TTCN-3

Команда	Связанное ключевое слово или символ	Может использоваться в управлении модулем	Может использоваться в функциях, тестовых примерах и altstep	Может использоваться в функциях, вызываемых из шаблонов, Булевых ограничений, или из инициализаций локальных определений altstep
Выражения	`( . . . )`	Да	Да	Да
Базовые программные команды				
Присвоения	`: =`	Да	Да	Да (см. Примечание 3)
Регистрация	`log`	Да	Да	Да
Label и Goto (Метка и перейти к)	`label/goto`	Да	Да	Да
Если - еще If-else (-)	`if ( . . . ) { . . . }` `else { . . . }`	Да	Да	Да
Для цикла	`for ( . . . ) { . . . }`	Да	Да	Да
Во время цикла	`while ( . . . ) { . . . }`	Да	Да	Да
Делать во время цикла	`do { . . . }` `while ( . . . )`	Да	Да	Да
Остановить выполнение	`stop`	Да	Да	
Select case	`select case {…}` `{ case {…} {…}` `case else {…}}`	Да	Да	Да
Программные команды поведения				
Альтернативное поведение	`alt { … }`	Да (см. Примечание 1)	Да	
Повторная оценка альтернативного поведения	`repeat`	Да (см. Примечание 1)	Да	
Перемежающееся поведение	`interleave { . . . }`	Да (см. Примечание 1)	Да	
Возвращение управления	`return`		Да (см. Примечание 4)	Да
Команды для обработки по умолчанию				
Активировать элемент по умолчанию	`activate`	Да (Примечание 1)	Да	
Деактивировать элемент по умолчанию	`deactivate`	Да (Примечание 1)	Да	
Операции конфигурации				
Создать параллельный тестовый компонент	`create`		Да	
Соединить порт компонента с другим портом компонента	`connect`		Да	
Разъединить два порта компонента	`disconnect`		Да	
Отобразить порт в тестовый интерфейс	`map`		Да	
Убрать отображение порта из интерфейса тестовой	`unmap`		Да	

системы				
Взять значение ссылки на компонент MTC	`mtc`		Да	Да
Взять значение ссылки на компонент интерфейса тестовой системы	`system`		Да	Да
Взять собственный значение ссылки на компонент	`self`		Да	Да
Запустить выполнение поведения тестового компонента	`start`		Да	
Остановить выполнение поведения тестового компонента	`stop`		Да	
Удалить тестовый компонент из системы	`kill`		Да	
Проверить окончание поведения PTC	`running`		Да	
Проверить существование PTC в тестовой системе	`alive`		Да	
Ожидать окончания PTC	`done`		Да	
Ожидать прекращения существования PTC	`killed`		Да	
Операции связи				
Передать сообщение	`send`		Да	
Обратиться к вызову процедуры	`call`		Да	

Таблица 10/Z.140 – Обзор команд и операций TTCN-3

Команда	**Связанное ключевое слово или символ**	**Может использоваться в управлении модулем**	**Может использоваться в функциях, тестовых примерах и именованных альтернативах**	
Ответить на вызов процедуры от удаленного объекта	`reply`		Да	
Породить особое состояние (для принятого вызова)	`raise`		Да	
Получить сообщение	`receive`		Да	
Запустить сообщение	`trigger`		Да	
Принять вызов процедуры от удаленного объекта	`getcall`		Да	
Обработать ответ на предыдущий вызов	`getreply`		Да	
Уловить особое состояние (от вызываемого объекта)	`catch`		Да	
Проверить (текущее) сообщение/принятый вызов	`check`		Да	
Очистить очередь порта	`clear`		Да	
Очистить очередь и разрешить передачу и прием для порта	`start`		Да	
Прекратить операции передачи и приема по согласованию для порта	`stop`		Да	
Запретить операции передачу и прием по согласованию новых сообщений/вызовов	`halt`		Да	
Таймерные операции				
Запустить таймер	`start`	Да	Да	
Остановить таймер	`stop`	Да	Да	
Считать истекшее время	`read`	Да	Да	
Проверить, считает ли таймер	`running`	Да	Да	

Событие тайм-аута	**timeout**	Да	Да	
Операции вердикта				
Установить местный вердикт	**setverdict**		Да	
Взять местный вердикт	**getverdict**		Да	Да
Внешние операции				
Симулировать (SUT) действие внешним образом	**action**	Да	Да	
Выполнение тестовых примеров				
Выполнить тестовый пример	**execute**	Да	Да (см. Примечание 2)	
ПРИМЕЧАНИЕ 1. – Может использоваться только для управления таймерными операциями. ПРИМЕЧАНИЕ 2. – Может использоваться только в функциях и altstep, которые используются в управлении модулем. ПРИМЕЧАНИЕ 3. – Не разрешается изменение компонентных переменных. ПРИМЕЧАНИЕ 4. – Может использоваться в функциях или altstep, но не может использоваться в тестовых примерах.				

19 Выражения и основные команды программ

19.0 Общие положения

Выражения и базовые программные команды могут использоваться в управляющей части модуля, в функциях, altstep и тестовых вариантах TTCN-3.

Таблица 12a/Z.140 - Обзор базовых программных команд TTCN-3

Базовые программные команды	
Команда	**Связанное ключевое слово или символ**
Присвоения	**: =**
Регистрация	**Log**
Label и Goto	**label/goto**
If-else	**if (…) { … } else { … }**
Для цикла	**for (…) { … }**
Во время цикла	**while (…) { … }**
Делать во время цикла	**do { … } while (…)**
Остановить выполнение	**Stop**
Выбрать вариант	**select case (…) { case (…) {…} case else {…} }**

19.1 Выражения

19.1.0 Общие положения

TTCN-3 позволяет определять выражения с помощью операторов, указанных в разделе 15. Выражения создаются из других (простых) выражений. Выражения могут использовать только значения, возвращённые функциями. Результатом выражения должно быть значение определенного типа, причем используемые операторы должны быть совместимы с типом операндов.

ПРИМЕР:

```
(x + y – increment (z))*3;
```

19.1.1 Булевы выражения

Булевы выражения должны содержать только булевы значения и/или булевы (логические) операторы, и/или операторы отношения, а также должны выражать булево значение в виде **true** (ИСТИНА) или **false** (ЛОЖЬ).

ПРИМЕР:

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

19.2 Присвоения

Переменным могут присваиваться значения. Это указывается символом ":=". Во время выполнения присвоения правая сторона присвоения должна выражать значение,совместимое с типом, указанном в левой стороне. Действие присвоения есть присвоение переменной значение выражения. Присвоение не должно содержать несвязанных переменных. Все присвоения происходят в порядке, в котором они появляются, то есть они обрабатываются слева направо.

ПРИМЕР:

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

19.3 Команда Log

Команда **log** обеспечивает средства для записи одного или более пунктов журнала в каком-либо регистрирующем устройстве, связанном с управлением тестом или тестовым компонентом, в котором эта команда используется. Пункты, которые будут записаны, должны быть определены списком аргументов команды Log, которые разделяются между собой запятыми. Пунктами журнала могут быть индивидуальные элементы языка, указанные в таблице 12b, или выражениях, составленных из таких пунктов .

Настоятельно рекомендуется, чтобы исполнение команды log не влияло на тестовое поведение. В частности функции, используемые в записях журнала, не должны явным или неявным образом изменять переменные значения компонентов, статус порта или статус таймера, и не должны изменять значение любого из ее параметров **inout** или **out**.

ПРИМЕР:

```
var integer myVar:= 1;
log ("Line 248 in PTC_A:  ", myVar, " (actual value of myVar)");
// Строка "Line 248 in PTC_A" записывается в некоторое
// регистрирующее устройство тестовой системы
```

ПРИМЕЧАНИЕ 1 – Функции, используемые в командах log не должны ни прямо, ни косвенно использовать выражения **if…else**, **for**, **while**, **do…while**, **label**, **goto**, **return**, **mtc**, **system**, **self**, **running** (PTC или таймер), **read** и **getverdict**

ПРИМЕЧАНИЕ 2. – В настоящей Рекомендации не рассматриваются комплексные возможности регистрации и трассировки, которые могут зависеть от инструментальных средств.

Таблица 12b/Z.140 –TTCN-3 Элементы языка, которые могут быть записаны в журнал

Использовано в формулировке журнала	Что записано	Комментарий
идентификатор параметра модуля	фактическое значение	
буквенное значение	значение	Это также включает свободный текст.
идентификатор константы данных	фактическое значение	
идентификатор внешней константы	фактическое значение	
экземпляр шаблона	фактический шаблон или поле значений и символы сопоставления	
переменный идентификатор типа данных	фактическое значение или «неинициализировано»	Смотри Примечания 3 и 4
переменный идентификатор **self, mtc, system** или типа компонента	фактическое значение и ,если задано,	По записи фактического значения

	имя экземпляра компонента или "неинициализировано"	смотри примечания 2-4. Фактическое состояние компонента должно быть записано в соответствии с Примечанием 5
выполняемая операция (компонент или таймер)	возвращаемое значение	**`true`** или **`false.`** В случае массива компонента или таймеров, должно быть включено описание элемента вектора
действующая операция (компонент)	возвращаемое значение	**`true`** или **`false`**. В случае массива должно быть включено описание элемента массива
экземпляр порта	фактическое состояние	Состояние порта должно быть записано в соответствии с Примечанием 6.
переменный идентификатор типа по умолчанию	фактическое состояние или "неинициализированно"	Состояния по умолчанию должны быть записаны в соответствии с Примечанием 7, см. также примечания 2-4.
имя таймера	фактическое состояние	Состояние таймера должно быть записано в соответствии с Примечанием 8.
операция чтения	возвращаемое значение	Смотри 24,3
преопределенные функции	возвращаемое значение	См. Примечание C.
экземпляр функции	возвращаемое значение	Разрешены только функции с условием возвращения.
экземпляр внешней функции	возвращаемое значение	Разрешены только внешние функции с условием возвращения.
идентификатор формального параметра	Смотри комментарий	Регистрация фактических параметров должна следовать правилам, указанными для элементов языка, которые они заменяют. Будет записано: в случае параметров величины фактическая величина параметра; в случае параметров типа шаблона фактический шаблон или поле величин и согласованные символы; в случае параметров типа компонента фактическая ссылка компонента, и т. д.; Разрешается также использовать для параметров таймера операции чтения, а для типа компонента и параметров таймера операция выполнения.

Примечание 1. – Фактическое значение/шаблон является значением/шаблоном в момент выполнения команды log.

Примечание 2. – Тип зарегистрированного значения зависит от инструмента.

Примечание 3. –В случае идентификаторов массива без спецификации элемента массива, должны быть зарегистрированы фактические значения и имена соответствующих компонентов всех элементов массива.

Примечание 4. – Строка "неинициализированно"должна быть записана только если пункт журнала свободен (неинициализирован).

Примечание 5. – Состояния компонента, которые могут быть зарегистрированы: Inactive (Бездействие), Running (Работа), Stopped (Остановка) и Killed (Убит) (подробная информация приводится в Приложении F).

Примечание 6. – Состояние порта, которое может быть зарегистрировано: Started (Запущен) и Stopped (Остановлен)

(подробная информация приводится в Приложении F).

Примечание 7. – Состояние по умолчанию, которое может быть зарегистрировано: Activated (Активирован) и Deactivated (Дезактивирован).

Примечание 8. – Состояния таймера, которое могут быть зарегистрированы: Inactive (Бездействует), Running (Работает) и Expired (Переполнен) (для более точной информации, см. Приложение F).

19.4 Команда Label

Команда **label** позволяет определять метки в тестовых примерах, функциях, altstep и в управляющей части модуля. Подобно другим программным командам поведения TTCN-3 команда **label** может свободно использоваться , в соответствии с синтаксическими правилами, определенными в Приложении A. Она может использоваться до или после любой команды TTCN-3, но не в качестве первой команды в альтернативе или верхней альтернативе команды **alt**, **interleave** или **altstep.** Метки, использованные после ключевого слова **label,** должны быть уникальны среди меток определённых в этом тестовом примере, функции altstep или управляющей части.

ПРИМЕР:

```
label MyLabel;                // Определение метки MyLabel

// Метки L1, L2 and L3 определены в следующем фрагменте кода TTCN-3:
label L1;                   // Определение метки L1
alt{
[]  PCO1.receive(MySig1)
    {    label L2;                // Определение метки L2
     PCO1.send(MySig2);
     PCO1.receive(MySig3)
    }
[]  PCO2.receive(MySig4)
    {    PCO2.send(MySig5);
     PCO2.send(MySig6);
     label L3;               // Определение метки L3
     PCO2.receive(MySig7);
    }
}
 :
```

19.5 Команда Goto

Команда **goto** может использоваться в функциях, тестовых примерах, altstep и в управляющей части модуля TTCN-3. Команда **goto** выполняет переход к **label** .

Оператор Goto обеспечивает возможность свободно переходить, то есть, вперед и назад, в пределах последовательности операторов, перемещаться из единого составного оператора (например, цикла **while**) и перескакивать через несколько уровней из вложенных составных выражений (например, вложенные альтернативы). Однако использование оператора **goto** должно быть ограничено следующими правилами:

a) Запрещено переходить из или в функции, тестовые примеры, altstep и управляющие части TTCN-3 модуля.

b) Запрещено переходить в последовательность выражений, определенных в составном выражении (то есть выражения **alt**, цикл **while**, цикл **for**, выражение **if-else**, цикл **do-while** и выражение **interleave**).

c) Запрещено использовать оператор goto в пределах команды **interleave**.

ПРИМЕР:

```
// Следующий фрагмент кода TTCN-3 включает
 :
```

```
label L1;                    // … определение метки L1,
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }        // … переход назад к L1,
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; } // … переход назад к L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                          // … определение метки L2,
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
    label L3;                      // … определение метки L3,
    PCO2.send(MyVar);
    alt {
      [] PCO1.receive { }
      [] PCO2.receive(integer: 90) {
         PCO2.send(integer: 33);
         PCO2.receive(integer: 13);
         goto L4;                      // … выход из двух вложенных
                                   // альтернативных выражений,
         }
      [] PCO2.receive(MyError) {
         goto L3;                      // … переход назад из текущего
                                   // альтернативного выражения,
         }
      [] any port.receive {
         goto L2;                      // … переход назад из двух вложенных
                                   // альтернативных выражений,
         }
    }
     }
  [] any port.receive {
    goto L2;     // … и больший переход назад из альтернативного выражения.
     }
}
label L4;
 :
```

19.6 Команда If-else

Команда **`if-else`**, называемая также "условной командой", используется для указания на ветвление в потоке управления благодаря булевым выражениям. Схематически условность выглядит следующим образом:

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

где `statementblock`$_x$ указывает на блок команд.

ПРИМЕР:

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

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

Возможна более сложная схема:

```
if (expression1)
    statementblock1
else if (expression2)
    statementblock2
:
else if (expressionn)
    statementblockn
else
```

```
statementblockn+1
```

В таких случаях удобочитаемость в значительной степени зависит от форматирования, однако форматирование не имеет синтаксического или семантического смысла.

19.7 Команда For

Команда **for** определяет цикл счетчика. Значение переменной индекса увеличивается, уменьшается или изменяется так, чтобы после определенного числа циклов выполнения был достигнут критерий окончания.

Команда **for** содержит два присвоения и булево выражение. Первое присвоение необходимо для инициализации переменной индекса (или счетчика) цикла. Булево выражение заканчивает цикл, а второе присвоение используется для изменения переменной индекса.

ПРИМЕР 1:

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

Критерий окончания цикла указывается булевым выражением. Он проверяется в начале каждой новой циклической итерации. Если он имеет значение **true** (ИСТИНА), то выполнение продолжается с блоком команд в выражении **for** , если **false** (ЛОЖЬ) то выполнение продолжается с команды, которая непосредственно следует за циклом **for**.

Индексная переменная цикла **for** может быть объявлена до ее использования в команде **for** либо объявлена и инициализована в заголовке команды **for**. Если индексная переменная объявлена и инициализована в заголовке команды **for**, то контекст этой индексной переменной ограничивается телом цикла, то есть она видна только внутри тела цикла.

ПРИМЕР 2:

```
var integer j;                              // Объявление целочисленной переменной j
for (j:=1; j<=10; j:= j+1) { … }               // Использование переменной j в качестве
                                        // индексной переменной цикла for

for (var float i:=1.0; i<7.9; i:= i*1.35) { … }   // Индексная переменная i
                                        // объявлена и инициализована в заголовке
                                        // цикла for. Переменная i видна только в
                                        // теле этого цикла.
```

19.8 Команда While

Цикл **while** выполняется, пока удерживается состояние "loop". Состояние "loop" проверяется в начале каждой новой циклической итерации. Если состояние "loop" не удерживается, то цикл завершается, а выполнение будет продолжаться с командой, которая тотчас повторяет цикл **while**.

ПРИМЕР:

```
while (j<10) { … }
```

19.9 Команда Do-while

Цикл **do-while** идентичен циклу **while,** за исключением того, что состояние "loop" проверяется в конце каждой циклической итерации. Это означает, что при использовании цикла **do-while** режим выполняется по крайней мере один раз до того, как состояние "loop" будет оценено в первый раз.

ПРИМЕР:

```
do { …} while (j < 10);
```

19.10 Команда Stop для выполняемой операции

Команда **stop** останавливает выполнение различными способами в зависимости от контекста, в котором она используется. Когда эта команда используется в управляющей части модуля или в функции, используемой в управляющей части модуля, она останавливает функционирование всей управляющей части модуля. Когда она используется в тестовом варианте, altstep или функции, которая выполняет тестовый компонент, она останавливает соответствующий тестовый компонент.

ПРИМЕР:

```
module MyModule {
     :   // Module definitions
    testcase MyTestCase() runs on MyMTCType system MySystemType{
     var MyPTCType ptc:= MyPTCType.create;// создание PTC
     ptc.start(MyFunction());              // начало выполнения PTC
      :       // продолжается поведение для тестового варианта
     stop  // Остановка MTC, всез PTCs ицеликом тестового варианта
    }
    function MyFunction() runs on MyPTCType {
      :
     stop  // Остановка только PTC, тестовый вариант продолжается
    }
    control {
      :  // тест выполняется
     stop// остановка тестовой операции
    } // конец управления
} // конец модуля
```

Примечание. – Семантика команды **stop**, которым заканчивает тестовый компонент, идентична действию компонента остановки **self.stop** (см. 22.6).

19.11 Команда Select Case

Команда **select case** – альтернатива использованию **if .. else** при сравнении значений с одним или несколькими другими. Команда содержит заголовок и ноль или большее количество ветвлений. Выполняется не более одной ветки. Схематично, выражение **select case** выглядит следующим образом:

```
select (expression)
   {
    case (templateInstance1a, templateInstance1b,…)
     statementblock1
    case (templateInstance2a, templateInstance2b,…)
     statementblock2
    case else
     statementblock3
   }
```

где templateInstance относится к определённому или in-line шаблону $\texttt{statementblock}_{\texttt{X}}$ привязан к блоку команд.

Примечание. – Вышеприведённая схема эквивалентена схеме, использующей **if-else** команду:

```
if (match(expression, templateInstance1a or match(expression, templateInstance1b
or …)
    statementblock1
else if (match(expression, templateInstance2a or match(expression, templateInstance2b or …)
    statementblock2
else
    statementblock3
```

В заголовке команды **select case** должно быть дано выражение. Каждая ветвь начинается ключевым словом **case**, сопровождаемым списком **templateInstance** (ветвь списка, который может кроме того содержать единичный элемент) или ключевым словом **else** (else ветвление) и блок выражений.

Все **templateInstance** во всех ветвях списка должны иметь тип, совместимый с типом выражения в заголовке. Выбирается ветвь списка и выполняется блок выражений отобранной ветви только тогда, когда какой-либо из **templateInstance** соответствует значению выражения в заголовке выражения. После выполнения блока выражений отобранной ветви (а именно, не переходящий по команде go to) выполнение продолжается с выражения после команды **select case.**

Блок выражений else ветви выполняется всегда, если только не была выбрана какая-то другая ветвь, предшествующая в тексте ветви else.

Ветви оцениваются в порядке их следования в тексте. Если ни один из **templateInstance** не соответствует значению выражения в заголовке, и выражение не содержит ветвь **else**, выполнение продолжается без выполнения какой-либо из ветвей **select case**.

ПРИМЕР:

```
select (MyModulePar)  // Где MyModulePar является типом charstring
{
    case ("firstValue")
     {
      log ("The first branch is selected");
     }
    case (MyCharVar, MyCharConst)
     {
      log ("The second branch is selected");
     }
    case else
     {
      log ("The value of the module parameter MyModulePar is selected");
     }
}
```

20 Программные команды поведения

20.0 Общие положения

Программные команды поведения могут использоваться в тестовых примерах, функциях и altstep, за исключением:

a) команды **return**, которая используется только в функциях; и

b) команд **alt, interleave** и **repeat,** которые могут также использоваться в управлении модулём.

Программные команды поведения описывают динамический режим тестовых компонентов в порту связи. Поведение теста может быть выражено последовательно или в виде набора альтернатив или комбинации того и другого. Оператор перемежения позволяет описывать перемежающиеся последовательность или альтернативы.

Таблица 13/Z.140 – Обзор программных команд поведения TTCN-3

a) Программные команды поведения	
i. Команда	**Связанное ключевое слово или символ**
Альтернативное поведение	**alt { … }**
Повторная оценка команды **alt**	**repeat**
Перемежающееся поведение	**interleave { …}**
Управление выдаваемыми результатами	**return**

20.1 Альтернативное поведение

20.1.0 Общие положения

При более сложной форме поведения последовательности команд выражаются в виде наборов возможных альтернатив, образующих дерево путей выполнения, как показано на рис. 9.

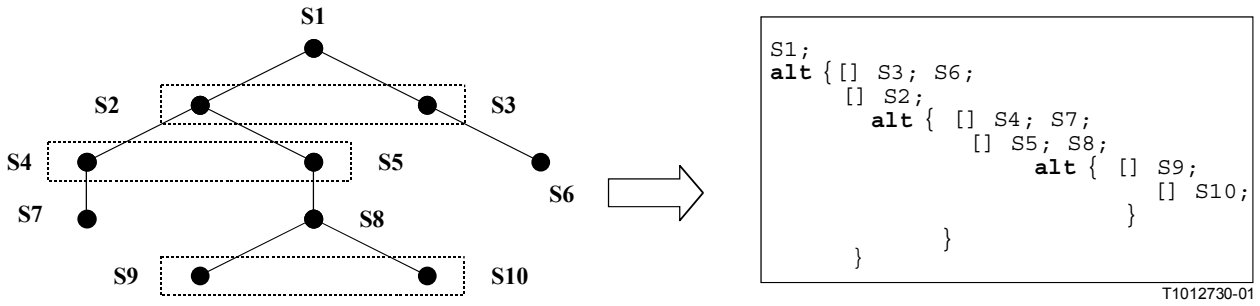


Рисунок 9/Z.140 – Иллюстрация альтернативного поведения

Команда **alt** обозначает ветвление тестового поведения вследствие приема и обработки событий связи, и/или таймера, и/или окончания параллельных тестовых компонентов, то есть она относится к использованию операций TTCN-3 **receive, trigger, getcall, getreply, catch, check, timeout, done и killed**. Команда **alt** обозначает набор возможных событий, которые должны быть сопоставлены с конкретным "снимком" (стоп-кадром) (см. п. 20.1.1).

ПРИМЕР:

```
// Использование вложенных альтернативных команд
:
alt
{
[]  L1.receive(DL_REL_CO:*)
{   setverdict .set(pass);
    TAC.stop;
    TNOAC.start;
    alt {
    []   L1.receive(t DL_EST_IN)
     {   TNOAC.stop;
         setverdict.set(pass);
     }
    []   TNOAC.timeout
     {   L1.send(DEL_EST_RQ);
         TAC.start;
         alt {
```

```
        []  L1.receive(DL_EST_CO:*)
            {  TAC.stop;
               setverdict.set(pass)
            }
       [] TAC.timeout {
               setverdict(inconc);
            }
       [] L1.receive {
               setverdict(inconc)
            }
    }
   }
    [] L1.receive {
    setverdict(inconc)
       }
  }
   }
[] TAC.timeout {
    setverdict(inconc)
   }
[] L1.receive
    setverdict (inconc)}
   :
```

20.1.1 Выполнение альтернативного поведения

При входе в команду **alt** производится мгновенный снимок. Считается, что мгновенный снимок является частным состоянием тестового компонента, который включает всю информацию, необходимую для оценки булевского состояния, которое блокирует альтернативные ветви, все фактические остановленные тестовые компоненты, все актуальные события тайм-аута и верхние сообщения, запросы, ответы и исключения в актуальных входящих очередях порта. Любой тестовый компонент, таймер и порт, на который ссылаются по крайней мере в одной альтернативе выражения **alt**, или в верхней альтернативе **altstep**, которая вызывается как альтернатива в выражении **alt** или активированы по умолчанию, рассматриваются как важные. Детальное описание семантики мгновенного снимка дается в операционной семантике TTCN-3 (ITU-T Rec. Z.143 [3]).

ПРИМЕЧАНИЕ 1. – Мгновенные снимки являются только концептуальными средствами для описания поведения выражения **alt**. Конкретные алгоритмы для обработки мгновенного снимка могут быть найдены в Рек. МСЭ-Т Z.143 [3].

ПРИМЕЧАНИЕ 2. – TTCN-3 семантика предполагает, что фиксация мгновенного снимка является мгновенной, то есть, не имеет никакой продолжительности. В реальности фиксация мгновенного снимка может происходить некоторое время и могут возникнуть конфликты типа состязания. Обработка таких конфликтов состязаний выходит за пределы этой Рекомендации.

Альтернативные ветви в **alt** утверждении и высших альтернативах для вызванных активизированных по умолчанию altstep и altstep обрабатываются в порядке их появления. Если активны несколько выражений по умолчанию, обратный порядок их активации определяет порядок оценки верхних альтернативных вариантов в выражениях по умолчания. Альтернативные ветви в активных выражениях по умолчанию достигаются стандартным механизмом, описанным в разделе 21.

Индивидуальные альтернативные ветви - либо ветви, которые могут быть блокированы булевым выражением - либо ветви, то есть, альтернативные ветви, начинающиеся с **[else]**.

Когда Else-ветви были достигнуты, они всегда будут выбраны и выполнены (см. 20.1.3).

Ветви, которые могут блокироваться булевыми выражениями, либо вызваны altstep (altstep-ветвь), либо начинают с операции **done** (done -ветвь), операции **killed** (killed - ветвь), операции **timeout** (timeout - ветвь) или операции receiving (receiving - ветвь), то есть операций, **receive**, **trigger**, **getcall**, **getrepy**, **catch** или **check**. Оценка булевой блокировки должна основываться на мгновенном снимке. Считается, что Булева блокировка исполнена, если либо Булева защита не определена или при ее оценке мы получаем **true**. Ветви обрабатываются и выполняются следующим образом.

Altstep-ветвь будет выбрана тогда, когда Булева блокировка исполнена. Выбор altstep-ветви приводит к вызову altstep, на который указывает ссылка, то есть altstep вызывается и оценка мгновенного снимка продолжается в пределах altstep.

Done-ветвь будет выбрана тогда, когда Булева блокировка исполнена и если указанный тестовый компонент находится в списке остановленных компонентов мгновенного снимка. Выбор вызывает выполнение блока команд, который стоит после операции **done**. Команда **done** далее не действует.

Killed-ветвь будет выбрана, если Булева блокировка исполнена и если указанный тестовый компонент находится в списке killed-компонентов мгновенного снимка. Выбор вызывает выполнение блока команд после операции **killed**. Сама операция **killed** далее не действует.

Ветвь timeout будет выбрана, если Булева блокировка исполнена и если указанный тестовый компонент находится в списке timeout-компонентов снимка. Выбор вызывает исполнение указанной операции **timeout**, то есть, удаление события timeout из timeout-списка, и выполнение блока выражений, следующего после операции **timeout**.

Ветвь receiving будет выбрана, если Булева блокировка исполнена и если соответствующие критерии операции receiving выполнены одним из сообщений, запросов, ответов или исключений в мгновенном снимке. Выбор вызывает исполнение операции receiving, то есть удаление совпадающего сообщения, вызов, ответ или исключение из очереди порта, возможно назначение полученной информации переменной и выполнение блока команд, который следует за операцией receiving. В случае операции **trigger**, верхнее сообщение очереди также удаляется, если Булева блокировка исполнена, но соответствующие критерии - нет. В этом случае блок выражений данной альтернативы не выполняется.

ПРИМЕЧАНИЕ 3. – Семантика TTCN-3 описывает оценку мгновенного снимка как ряд неделимых действий испытательного компонента. Семантика не предполагает, что оценка мгновенного снимка не имеет никакой продолжительности. В течение оценки мгновенного снимка, тестовые компоненты могут останавливаться, таймеры могут переполниться и новые сообщения, запросы, ответы или исключения могут появиться в очереди порта компонента. Однако эти события не изменяют фактический мгновенный снимок и, таким образом, не рассматриваются для оценки мгновенного снимка.

Если не может быть выбрана и выполнена ни одна из альтернативных ветвей в выражении **alt** и верхних альтернативах в вызванном altstep и активных умолчаниях, выражение **alt** должно быть выполнено снова; то есть, выполняется новый мгновенный снимок и оценка повторяется для альтернативных ветвей с новым мгновенным снимком. Эта повторная процедура должна продолжаться до того момента, как будет выбрана и выполнена альтернативная ветвь, или же тестовый пример будет остановлен другим компонентом, тестовой системой (например, по причине остановки МТС) или динамической ошибкой.

Если тестовый компонент полностью заблокирован, то тестовый пример должен остановиться и показать динамическую ошибку. Это означает, что ни одна альтернатив может быть выбрана, никакой соответствующий тестовый компонент не исполняется, никакой соответствующий таймер не работает, и все соответствующие порты содержат по крайней мере одно сообщение, вызов, ответ или исключение, которые не согласуются.

ПРИМЕЧАНИЕ 4. – повторная процедура снятия полного мгновенного снимка и повторная оценка всех альтернатив является только концептуальными средствами для описания семантики выражения **alt**. Конкретный алгоритм, который осуществляет эту семантику, не рассматривается в этой Рекомендации.

20.1.2 Выбор/отмена выбора альтернативы

Если необходимо, можно разрешать/запрещать какую-либо альтернативу с помощью Булева выражения, размещаемого между квадратными скобками «[ ]» в этой альтернативе.

Оценка Булева выражения, которое блокирует альтернативу, может иметь побочные эффекты. Для того чтобы избежать побочных эффектов, ввызываемых несогласованностью между фактическим мгновенным снимком и состоянием компонента, должны применяться те же самые ограничения, что и ограничения на инициализацию локальных определений в пределах altstep, (см. 16.2.2.1).

Открывающая и закрывающая квадратные скобки '[' ']' должны располагаться в начале каждой альтернативы, даже если они пустые. Это не только повышает читаемость, но также необходимо для того, чтобы синтаксически различать альтернативы.

ПРИМЕР:

```
// Использование альтернативы с Булевым выражением (или защитой)
 :
alt {
   [x>1] L2.receive {          // Булева блокировка/выражение
            setverdict(pass);
      }
   [x<=1] L2.receive {             // Булева блокировка /выражение
             setverdict(inconc);
      }
}
 :
```

20.1.3 Ветвь Else в альтернативах

Любая ветвь в alt утверждении может быть определена как else ветвь с помощью включения ключевого слова else между открывающими и заключительными скобками в начале альтернативы. Else ветвь не должна содержать никакое из действий, разрешенных в ветвях, защищенных Булевым выражением (то есть, запросом **altstep** или **done**, **killed**, **timeout** или операцией receiving). Блок команд else ветви выполняется всегда, если не выполнена никакая другая альтернатива, предшествующая в тексте ветви else.

ПРИМЕР:

```
// Использование альтернативы с Булевым выражением (или блокировкой)и else ветвью

alt {
 [x>1] L2.receive {
   setverdict(pass);
     }
 [x<=1] L2.receive {
   setverdict(inconc);
         }
 [else] {            // else ветвь
   MyErrorHandling();
setverdict(fail);
   stop;
     }
}
:
```

Следует отметить, что механизм по умолчанию (см. раздел 21) всегда вызывается в конце всех альтернатив. Если ветвь **else** определена, то механизм по умолчанию никогда не будет вызван, то есть не будет произведен вход в активированное **default**.

ПРИМЕЧАНИЕ 1. – Также возможно использовать **else** в **altstep**.

ПРИМЕЧАНИЕ 2. – Допускается использование инструкции **repeat** в **else** ветви

ПРИМЕЧАНИЕ 3. – Допускается определение более одной **else** ветви в выражении **alt** или в **altstep**, тем не менее будет выполнена только первая **else** ветвь.

20.1.4 Утратило значение

20.1.5 Повторная оценка выражений alt

Повторная оценка выражений **alt** может быть установлена при помощи инструкции **repeat (см. 20.2)**

ПРИМЕР:

```
alt {
   [] PCO3.receive {
     count := count + 1;
     repeat                  // использование repeat
      }
```

```
  [] T1.timeout { }
  [] any port.receive {
    setverdict(fail);
    stop;
     }
}
```

20.1.6 Вызов altstep как альтернативы

TTCN-3 позволяет запускать `altstep` как альтернативы в выражениях **`alt`** (см. 16.2.3).

ПРИМЕР:

```
 :
alt {
  [] PCO3.receive { }
  [] AnotherAltStep();   // явный вызов altstep AnotherAltStep как альтернативы
                     // alt инструкции
  [] MyTimer.timeout { }
}
 :
```

20.2 Инструкция repeat

Когда инструкция **`repeat`** используется в блоке выражений и объявлений для альтернатив для команд **`alt`**, то это приводит к повторной оценке команды **`alt`**, то есть, выполняется новый мгновенный снимок и альтернативы команды **`alt`** оцениваются в порядке их спецификации. При использовании в блоках команд и объявлений ответа, и частей обработки исключений вызовов блокирующей процедуры повторная команда приводит к повторной оценке ответа и части обработки исключений вызова (см. 23.3.1.5).

ПРИМЕР 1:

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

Если команда **`repeat`** используется в верхней альтернативе в определении altstep , это вызывает новый мгновенный снимок и повторную оценку выражения **`alt`**, из которого был вызван altstep. Вызов altstep может или быть сделан неявно с помощью механизма по умолчанию (см. статью 21) или явно в выражении **`alt`** (см. 20.1.6).

ПРИМЕР 2:

```
// Использование repeat в altstep
altstep AnotherAltStep() runs on MyComponentType {

  [] PCO1.

  setverdict(inconc);
  repeat            // Использование  repeat
   }
[] PCO2.receive {}
}
```

20.3 Перемежающееся поведение

Инструкция **interleave** позволяет определять перемежающееся возникновение и обработку утверждений **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **catch** и **check**

В командах **interleave** не должны использоваться команды передачи управления **for, do-while, goto, activate, deactivate, stop, repeat, return,** прямые вызовы **altstep** в виде альтернатив и вызовы (прямые и непрямые) определяемых пользователем функций, которые включают операции связи. Кроме того, не разрешаются защитные ветви в команде **interleave** с булевыми выражениями (то есть скобки '[ ]' всегда должны быть пустыми). Не разрешается определять ветви **else** в перемежающемся поведении.

Перемежающееся поведение может быть всегда заменено эквивалентным набором вложенных альтернатив. Процедуры такой замены и операционная семантика перемежения описываются в Рекомендации МСЭ-Т Z.143 [3].

Правила определения перемежающейся команды таковы:

a) сразу после выполнения какой-либо команды приема последовательно выполняются команды не имеющие отношения к приему, пока не поступит следующая команда приема или не закончится перемежающаяся последовательность;

ПРИМЕЧАНИЕ. – Командами приема являются командами TTCN-3, которые могут появляться в наборах альтернатив, то есть команды **receive, check, trigger, getcall, getreply, catch**, **done, killed** и **timeout**. Командами, не имеющими отношения к приему, обозначены все другие не контролирующие передачу команды, которые могут использоваться в команде **interleave**.

b) затем определение продолжается с помощью следующего мгновенного кадра.

Операционная семантика перемежения полностью определена в Рекомендации МСЭ-Т Z.143 [3].

ПРИМЕР:

```
// Следующий кодовый фрагмент TTCN-3
 :
interleave {
[]  PCO1.receive(MySig1)
    {    PCO1.send(MySig2);
     PCO1.receive(MySig3);
    }
[]  PCO2.receive(MySig4)
    {    PCO2.send(MySig5);
     PCO2.send(MySig6);
     PCO2.receive(MySig7);
    }
}
:

// краткая запись для
:
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 Команда Return

Команда **return** заканчивает выполнение функции или alstep и возвращает управление в точку, из которой эта функция или alstep были вызваны. Если она используется в функции, команда **return** факультативно может быть связана с некоторым возвращаемым значением.

ПРИМЕЧАНИЕ. – Когда команда return используется в altstep, то она имеет тот же самый эффект, как будто был достигнут конец блока выражений и деклараций выбранной альтернативы; например, когда altstep вызывается из выражения alt, выполнение продолжается с первого выражения, следующего после **alt** команды.

ПРИМЕР:

```
  function MyFunction() return boolean {
       :
      if (date == "1.1.2005") {
       return false; // Выполнение останавливается на 1.1.2005 и возвращает
                     // булево значение false
   }
       :
  return true; // возвращается истина
 }

 function MyBehaviour() return verdicttype {
   :
  if (MyFunction()) {
     setverdict(pass); // использование MyFunction в выражении if
  }
  else {
     setverdict(inconc);
  }

   :
  return getverdict; // явно возвращает вердикт

  }
:
```

21 Обработка по умолчанию

21.0 Общие положения

TTCN-3 разрешает активацию altstep (см. 16.2) по умолчанию. Каждый тестовый компонент по умолчанию, то есть активные altstep, хранятся в упорядоченном списке. Они перечисляются в порядке, обратном порядку их активации; то есть последний активированный по умолчанию является первым элементом в списке. Операции TTCN-3 **activate** (см. 21.3) и **deactivate** (см. 21.4) работают со списком значений по умолчанию. **Activate** помещает новое умолчание первым элементом списка, а **deactivate** убирает умолчание из списка. Значение по умолчанию в списке значений по умолчанию может идентифицироваться с помощью ссылки по умолчанию, которая создается в результате ссответствующей операции **activate** .

Таблица 14/Z.140 – Обзор команд TTCN-3 обработки поумолчанию

Команды обработки поумолчанию	
Команды	Соответствующее ключевое слово или символ
Активировать по умолчанию	**Activate**
Деактивировать по умолчанию	**deactivate**

21.1 Механизм по умолчанию

Механизм по умолчанию вызывается в конце каждой **alt** инструкции, если по причине фактического мгновенного снимка ни одна из указанных альтернатив не может быть выполнена. Вызванный механизм по умолчанию вызывает первый altstep в списке значений по умолчанию, то есть последнее активизированное значение по умолчанию, и ждет результата его завершения. Завершение может быть успешным или нет. Неудачное завершение означает, что ни одна из верхних альтернатив для **altstep** (см. 16.2), определяющая поведение по умолчанию, не могла быть выбрана, успешное завершение означает, что одна из верхних альтернатив по умолчанию была выбрана и выполнена.

В случае неудачного завершения механизм по умолчанию вызывает следующее значение по умолчанию в списке. Если последнее значение по умолчанию в списке закончилось неудачно, механизм по умолчанию возвратится в ту точку в выражении **alt**, откуда он был вызван, то есть, в конец выражения **alt,** и укажет на неудачное выполнение по умолчанию. Неудачное выполнение по умолчанию будет также обозначено в том случае, если список значений по умолчанию будет пуст.

Неудачное выполнение по умолчанию может вызвать новый мгновенный кадр или динамическую ошибку в том случае, если тестовый компонент заблокирован (см. 20.1).

В случае успешного завершения значение по умолчанию может либо остановить тестовый компонент посредством **stop** команды, либо главный поток управления тестового компонента немедленно продолжится после **alt** команды, из которой механизм по умолчанию вызвали, или тестовый компонент выполнит новый мгновенный снимок и повторно оценит выражение **alt**. Последнее должено быть определено посредством команды **repeat** (см. 20.2). Если выбранная верхняя альтернатива по умолчанию заканчивается без **repeat** команды, то поток управления тестового компонента продолжится непосредственно после выражения **alt**.

> ПРИМЕЧАНИЕ. – TTCN-3 никак не ограничивает реализацию механизма по умолчанию. Это может, например, быть осуществлено в форме процесса, который неявно вызывают в конце каждой команды **alt** или в форме отдельного потока, который ответствен только за обработку значения по умолчанию. Единственное требование заключается в том, что значения по умолчанию при выходе из механизма по умолчанию вызываются в порядке, обратном порядку их активации.

21.2 Ссылки по умолчанию

Ссылки по умолчанию - это уникальные ссылки на активированные значения по умолчанию. Такая уникальная ссылка по умолчанию создается тестовым компонентом, когда altstep активизирован по умолчанию; то есть, ссылка по умолчанию является результатом операции **activate** (см. 21.3).

У ссылок по умолчанию есть специальный и предопределенный тип **default**. Переменные типа **default** могут использоваться для того, чтобы обращаться с активизированными значениями по умолчанию в тестовых компонентах. Имеется специальное значение **null**, которое указывает на неопределенную ссылку по умолчанию, например, для инициализации переменных, которые должны обрабатывать ссылки по умолчанию.

Ссылки по умолчанию используются в операции **deactivate** (см. 21.4) с целью идентифицировать значение по умолчанию, которое должно быть дезактивировано.

Фактическое представление данных типа **default** должно быть решено тестовой системой внешним образом. Это позволяет абстрактным тестовым вариантам быть определенными независимо от реальной окружающей среды во время выполнения TTCN-3; другими словами TTCN-3 не ограничивает выполнение тестовой системы относительно обработки и идентификации значений по умолчанию.

ПРИМЕР:

```
// Объявление переменной для обработки по умолчанию
// и инициализация значением null
var default MyDefaultVar := null;
 :
// Использование MyDefaultVar для хранения активированного значения по
// умолчанию
MyDefaultVar := activate(MyDefAltStep()); // MyDefAltStep активирован по умолчанию
 :
// Использование MyDefaultVar для деактивирования по умолчанию
// MyDefAltStep
deactivate(MyDefaultVar);
 :
```

21.3 Операция Activate

21.3.0 Общие положения

Операция **activate** используется для того, чтобы активизировать altstep по умолчанию. Операция **activate** поместит altstep, на который указывает ссылка, первым элементом в список значений по умолчанию и возвратит ссылку по умолчанию. Ссылка по умолчанию является уникальным идентификатором для значения по умолчанию и может использоваться в операции **deactivate** для дезактивации значения по умолчанию.

Действие операция **activate** является локальным по отношению к тестовому компоненту, в котором его вызывают. Это означает, что тестовый компонент не может активизировать значения по умолчанию в другом тестовом компоненте.

ПРИМЕР 1:

```
 :
// Объявление переменной для обработки по умолчанию

var default MyDefaultVar := null;
 :
// Объявление переменной ссылки по умолчанию и активация altstep по умолчанию
var default MyDefVarTwo := activate(MySecondAltStep());
 :
// Активация altstep MyAltStep по умолчанию
MyDefaultVar := activate(MyAltStep()); // MyAltStep активировано по умолчанию
 :
```

Операция **activate** может быть вызвана без сохранения возвращаемой ссылки по умолчанию. Эта форма полезна в тех тестовых примерах, которые не требуется явной деактивации активного значения по умолчанию; то есть, деактивация значения по умолчанию сделана неявно по завершении МТС.

ПРИМЕР 2:

```
 :
// Активация altstep по умолчанию, без назначения ссылки по умолчанию
```

```
activate(MyCommonDefault());
```

21.3.1 Активация параметризованных altstep

Фактические параметры параметризованных altstep (см. 16.2.1), которые должны быть активированы по умолчанию, должны быть предоставлены в соответствующей команде **activate**. Это означает, что фактические параметры связаны со значением по умолчанию во время его активации (а не, например, во время ее запроса механизмом по умолчанию). Все экземпляры таймеров, которые присутствуют в списке фактических параметров, должны быть объявлены как локальные таймеры компонентного типа (см. 8.5.1).

ПРИМЕР :

```
altstep MyAltStep2 ( integer    par_value1, MyType par_value2,
                     MyPortType par_port,   timer  par_timer )
{
 :
}

function MyFunc () runs on MyCompType
{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);

                // MyAltStep2 активизирован по умолчанию с фактическими параметрами 5 и

                //значением myVar. Изменение myVar перед вызовом MyAltStep2 механизмом по

                 //умолчанию не изменит фактические параметры запроса.
 :
}
```

21.4 Операция deactivate

Операция **deactivate** используется для того, чтобы деактивировать значения по умолчанию, то есть, ранее активизированные altstep. Операция **deactivate** удалит значение по умолчанию, на который указывает ссылка, из списка значений по умолчанию.

Действие операции **deactivate** является локальным по отношению к тестовому компоненту, в котором ее вызывают. Это означает, что тестовый компонент не может деактивировать значения по умолчанию в другом тестовом компоненте.

Операция **deactivate** без параметра деактивирует все значения по умолчанию тестового компонента.

Вызов операции **deactivate** со специальным указателем значения **null** не имеет никакого эффекта. Вызов операции **deactivate** с неопределенной ссылкой по умолчанию, например, старой ссылкой по умолчанию, которая была уже деактивирована или неинициализированной переменной ссылки по умолчанию, приводит к ошибке времени выполнения.

ПРИМЕР:

```
 :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
deactivate(MyDefaultVar); // деактивация MyAltStep
 :
deactivate; // деактивирует все другие значения по умолчанию, то есть, в этом
          //случае MySecondAltStep и MyThirdAltStep
 :
```

22 Операции конфигурации

22.0 Общие положения

Операции конфигурации (см. таблицу 15) используются для установления тестовых компонентов и управления ими. Эти операции используются только в тестовых примерах, функциях TTCN-3 (то есть не используются в управляющей части модуля).

Таблица 15/Z.140 –Обзор операций конфигурации TTCN-3

Операция	Пояснение	Пример синтаксиса
Операции связи		
connect	Соединяет порт одного тестового компонента с портом другого	**connect**(ptc1:p1, ptc2:p2);
disconnect	Рассоединяет два или более соединённых порта	**disconnect**(ptc1:p1, ptc2:p2);
map	Отображает порт тестового компонента на порт тестовой системы	**map**(ptc1:q, **system**:sutPort1);
unmap	Устранить отображение двух или более соединённых портов	**unmap**(ptc1:q, **system**:sutPort1);
Операции тестового компонента		
create	Создание нормального или действующего тестового компонента; различие между нормальными и действующими тестовыми компонентами возникает в процессе создания (MTC, ведет себя как нормальный тестовый компонент)	Не действующие тестовые компоненты: **var** PTCType c := PTCType.**create**; Действующие тестовые компоненты: **var** PTCType c := PTCType.**create alive**;
start	Старт тестового поведения на тестовом компоненте. Старт поведения не затрагивает статус переменных, таймеров или портов компонента	c.**start**(PTCBehaviour());
stop	Остановка тестового поведения на тестовом компоненте	c.**stop**;
kill	Прекращение существования тестового компонента	c.**kill**;
alive	Возвращает true,если тестовый компонент создан и готов к выполнению или уже выполняет поведение, в противном случае возвращает **false**	**if** (c.**alive**) …
running	Возвращает true, пока тестовый компонент исполняет поведение; в противном случае возвращает **false**	if (c.**running**) …
done	Проверяет, закончена ли функция, работающая на тестовом компоненте	c.**done**;
killed	Проверяет закончено ли существования тестового компонента	c.**killed** { … }
Операции ссылок		
mtc	Дает ссылку на MTC	**connect**(**mtc**:p, ptc:p);
system	Дает ссылку на интерфейс тестовой системы	**map**(c:p, **system**:sutPort);
self	Дает ссылку на тестовый компонент , который выполняет эту операцию	**self.stop;**

21.1 Операция Create

MTC является единственным тестовым компонентом, который автоматически создается при запуске тестового примера. Все остальные тестовые компоненты (PTC) должны быть явно созданы во время выполнения теста с помощью операций **create**. Компонент создается с полным набором его портов, у которых входящие очереди пусты и с полным набором его констант, переменных и таймеров. Кроме того, если указанный порт должен иметь тип **in** или **inout** то он в состоянии ожидания должен быть готов к приему трафика по соединению.

Когда компонент создается явным или неявным образом, все переменные и таймеры компонента сбрасываются в их начальные значения, и все константы компонента сбрасываются к их назначенным значениям.

Отличают два типа PTC: PTC, который может выполнить поведенческую функцию только один раз и PTC, который остается действующим после завершения поведенческой функции и по этой причине может быть вновь использоваться для выполнения другой функции. Последний создается, используя дополнительное ключевое слово **alive**. Действующий тип PTC должен быть разрушен явно, используя операцию **kill** (см. 22.9), тогда как недействующий PTC разрушается неявно после того, как его поведенческая функция заканчивается. Завершение тестового примера, то есть завершение MTC, также завершает все PTC, которые все еще существуют.

Так как все тестовые компоненты и порты при окончании каждого тестового примера неявно уничтожаются, при вызове каждого тестового примера должна полностью создаваться его требуемая конфигурация компонентов и соединений.

Операция **create** должна выдавать уникальную ссылку на компонент – на его вновь созданный экземпляр. Уникальная ссылка на компонент обычно будет запоминаться в некоторой переменной (см. п. 8.7) и может использоваться для соединения экземпляров и для целей связи, таких, как передача и прием.

По выбору, название может быть связано с недавно созданным экземпляром компонента. Название должно быть значением **charstring,** и при присваивании оно должно появиться как аргумент функции **create**. Тестовая система должна автоматически привязать названия 'MTC' к MTC и 'SYSTEM' к интерфейсу тестовой системы при создании. Связанные названия компонента не обязаны быть уникальными.

ПРИМЕЧАНИЕ. – Название экземпляра компонента используется только для регистрационных целей (см. 19.3), должно использоваться только для обращения к экземпляру компонента (с этой целью должна использоваться ссылка компонента) и не влияет на соответствие.

ПРИМЕР:

```
     // Этот пример объявляет переменную типа MyComponentType, которая используется
     //для хранения ссылок на на вновь созданные экземпляры компонента
     //  MyComponentType, которые являются результатом операции create.
// Соответствующее имя   зарезервировано за некоторыми созданными экземплярами
// компонента.
      :
     var MyComponentType MyNewComponent;
     var MyComponentType MyNewestComponent;
     var MyComponentType MyAliveComponent;
     var MyComponentType MyAnotherAliveComponent;
      :
     MyNewComponent := MyComponentType.create;
     MyNewestComponent := MyComponentType.create("Newest");
     MyAliveComponent := MyComponentType.create alive;
     MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
      :
```

Компоненты могут создаваться в любой точке определения поведения, что обеспечивает наивысшую степень гибкости по отношению к динамическим конфигурациям (то есть любой компонент может создать любой другой PTC). Видимость ссылок на компоненты подчиняется тем же контекстным правилам, что и видимость ссылок на переменные, а чтобы сослаться на компоненты, находящиеся вне этого контекста создания, ссылка на компонент должна быть передана в виде параметра или в виде поля сообщения.

22.2 Операции Connect и Map

22.2.0 Общие положения

Порты тестового компонента могут быть соединены с другими компонентами или с портами интерфейса тестовой системы. В случае соединений между двумя тестовыми компонентами используется операция **connect**. Когда тестовый компонент соединяется с интерфейсом тестовой системы, используется операция **map**. Операция **connect** прямо соединяет один порт с другим, причем сторона **in** соединяется со стороной **out** и наоборот. С другой стороны, операцию **map** можно рассматривать исключительно как преобразование имен, которое определяет, как следует указывать потоки связи

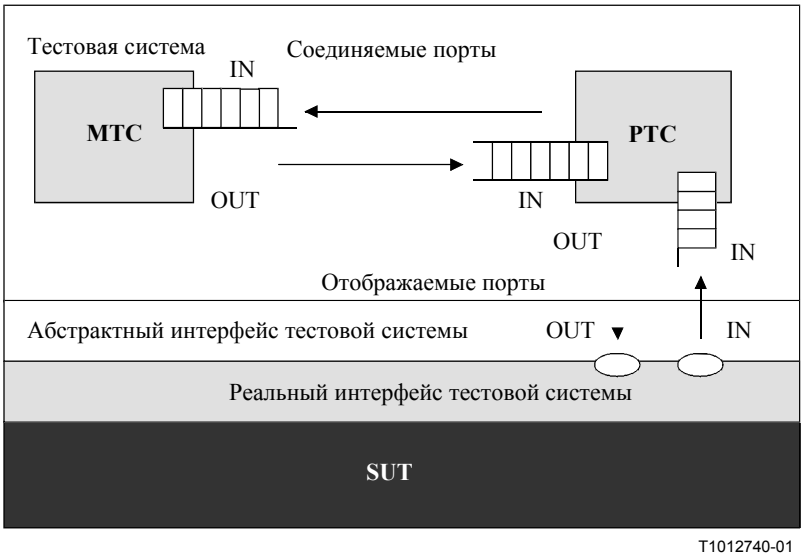


Рисунок 10/Z.140 – Иллюстрация операций connect и map

Как при операции **connect**, так и при операции **map** соединяемые порты определяются ссылками на компоненты, которые должны быть соединены, и именами портов, которые необходимо соединить.

Операция **mtc** идентифицирует MTC, операция **system** идентифицирует тестовый интерфейс системы, и операция **self** идентифицирует тестовый компонент, в котором **self** был вызван (см. 22.4). Все эти операции могут использоваться для идентификации и соединения портов.

Операции **connect** и **map** могут быть вызваны из любого определения поведения , за исключением управляющей части модуля. Однако до вызова любой из этих операций должны быть созданы компоненты, которые необходимо соединить, а их ссылки на компоненты должны быть известны вместе с именами соответствующих портов.

Операции **connect** и **map** позволяют соединять один порт с несколькими другими портами. Не разрешается соединять с отображенным портом или отображать в соединенный порт.

ПРИМЕР:

```
   // Предполагается, что порты Port1, Port2, Port3 и PCO1
   // надлежащим образом определены и объявлены в соответствующих
   // определениях типа порта и типа компонента
   :
   var MyComponentType  MyNewPTC;
   :
   MyNewPTC : = MyComponentType.create;
   :
   :
   connect (MyNewPTC : Port 1, mtc : Port 3);
   map (MyNewPTC : Port2, system : PCO1);
   :
   :
// В этом примере создается новый компонент типа MyComponentType,
// а ссылка на него записывается в переменной MyNewPTC. Потом при
// операции connect Port1 этого нового компонента соединяется с Port3 в MTC.
// С помощью операции map Port2 нового компонента затем соединяется с
// портом PCO1 интерфейса тестовой системы
```

22.2.1 Совместимые соединения и отражения

Для обеих операций **connect** и **map** разрешаются только совместимые соединения.

Предполагается следующее:

a) порты PORT1 и PORT2 – это порты, которые необходимо соединить;

b) список in в PORT1 определяет сообщения или процедуры для направления in этого PORT1;

c) список out в PORT1 определяет сообщения или процедуры для направления out этого PORT1;

d) список in в PORT2 определяет сообщения или процедуры в направлении in этого PORT2; и

e) список out в PORT2 определяет сообщения или процедуры в направлении **out** этого PORT2.

Операция **connect** разрешена тогда и только тогда, когда:

•список out в PORT1 ⊆ список in в PORT2 *и* список out в PORT2 ⊆ список in в PORT1;

Операция **map** (в предположении, что PORT2 является портом интерфейса тестовой системы) разрешена тогда и только тогда, когда:

•список out в PORT1 ⊆ список out в PORT2 *и* список in в PORT2 ⊆ список in в PORT1.

Во всех остальных случаях эти операции не разрешены.

Так как TTCN-3 разрешает динамические конфигурации и адреса, не все из этих проверок совместимости могут быть осуществлены статически во время трансляции. Все проверки, которые не могут быть осуществлены во время трансляции, должны быть проведены во время выполнения и, в случае если они окажутся неуспешными, должны указать на ошибку тестового примера.

22.3 Операции Disconnect и Unmap

Операции **disconnect** и **unmap** противоположны операциям **connect** и **map**. Они осуществляют разъединение (ранее соединенных) портов тестовых компонентов и устранение отображения (ранее отображенных) портов тестовых компонентов и портов в интерфейсе тестовой системы.

Операции **disconnect** и **unmap** могут быть вызваны из любого компонента, если известны соответствующие ссылки на компоненты вместе с именами соответствующих портов. Операция **disconnect** или **unmap** действует только в случае, когда соединение или отображение, которое необходимо отменить, было создано ранее.

ПРИМЕР 1:

```
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
disconnect(MyNewComponent:Port1, mtc:Port3); // Разъединить ранее установленное
                                             // соединение
unmap(MyNewComponent:Port2, system:PCO1);    // Устранить отображение ранее
                                             // осуществленного отображения
```

Для упрощения операций **disconnect** и **unmap,** относящихся ко всем соединениям или отражениям компонента или порта, разрешено использовать операции **disconnect** и **unmap** с одним аргументом . Этот аргумент определяет одну сторону связей, которые должны быть соединены или отражены. Можно использовать ключевое слово **all port** , чтобы обозначить все порты компонента.

ПРИМЕР 2:

```
 :
disconnect(MyNewComponent:Port1);     // отсоединить все соединения Port1,
                                      // принадлежащие MyNewComponent.
unmap(MyNewComponent:all port);       // устранить отображения всех портов
                                      // компонента MyNewComponent
 :
```

Использование операций **disconnect** или **unmap** без параметров есть краткая форма использования операции с параметром **self:all port**. Эта операция разъединяет или устраняет отображения всех портов компонента, вызвавшего операцию .

ПРИМЕР 3:

```
 :
disconnect;                           // краткая форма  …
disconnect(self:all port);            // которая отсоединяет все порты
 :
```

```
unmap;                                  // краткая форма  … ,
 unmap(self:all port);                      // которая устраняет отображения всех
                                        // портов компонента
                                        // вызвавшего операцию
 :
```

Ключевое слово **all component** должно использоваться только в комбинации с ключевым словом **all port,** то есть **all component**:**all port**, и оно должно использоваться только MTC. Кроме того, аргумент **all component:all port** должен использоваться исключительно как аргумент операций **disconnect** или **unmap**, что позволяет произвести отключение всех связей и отображений тестовой конфигурации.

ПРИМЕР 4:

```
 :
 :
disconnect(all component:all port);       // MTC рассоединяет все порты всех
                                          // компонентов тестовой конфигурации
 :
 :
unmap(all component:all port);            //MTC устраняет отображения всех
                                             //портов всех компонентов тестовой
                                          // конфигурации

 :
```

22.4 Операции MTC, System и Self

Компонентные ссылки (см. п. 8.7) поддерживают три операции: **mtc** и **system**, которые выдают ссылки на главный тестовый компонент и на интерфейс тестовой системы соответственно. Может использоваться операция **self**

ПРИМЕР :

```
var MyComponentType MyAddress;
MyAddress := self; // Запомнить текущую компонентную ссылку
```

С компонентными ссылками разрешаются только операции присвоения, равенства и неравенства.

22.5 Операция Start тестового компонента

Как только PTC создан и соединен, к нему должно быть привязано поведение и должно начаться выполнение этого поведения. Это делается с помощью операции **start** (создание PTC не запускает выполнение поведения компонента). Разграничение между **create** и **start** сделано для того, чтобы позволить осуществлять операции соединения до реального выполнения тестового компонента.

Операция **start** привязывает требуемое поведение к тестовому компоненту. Это поведение определяется путем ссылки на уже определенную функцию.

Действующий тип PTC может выполнить несколько поведенческих функций в последовательном порядке. Старт второй поведенческой функции на недействующем PTC или старт функции на PTC, который все еще работает, приводит к ошибке тестового примера. Если функция запущена на действующем типе PTC после завершения предыдущей функции, то она использует переменные значения, таймеры, порты, и локальный вердикт, как если бы они остались после завершения предыдущей функции. В частности если бы таймер был запущен в предыдущей функции, то в последующей функции необходимо обрабатывать возможное событие timeout.

ПРИМЕР :

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }

 :
 var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
 :
MyNewPTC := MyComponentType.create;               // Создание нового недействующего
                                                  // тестового компонента
MyAlivePTC := MyComponentType.create alive;       // Создание нового действующего
                                                  // тестового компонента
 :
MyNewPTC.start(MyFirstBehaviour());              // Запуск недействующего компонента
```

```
MyNewPTC.done;                                  // Ожидание завершения
MyNewPTC.start(MySecondBehaviour());            // Проверка ошибки
 :
MyAlivePTC.start(MyFirstBehaviour());           // Запуск компонента действующего
                                                // типа
MyAlivePTC.done;                                // Ожидание завершения
MyAlivePTC.start(MySecondBehaviour());          // Запуск следующей функции на этом
                                                // же компоненте
  :
```

К функции, вызванной при операции **start** тестового компонента, применяются следующие ограничения:

• Если эта функция имеет параметры, то они могут быть только параметрами **in**, то есть параметрами значения.

• Эта функция должна иметь определение **runs on**, которое ссылается на тип компонента, совместимый с заново созданным компонентом (см. 6.7.3).

• Порты и таймеры не должны передаваться в эту функцию.

ПРИМЕЧАНИЕ. – Когда компонент создан, и порты in и inout ожидают сигнала, в момент их включения уже могут присутствовать сообщения во входящей очереди, которые ожидают обработки

22.6 Операция Stop тестового компонента

Тестовый компонент может остановить выполнение своего собственного тестового поведения или тестового поведения, выполняемого другим тестовым компонентом, при помощи команды **stop.** Если компонент останавливает не свое собственное поведение, а поведение, исполняемое на другом компоненте в тестовой системе, компонент, который должен быть остановлен, должен быть идентифицирован с помощью ссылки на компонент. Компонент может остановить свое собственное поведение при использовании простого выражения исполнения **stop** (см. 19.10), или обращаясь к себе в операции **stop**, например, при использовании операции **self**.

ПРИМЕР 1:

ПРИМЕЧАНИЕ 1. – В то время как операции **create**, **start**, **running, done** и **killed** могут использоваться только для PTC (ов), операция **stop** может также быть применена к MTC.

```
var MyComponentType MyComp := MyComponentType.create;     // Создаётся новый
                                                          //  компонент
MyComp.start(CompBehaviour());                            // Новый компонент запущен
 :
if (date = = "1.1.2005") { MyComp.stop }                  // Компонент «MyComp» остановлен
}
 :
if (a < b ) {
    :
   self.stop;   // Тестовый компонент, исполняемый  в настоящее время , останавливает свое
                // собственное поведение
}
 :
   stop        // Тестовый компонент останавливает свое собственное поведение
```

Остановка испытательного компонента является явной формой завершения выполняемого в настоящее время поведения. Тестовое поведение компонента заканчивается также при достижении конца тестового примера или функции, которая начата на этом компоненте, или при помощи явного выражения **return**. Такое завершение также называют неявной остановкой. Неявная остановка имеет тот же самый эффект, что и явная остановка; то есть, глобальное заключение обновляется при помощи локального заключения остановленного тестового компонента (см. раздел 25).

Если остановленный тестовый компонент является MTC, то ресурсы всех существующих PTC должны быть освобождены, PTC должны быть удалены из тестовой системы, и тестовый пример должен быть завершен (см. 27.2).

Остановка недействующего тестового компонента (неявно или явно) должна разрушить его, а все ресурсы, связанные с ним должны быть разблокированы.

Остановка действующего компонента должна остановить только исполняемое в настоящее время поведение, а компонент продолжает существовать и может выполнить новое поведение (запускаемое на нем с помощью операции **start**). После остановки его поведения компонент должен быть оставлен в согласованном состоянии.

ПРИМЕЧАНИЕ 2. – Например, если поведение компонента действующего типа будет остановлено во время присваивания нового значения к уже связанной переменной, то переменная должна остаться связанной и после того, как

компонент будет остановлен (со старым или новым значением). Аналогично, если компонент будет остановлен во время перезапуска уже работающего таймера, то таймер нужно оставить в запущенном состоянии после завершения поведения.

Правила завершения тестовых случаев и вычисления заключительного тестового вердикта описаны в статье 25. Ключевое слово **all** может использоваться только MTC, чтобы остановить все работающие PTC за исключением самого MTC.

ПРИМЕЧАНИЕ 3. – PTC может остановить выполнение тестовое варианта, остановив MTC.

ПРИМЕР 2:

```
   :
all component.stop      // MTC останавливат все PTC тестового примера, но не
                        // себя
```

ПРИМЕЧАНИЕ 4. – Конкретный механизм остановки всех оставшихся работающих PTC не рассматривается в данной Рекомендации.

22.7 Операция Running

Операция **running** позволяет при выполнении поведения в тестовом компоненте установить, завершено ли выполнение поведения в другом тестовом компоненте. Операция **running** может быть использована только для PTC. Операция **running** возвращает истину для PTCов, запущенных, но которые еще не были завершены или остановлены. Операция **running** считается булевым выражением и поэтому выдает булево значение, чтобы указать, окончен ли указанный тестовый компонент (или все тестовые компоненты). В отличие от операции **done** операция **running** может свободно использоваться в булевых выражениях.
Когда ключевое слово **all** используется с операцией **running** , оно возвратит **true**, если все запущенные и не остановленные явно другим компонентом PTC выполняют свое поведение. Иначе, возвращается **false**.
Когда ключевое слово **any** используется с операцией **running**, оно возвратит **true**, если по крайней мере один PTC выполняет свое поведение. Иначе возвращается **false**.

ПРИМЕР:

```
if (PTC1.running)                          // использование running в команде if
{
    // Делайте что-нибудь!
}


while (all component.running != true) {   // использование running в условии цикла
    MySpecialFunction()
}
```

22.8 Операция Done

Операция **done** позволяет при выполнении поведения в тестовом компоненте установить, завершено ли выполнение поведения в другом тестовом компоненте. Операция **done** может использоваться только для PTC.

Операция **done** используется так же, как операция приема или операция **timeout**. Это значит, что она не должна использоваться в булевых выражениях, но она может использоваться для определения альтернативы в команде **alt** или в качестве автономной команды в описании поведения. В последнем случае операция **done** рассматривается как краткая форма команды **alt** только с одной альтернативой, то есть она имеет блокирующуюся семантику и поэтому обеспечивает способность пассивного ожидания окончания тестового компонента.

Когда операция **done** применяется к PTC, она подходит только в том случае, если поведение PTC было остановлено (неявно или явно) или PTC был уничтожен. В противном случае соответствие является неудачным.
Когда ключевое слово **all** используется с операцией **done**, то оно соответствует, если ни один PTC не выполняет свое поведение. Это также соответствует, если ни один PTC не был создан.
Когда ключевое слово **any** используется с операцией **done**, то оно соответствует в том случае, если по крайней мере поведение одного PTC было остановлено или уничтожено. Иначе, соответствие является неудачным.

ПРИМЕЧАНИЕ. – Остановка поведения недействующего компонента также приводит к удалению такого компонента из тестовой системы, в то время как остановка компонента действующего типа оставляет компонент действующим в тестовой системе. В обоих этих случаях операция **done** является соответствующей.

ПРИМЕР:

```
// Использование done в альтернативах
:
alt {
    []   MyPTC.done {
    setverdict(pass)
   }
    }

    []   any port.receive {
    repeat
    }
}
:

var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
 // сответствует до того момента, как функция MyPTCBehaviour (или
 // функция/altstep которую она вызывает) остановилась
c.done;
 // тоже соответствует завершению MyPTCBehaviour (или функция/altstep, которую
 // она вызывает)
 if(c.running) {c.done}
    // операция done здесь соответствует только завершению следующего поведения


 // следующая done в качестве автономной команды
  :
 all component.done;
  :

 // имеет следующий смысл:
  :
 alt {
    []   all component.done {}
 }
  :

 // и таким образом блокирует выполнение до окончания всех
 // параллельных тестовых компонентов
```

22.9 Операция тестового компонента Kill .

Операция **kill,** примененная к тестовому компоненту, останавливает выполнение текущего поведения - если оно существует – и освобождает все ресурсы, связанные с этим компонентом (включая все связи порта уничтоженного компонента), а также удаляет компонент из тестовой системы. Операция **kill** может быть применена к самому тестовому компоненту с помощью непосредственного выражения **kill,** или же с помощью обращения к операции **self** совместно с операцией уничтожения. Операция **kill** может также быть применена к другому тестовому компоненту. В этом случае к компоненту, который будет уничтожен, нужно обратиться, используя ссылку на него. Если операция **kill** применена к МТС, например, **mtc.kill**, то это заврешает тестовый пример.

ПРИМЕР 1:

```
var PTCType MyAliveComp := PTCType.create alive;  // Создает действующий
                                                  // тестовый компонент
MyAliveComp.start(MyFirstBehavior());        // Новый компонент запущен
MyAliveComp.done;                            // Ожидание завершения
MyAliveComp.start(MySecondBehavior());        // Запуск компонента второй раз
MyAliveComp.done;                            // Ожидание завершения
MyAliveComp.kill;                            // Освобождение его ресурсов
```

Ключевое слово **all** может использоваться МТС только для того, чтобы остановить и уничтожить все исполняемые PTC за исключением самого МТС.

ПРИМЕР 2:

```
all component.kill;  // МТС останавливает все PTC  первого тестового варианта
                     // и освобождает их ресурсы.
```

22.10 Операция Alive

Операция **alive** - логическая операция, которая проверяет, был ли тестовый компонент создан и готов ли выполнить или уже выполняет функцию поведения. Примененная к нормальному тестовому компоненту, операция **alive** возвращает **true**, если компонент не активен или выполняет функцию, и в противном случае возвращает **false**. Примененная к действующему типу тестового компонента, операция возвращает **true**, если компонент не активен, работает или остановлен. Операция возвращает **false**, если компонент был уничтожен. Подобно операции **running**, операция **alive** может использоваться только для PTCs (см. 22.7). В частности, в комбинации с ключевым словом **all** она возвращает **true** в том случае, если все (действующего типа или нормального) PTC являются действующими.

Операция **alive** в комбинации с ключевым словом **any** возвращает **true** в том случае, если по крайней мере один PTC является действующим.

ПРИМЕР:

```
 :
PTC1.done;                          // Ожидает завершения компонента
if (PTC1.alive) {                       // Если компонент все еще действует …
    PTC1.start(AnotherFunction());      // … выполняет для него другую функцию.
}
```

22.11 Операция killed

Операция **killed** позволяет устанавливать, действует ли другой тестовый компонент или был удален из тестовой системы.

Операция **killed** должна использоваться таким же образом как и операция **receiving**. Это означает, что она не должно использоваться в булевых выражениях, но она может использоваться, чтобы определить альтернативу в выражении **alt** или как автономное утверждение в описании поведения. В последнем случае операция **done** рассматривается как краткий вариант выражения **alt** с единственной альтернативой; то есть, она имеет семантику блокирования, и поэтому обеспечивает возможность пассивного ожидания завершения тестовых компонентов.

ПРИМЕЧАНИЕ. – При проверке нормальных тестовых компонентов, операция **killed** соответствует в том случае, если остановлено (явно или неявно) выполнение его поведения или он был **killed** явно, то есть, операция эквивалентна операции **done** (см. 22.8). При проверке действующего типа тестового компонентов, тем не менее, операции **killed** подходит только в том случае, если компонент был уничтожен, используя операцию **kill**. Иначе операция **killed** является неудачной.

Операция **killed** может использоваться только для PTCs .

Когда ключевое слово **all** используется с операцией **killed** , оно подходит, если все PTCs тестового примера прекратили свое существование. Также оно подходит, если не было создано никакого PTC.

Когда ключевое слово **all** используется с операцией **killed**, оно подходит, если по крайней мере один PTC прекратил существовать. Иначе, соответствие является неудачным.

ПРИМЕР:

```
var MyPTCType ptc := MyPTCType.create alive; // создает тестовый компонент
                                    // действующего типа
timer T(10.0);                          // создает таймер
T.start;                            // запуск таймера
ptc.start(MyTestBehavior());                // запуск выполнения функции PTC
alt {
[] ptc.killed {                         // если во время выполнения PTC был уничтожен …
    T.stop;                             // … остановка таймера и …
    setverdict(inconc);                     // … ставится заключение
                                    // «незавершено»
   }
[] ptc.done {                           // если PTC заканчивается
                                    // регулярно …
    T.stop;                             // … остановка таймера и …
    ptc.start(AnotherFunction());           // … запуск другой функции на PTC
   }
[] T.timeout {                          // если произошел таймаут до остановки PTC
    ptc.kill;                           // … уничтожение PTC и …
    setverdict(fail);                   // … ставится заключение «неудача»
   }
}
```

22.12 Использование массивов компонентов

Операции **create, connect, start** , **stop** и **kill** не работают прямо в массивах компонентов. Вместо этого определенный элемент массива должен быть предоставлен как параметр этих операций. Для компонентов для создания массива необходимо использовать ссылку на массив компонентов и присвоить соответствующий элемент массива результату операции **create**.

ПРИМЕР:

```
//    Этот    пример    показывает,    как    моделировать    эффект    создания,    соединения
  //    и    исполнения    массивов    компонентов,    используя    цикл    и    запоминание    ссылки
  // на создаваемый компонент в массиве ссылок на компоненты.
    testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
      {
       :
      var integer i;
      var MyPTCType1 MyPtc [11];
       :
      for (i:= 0; i<=10; i:=i+1)
      {
          MyPtc [i] := MyPtcType1.create;
          connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCordination);
          MyPtc [i].start(MyPtcBehaviour());
          }
          :
      }
```

22.13 Резюме использования any и all с компонентами

Ключевые слова **any** и **all** могут использоваться с операциями конфигурации, как показано в таблице 16.

Таблица 16/Z.140 – Any и All с компонентами

Операция	Разрешено		Пример	Комментарий
	any (см. примеч)	**all** (см. примеч)		
create				
start				
running	Да, но только из MTC	Да, но только из MTC	**any component.running;** **all component.running;**	Выполняет ли какой-нибудь PTC тестовое поведение? Выполняют ли все PTC тестовое поведение?
alive	Да, но только из MTC	Да, но только из MTC	**any component.alive;** **all component.alive;**	Есть ли какой-нибудь действующий PTC? Все ли PTC действующие?
done	Да, но только из MTC	Да, но только из MTC	**any component.done;** **all component.done;**	Есть ли какой-нибудь завершивший выполнение PTC? Завершили ли работу все PTC?
killed	Да, но только из MTC	Да, но только из MTC	**any component.killed;** **all component.killed;**	Есть ли какой-нибудь прекративший существование PTC? Завершили ли существование все PTC?
stop		Да, но только из MTC	**all component.stop;**	Остановить поведение всех PTC.
kill		Да, но только из MTC	**all component.kill;**	Уничтожить все PTC, то есть прекратить их существование.
ПРИМЕЧАНИЕ. – **any** и **all** относятся только к PTC; то есть MTC не рассматривается.				

23 Операции связи

23.0 Общие положения

TTCN-3 поддерживает связь на базе сообщений и на базе процедур Кроме того, TTCN-3 позволяет исследовать главный элемент поступающих очередей порта и управлять доступом к портам посредством *операций управления.*

Таблица 17/Z.140 – Обзор коммуникационных операций TTCN-3

Коммуникационные операции			
Коммуникационные операции	**Ключевое слово**	**Может быть использовано в портах на базе сообщений**	**Может быть использовано в портах на базе процедур**
Коммуникации на базе сообщений			
Послать сообщение	**send**	Да	
Принять сообщение	**receive**	Да	
Ожидание сообщения	**trigger**	Да	
Коммуникации на базе процедур			
Вызвать процедуру call	**call**		Да
Принять процедуру call от внешнего объекта	**getcall**		Да
Ответить на процедуру call от внешнего объекта	**reply**		Да
Порождает особое состояние (по принятому вызову)	**raise**		Да
Обработать ответ на предыдущий вызов	**getreply**		Да
Захватить особое состояние (от вызываемого объекта)	**catch**		Да
Проанализировать верхний элемент входящей очереди порта			
Проверить принятые сообщение/вызов/исключение/ответ	**check**	Да	Да
Операции управления			
Очистить очередь команд	**clear**	Да	Да
Очистить очередь и разрешить прием и посылки по порту	**start**	Да	Да
Блокировать передачу и запретить операции приема для соответствия в порту	**stop**	Да	Да
Блокировать передачу и запретить операции приема для соответствия новых сообщений/вызовов	**halt**	Да	Да

23.1 Общий формат операций связи

23.1.0 Общие положения

Операции **send** и **call** используются для обмена информацией между тестовыми компонентами и между SUT и тестовыми компонентами. Для того, чтобы объяснить в общем виде формат данных операций, они могут быть разделены на две группы:

a) тестовый компонент передает сообщение (операция **send**), вызывает какую-либо процедуру (операция **call**), или отвечает на принятый вызов (операция **reply**), или порождает исключение (операция **raise**). Эти действия вместе называются *операциями передачи*;

b) компонент принимает сообщение (операция **receive**), ожидает сообщения (операция **trigger**), принимает вызов процедуры (операция **getcall**), получает ответ на ранее вызванную процедуру (операция **getreply**), или обрабатывает исключение (операция **catch**),. Эти действия вместе называются *операциями приема*.

23.1.1 Общий формат операций передачи

Операции передачи состоят из части *передачи* и, в случае блокирующей операции **call** на базе процедур, части *ответ и обработка исключения*.

Часть "передача":

- описывает порт, в котором должна выполняться определяемая операция;
- определяет сообщение или процедуру, предназначенные для передачи;
- дает (дополнительную) часть адреса, которая однозначно идентифицирует одного или более участников коммуникаций, которым необходимо послать сообщение, вызов, ответ или исключение.

Имя порта, имя операции и значение должны присутствовать во всех операциях передачи. Указание части адреса (обозначаемое ключевым словом **to**) факультативно и требуется только в случаях связей "один ко многим", в которых:

- используется связь с одним субъектом, и один субъект получения должен быть явно идентифицирован.
- используется связь со многими субъектами, и ряд субъектов получения должен быть явно идентифицирован;
- используется широковещательная связь, и необходимо обратиться ко всем объектам, которые связаны с указанным портом.

ПРИМЕЧАНИЕ. – Коммуникационные термины "unicast", "multicast" и "broadcast", используются применительно к коммуникации портов. Это означает, что возможно обратиться только к одному, нескольким или всем тестовым компонентам, которые связаны с указанным портом. Unicast, multicast и broadcast могут также использоваться для портов, для которых задано отображение. В этом случае, один, несколько или все объекты в пределах SUT могут быть доступны через указанный отображенный порт.

EXAMPLE 1:

Часть послания			**(Дополнительно) ответ и исключение**
Порт и операция	**Часть значения**	**(Дополнительно) часть адреса**	**Часть обработки**
`MyP1.send`	`(MyVariable + YourVariable - 2)`	`to MyPartner;`	

Ответ и обработка особого состояния нужны только в случаях связи на основании процедур. Часть "ответ и обработка особого состояния" в операции **call** факультативна и требуется в случаях, когда вызванная процедура выдает какое-либо значение либо имеет параметр **out** или **inout**, значение которого необходимо для вызывающего компонента, а также в случаях, когда вызванная процедура может породить особые состояния, которые должны быть обработаны вызывающим компонентом.

Часть "ответ и обработка особого состояния" операции вызова использует операции **getreply** и **catch** для обеспечения требуемых функциональных возможностей.

ПРИМЕР 2:

Часть послания			**(Факультативно) ответ и обработка исключения**
Порт и операция	**Часть значения**	**(Факультативно) часть адреса**	

MyP1.**call**	(MyProc:{MyVar 1})		{
			[]
			MyP1.**getreply**(MyProc:{MyVar2}) {}
			[] MyP1.**catch**(MyProc, ExceptionOne) {}
			}

23.1.2 Общий формат операций приема

Операция приема состоит из части *прием* и (факультативно) части *присвоение*.

Часть "прием":

a) описывает порт, в котором должна выполняться эта операция;

b) определяет порт сопоставления, который указывает подходящий вход, который будет сопоставлять команду;

c) выдает (возможно) адресное выражение, которое однозначно определяет партнера по связи (в случае соединений типа "один ко многим").

Должны присутствовать имя порта, имя операции и часть "значение" для всех операций приема. Указание партнера по связи (отмечаемое ключевым словом **from**) факультативно, его требуется указывать только в случаях соединений типа "один ко многим", когда требуется явно указать передающий объект.

Часть "присвоение" в операции приема является факультативной. Для портов на базе сообщений она используется в случаях, когда требуется запоминать принятые сообщения. В случае портов на базе процедур она используется для запоминания параметров **in** и **inout** полученного вызова, или запоминания возвращенных значений или для хранения особых состояний. Для части назначения требуется точное выражение; например, у переменной, используемой для того, чтобы хранить сообщение, должен быть тот же тип, как у поступающего сообщения. Кроме того, часть присваивания может также использоваться, чтобы назначить адрес **sender** сообщения, исключения, **reply** или **call** переменной. Это используемо для подключений "один ко многим", где, например, то же самое сообщение или запрос могут быть получены от различных компонентов, но сообщение, ответ или исключение нужно отослать назад к исходному компоненту отправки.

ПРИМЕР:

Часть приема				**(Факультативно) часть назначения**		
Порт и операция	**Часть соответствия**	**(Факультативно) адресное выражение**		**(Факультативно) назначение значения**	**(Факультативно) назначение значения параметра**	**(Факультативно) назначение значения отправителя**
MyP1.getreply	(AProc:{?} value 5)		->		param (V1)	sender APeer

Часть приема				**(Факультативно) часть назначения**		
Порт и операция	**Часть соответствия**	**(Факультативно) адресное выражение**		**(Факультативно) назначение значения**	**(Факультативно) назначение значения параметра**	**(Факультативно) sender назначение значения отправител**

						я
MyP2.receive	(MyTemplate(5,7))	from APeer	->	value MyVar		

23.2 Связь, основанная на сообщениях

23.2.0 Общие положения

Связь, основанная на сообщениях, представляет собой связь, основанную на асинхронном обмене сообщениями. Связь, основанная на сообщениях, не блокируется на операции **send**, как показано на Рисунке 11, где обработка в ОТПРАВИТЕЛЕ продолжается немедленно после того, как происходит операция **send**. ПРИЕМНИК блокируется на операции **receive**, пока он не обработает полученное сообщение.

В дополнение к операции **receive** TTCN-3 обеспечивает операцию **trigger** , которая фильтрует сообщения с определенными критериями соответствия из потока полученных сообщений на данном входном порту. Верхние сообщения очереди, которые не соответствуют соответствующим критериям, удаляются из порта без каких-либо дальнейших действий.

Рисунок 11/Z.140 – Иллюстрация асинхронных операций передачи и приема

23.2.1 Операция передачи (Send)

23.2.1.0 Общие положения

Операция **send** используется, чтобы поместить сообщение в исходящий порт сообщения. Сообщение может быть определено, ссылаясь на определенный шаблон или может быть определено как in-line шаблон. Определяя сообщение in-line, необходимо использовать факультативную часть типа в том случае, если существует неопределенность типа для отправляемого сообщения.

Операция **send** должна использоваться на основанных на сообщении (или смешанных) портах, и тип шаблона, который должен быть послан, должны находиться в списке исходящих типов определения типа порта.

ПРИМЕР:

```
MyPort.send(MyTemplate(5,MyVar)); // Посылает шаблон MyTemplate                                    с
                                  // фактическим параметром 5 и MyVar через MyPort.

     MyPort.send(5);              // Посылает целое число 5 (как
                                  // действующий шаблон)
```

23.2.1.1 Посылка для одного участника (unicast), для многих участников (multicast) или широковещательная (broadcast)

TTCN-3 поддерживает посылку, предназначенную для одного участника (однонаправленную, unicast), для многих участников (многонаправленную, multicast), и широковещательную (broadcast) связь. Используемый механизм коммуникации может быть определен дополнительным пунктом **to** в операции **send**. Пункт **to** может быть опущен в случае непосредственной связи, где используется однонаправленная коммуникация, и приемник сообщения уникально определен тестовой структурой системы. Пункт **to** будет присутствовать в случае связей "один ко многим".

Однонаправленная (unicast) связь определена в том случае, если пункт **to** обращается только к одному участнику коммуникации. Многонапрвленная (multicast) связь используется в том случае, если пункт **to**

включает список участников коммуникации. Широковещательная (broadcast) связь определена в том случае, когда выражение **to** используется с ключевым словом **all component**.

ПРИМЕР:

```
MyPort.send(charstring:"My string") to MyPartner;
                    // Посылает компоненту строку  "My string" с

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
                    // Посылает результат арифметического выражения в MyPartner.

MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                    // Определяет многонаправленную связь,где значение
                    // MyTemplate посылается по двум ссылкам на компоненты заключенным
                    // в переменных MyPeerOne и MyPeerTwo.

MyPCO3.send(MyTemplate) to all component;
                    // Широковещательная связь: значение  Mytemplate посылается
                    // всем компонентам,которые могут адресоваться с этого порта. Если
                    // MyPCO3 является открытым портом, компоненты могут находиться внутри
                    // SUT.
```

23.2.2 Операция Receive

23.2.2.0. Общие положения

Операция receive применяется для того, чтобы принять сообщение из входящей очереди сообщений порта. Сообщение может быть определено с помощью ссылки на определенный шаблон или может быть определено как шаблон in-line. Когда определяется действующее сообщение, должна присутствовать дополнительная часть типа всякий раз, когда тип получаемого сообщения неоднозначен. Операция **receive** используется только в портах на базе сообщений (или смешанных), причем тип значения, которое будет приниматься, должен быть включен в список входящих типов в определении типа порта.

Операция **receive** удаляет верхнее сообщение из очереди соответствующего входящего порта тогда и только тогда, когда это верхнее сообщение удовлетворяет всем критериям сопоставления, связанным с этой операцией **receive**. Не должно происходить сцепления входящих значений с элементами выражения или с шаблоном.

Если сопоставление окажется неудачным, то верхнее сообщение не должно быть удалено из очереди порта; то есть, если операция **receive** будет использоваться как альтернатива выражения alt, и является неудачной, то выполнение тестового варианта должно продолжиться со следующей альтернативой выражения alt.

Соответствующие критерии связаны с типом и значением сообщения, которое будет получено. Тип и значение сообщения, которое будет получено, определены аргументом операции receive, то есть, могут или быть получены из определенного шаблона, или специфицируются как in-line. Дополнительная область типа в соответствующих критериях к операции **receive** должна использоваться, чтобы избежать любой двусмысленности типа получаемого значения.

ПРИМЕЧАНИЕ 1. – Кодовые признаки также участвуют в согласовании неявным способом, предотвращая производство декодером из полученного сообщения абстрактного значения, закодированного способом, отличным от способа, который определяется атрибутами.

В случае соединений один-многие операция **receive** может быть ограничена только одним участником связи. Данное ограничение обозначается при помощи ключевого слова **from**.

ПРИМЕР 1:

```
   MyPort.receive(MyTemplate(5, MyVar));// Соответствие сообщения, которое полностью
удовлетворяет
                          // условиям, определенным в шаблоне MyTemplate порта MyPort.

   MyPort.receive(A<B);             // Соответствие Булева значения, которое зависит Matches a
                            //от результата A<B

   MyPort.receive(integer:MyVar);          // Соответствие целого значения и MyVar
                           // на порту MyPort

   MyPort.receive(MyVar);                // Альтернатива предыдущему примеру
```

```
   MyPort.receive(charstring:"Hello")from MyPeer;    // Соответствие строки символов "Hello" из
MyPeer
```

Если соответствие успешно, то удаленное из очереди порта значение может быть сохранено в переменной, а также может быть восстановлена и сохранена в переменной. Это обозначается символом '-> ' и ключевым словом **value**.

Можно также запросить и сохранить ссылку компонента или адрес отправителя сообщения. Это обозначается ключевым словом **sender**.

ПРИМЕЧАНИЕ 2. – Когда сообщение получено на связанному порту, только ссылка компонента сохраняется в следующем ключевом слове **sender** , а тестовая система также должна внутренне сохранить название компонента, если таковой имеется (чтобы использоваться в протоколе).

ПРИМЕР 2:

```
MyPort.receive(MyType:?) -> value MyVar; // Значение принятого сообщения присвоено
                                  // MyVar.

MyPort.receive(A<B) -> sender MyPeer;  // Адрес отправителя просвоено MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;

        // Полученное значение сообщения  и адрес отправителя сохранены в  в MyPeer.
```

23.2.2.1 Прием любого сообщения

Операция **receive** без списка аргументов для критериев сопоставления типов и значений сообщения, которое необходимо принять, удаляет это сообщение из верхнего уровня очереди входящего порта (если таковая имеется), если выполняются все другие критерии сопоставления.

Сообщение, принятое с помощью *ReceiveAnyMessage*, не присваивается переменной.

ПРИМЕР:

```
MyPort.receive;      // Удаляет верхнее значение из MyPort.

MyPort.receive from MyPeer;     ;    // Удаляет верхнее сообщение из MyPort,



                //если отправитель MyPeer

MyPort.receive -> sender MySenderVar; // Удаляет верхнее сообщение из MyPort и присваивает
                // адрес отправителя MySenderVar.
```

23.2.2.2 Получение в любом порту

Для получения сообщения в любом порту используется ключевое слово **any port**.

ПРИМЕР:

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

22.2.3 Операция Trigger

23.2.3.0 Общие положения

Операция **trigger** удаляет верхнее сообщение из соответствующей поступающей очереди порта. Если верхнее сообщение удовлетворяет критериям соответствия, то операция trigger ведет себя также как и операция receive. Если верхнее главное сообщение не удовлетворяет критериям соответствия, то оно должно быть удалено из очереди без какого-либо дальнейшего действия. Операция **trigger** должна использоваться только на основанных на сообщениях (или смешанных) портах, и тип значений, который будет получен, должен быть включен в список входящих типов определения типа порта.

ПРИМЕЧАНИЕ. – Примечание 1 в 22.2.2.0 также действительно для операции **trigger**.

Операция **trigger** может использоваться как автономное выражение в описании поведения. В этом последнем случае операция **trigger** рассматривается как краткая форма для выражения **alt** только с одной альтернативой; то есть, она имеет семантику блокирования, и поэтому обеспечивает способность ожидания следующего сообщения, соответствующего указанному шаблону или значению в этой очереди.

ПРИМЕР 1:

```
MyPort.trigger(MyType:?);
//Определяет, что операция trigger вызовет при приеме первого полученного сообщения типа MyType
с //произвольным значением в порту MyPort.
```

Операция **trigger** требует имя порта, соответствия критериям для типа и величины, опционального ограничения **from** (то есть, выбор партнера связи) и опциональное присваивание соответствующего сообщения и компонента отправителя переменным.

ПРИМЕР 2:

```
MyPort.trigger (MyType : ?) from MyPartner;
    // Triggers на приеме первого сообщения типа MyType в порту MyPort.

MyPort.trigger (MyType : ?) from MyPartner -> value MyRecMessage;
// Этот пример почти идентичен предыдущему примеру. Добавлено, что сообщение, которое
// инициировано, то есть удовлетворяет всем критериям сопоставления, запоминается в
// переменной MyRecMessage.

MyPort.trigger (MyType : ?) -> sender MyPartner;
// Этот пример почти идентичен первому примеру. Кроме этого, будет запрошена и
// сохранена в переменной MyPartner ссылка на компонент.

MyPort.trigger (integer : ?) -> value MyVar sender MyPartner;
// Triggers на приеме произвольного значения целого числа, которое впоследствии будет
//сохранено в
//переменной MyVar. Ссылка компонента отправителя будет сохранена в переменном MyPartner.
```

23.2.3.1 Запуск любого сообщения

Операция **trigger** без списка аргументов инициирует прием любого сообщения. Поэтому ее смысл идентичен смыслу приема любого сообщения. Сообщение, принятое с помощью *TriggerOnAnyMessage*, не присваивается переменной.

ПРИМЕР:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 Запуск в любом порту

Для запуска сообщения в любом порту используется ключевое слово **any port**.

ПРИМЕР:

```
any port.trigger
```

23.3 Связь, основанная на процедуре

23.3.0 Общие положения

Принцип основанной на процедуре коммуникации есть вызов процедур в отдаленных объектах. TTCN-3 поддерживает *блокированную* и *неблокированную* связь, основанную на процедуре. Блокированная связь, основанная на процедуре, блокируется на вызывающей и вызываемой стороне, тогда как неблокированная связь, основанная на процедуре, блокируется только на вызываемой стороне. Сигнатура процедур, которые используются для того, чтобы не блокировать связь, основанную на процедуре, должна быть определена согласно правилам в пункте 13.

Схема блокирующей связи, основанной на процедуре, показана на Рисунке 12. CALLER вызывает удаленную процедуру в CALLEE при помощи операции **call**. CALLEE принимает запрос посредством операции **getcall** и реагирует или при помощи операции **reply**, чтобы ответить на запрос, или вызывает (операция **raise**)

исключение. CALLER обращается с ответом или особым состоянием при использовании операций **`catch`** или **`getreply`**. На рисунке 12 блокирование CALLER и CALLEE обозначено посредством пунктирных линий.

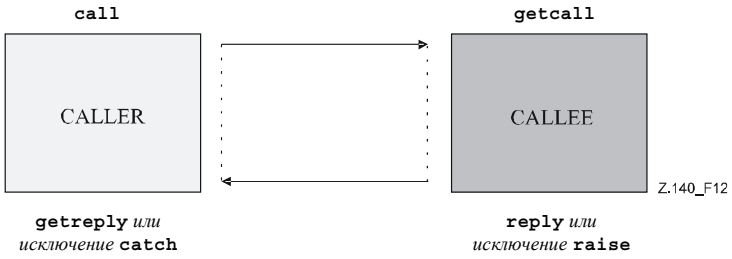


Рисунок 12/Z.140 – Иллюстрация блокированной связи, основанной на процедуре

Схема неблокирующей связи, основанной на процедуре, показана на рисунке 13. CALLER называет удаленную процедуру в CALLEE при помощи операции **call** и продолжает ее выполнение, то есть, не ждет ответа или исключения. CALLEE принимает вызов посредством операции **getcall** и исполняет требуемую процедуру. Если выполнение неудачно, CALLEE может создать исключение с целью проинформировать CALLER. CALLER может обработать исключение при помощи операции **catch** в выражении **alt**. На рисунке 13 блокирование CALLEE, до конца обработки вызова и возможное создание особого состояния обозначены посредством прерывистой линии.

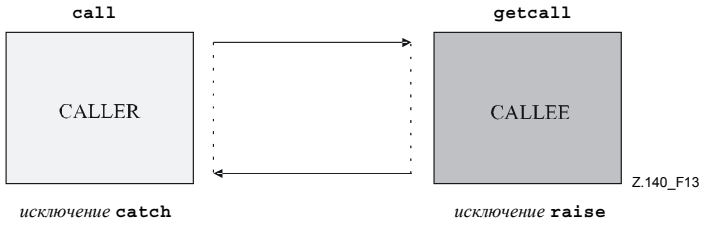


Рисунок 13/Z.140 – Иллюстрация неблокированной связи, основанной на процедуре

23.3.1 Операция Вызова

23.3.1.0 Общие положения

Операция call используется для того, чтобы указать, что тестовый компонент вызывает процедуру в SUT или в другом тестовом компоненте. Операция call должна использоваться только на основанных на процедуре (или смешанных), портах. Определение типа порта, в котором имеет место операция call, должно включать название процедуры в ее out или inout списке; то есть, должно быть разрешено вызывать эту процедуру в этом порту.
Информация, которая должна передаваться в передаваемой части операции **`call`**, должна иметь подпись, которая может или быть определенна в форме шаблона подписи или определяется in-line. Все in и inout параметры подписи должны иметь определенное значение; то есть, использование механизма соответствия, такого как *AnyValue*, запрещено.
Аргументы подписи операции **`call`** не используются, чтобы восстановить имена переменной для параметров **`inout`** и **`out`**. Фактическая привязка возвращаемого значения процедуры и параметров значений **`out`** и **`inout`** переменным должна явно быть сделана в ответе и в части обработке исключения операции call посредством операций catch и getreply. Это позволяет использовать шаблоны подписи в операции **`call`** таким же образом, как шаблоны могут использоваться и для типов.

ПРИМЕР 1:

```
// Дано …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// вызов MyProc
MyPort.call(MyProc:{ -, MyVar2}) {     // in-line шаблон сигнатуры для вызова MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}

// … и другой вызов MyProc
MyPort.call(MyProcTemplate) {            // использование шаблона сигнатуры для вызова
MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

В случаях связей "один ко многим" партнер коммуникации должен быть определен уникальным образом. Это должно быть обозначено с помощью ключевого слова **to**.

ПРИМЕР 2:

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

23.3.1.1 Обработка ответов и исключений для Call

В случае неблокированной связи основанной на процедуре (см. 23.3.1.4), обработка исключений операции **call** производится применением операции **catch** (см. 23.3.6), как альтернативы выражения **alt**.
Если используется вариант **nowait** (см. 23.3.1.2), то обработка ответов или исключений операции **call** производятся с помощью операций **getreply** (см. 23.3.4) и **catch** (см. 23.3.6), как альтернативы выражения **alt**.
В случае блокированной связи, основанной на процедуре, обработка ответов или исключений операции вызова производится в части обработки ответа и исключения операции **call** посредством операций **getreply** (см. 23.3.4) и **catch** (см. 23.3.6).
Часть обработки ответа и исключения операции **call** выглядят подобно телу выражения **alt**. Она определяет ряд альтернатив, описывая возможные ответы и исключения при вызове. Для вызванной процедуры выбор альтернатив должен быть основан только на операциях **getreply** и **catch**. Свободные операции **getreply** и **catch** должны рассматривать только ответы от и исключения, возникшие в вызванной процедуре.
Использование ветвей **else** и запросов altstep не разрешены.
В случае необходимости, возможно разрешать/блокировать альтернативу посредством **,** помещенного между «[]» скобками альтернативы.
Часть обработки ответа и исключений операции вызова выполняется как выражение **alt** без какого-либо активного значения по умолчанию. Это означает, что соответствующий мгновенный кадр включает всю информацию, необходимую для того, чтобы оценить (факультативно) Булевы блокировки, и может включать главный элемент (при наличии такового) порта, из которого процедуру вызвали, а также может включать исключение таймаута, созданное (факультативным) таймером, который контролирует вызов (см. 23.3.1.2).
При оценке Булевых блокировок контролирующих альтернативы в части обработки ответа и исключений могут быть побочные эффекты. Чтобы избежать неожиданных побочных эффектов, должны применяться те же самые правила, как и у Булевых блокировок в выражении **alt** (см. 20.1.1).

ПРИМЕР:

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

// Вызов MyProc3
MyPort.call(MyProc3:{ -, true }) to MyPartner {

  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

  [] MyPort.catch(MyProc3, MyExceptionOne) {
    setverdict(fail);
    stop;
     }
  [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
    setverdict(inconc);
     }
```

```
    [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

23.3.1.2 Обработка исключений таймаута вызова

Операция **call** может факультативно включать таймаут. Таймаут определяется явной величиной или постоянной типа **float** и определяет отрезок времени после того, как операция **call** запущена и до того, как тестовой системой будет сгенерировано исключение **timeout**. Если же в операции **call** нет никакой информации о значении таймаута, то никакого исключения **timeout** сгенерировано не будет.

ПРИМЕР 1:

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

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

  [] MyPort.catch(timeout) {           // исключение таймаута 20 мс
    setverdict(fail);
    stop;
     }
}
```

Использование в операции **call** ключевого слова nowait вместо значения исключения таймаута позволяет вызывать процедуру продолжения, не ожидая ни ответа, ни исключения, возникшего в соответствии с вызванной процедурой или особым случаем таймаута.

ПРИМЕР 2:

```
MyPort.call(MyProc:{5, MyVar}, nowait);  // Вызываемый тестовый компонент продолжит
                                 // свое выполнение без ожидания
                                 // завершения MyProc
```

При использовании ключевого слова **nowait**, возможный ответ или исключение вызванной процедуры должны быть удалены из очереди порта, при помощи операций **getreply** или **catch** в последующем выражении **alt**.

23.3.1.3 Вызов процедуры блокированния без значения возвращения, параметров out, параметров inout и исключений

Процедура блокирования может не иметь ни возвращаемых значений, ни параметров **out** и **inout** и может не вызывать исключения. У операции call для таких процедур должны также быть часть обработки ответа и исключения, чтобы обработать блокировку единым образом.

ПРИМЕР:

```
// Дано …
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// a call of MyBlockingProc
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

23.3.1.4 Запрос неблокируемых процедур

У неблокируемой процедуры нет ни параметров **out** и **inout**, ни возвращаемого значения, а свойство неблокируемости указываются в соответствующем определении сигнатуры посредством ключевого слова **noblock**.

Операция **call** для неблокируемой процедуры не должна иметь части обработки ответа и исключения, не должна вызывать исключение таймаута и не должна использовать ключевое слово **nowait** .

Возможные исключения, вызванные неблокируемой процедурой, должны быть удалены из очереди порта путем использования операций **catch** в последующих выражениях **alt** или **interleave**.

23.3.1.5 Однонаправленные, многонаправленные и всеобщие запросы процедур

Как и для операции **send**, TTCN-3 также поддерживает однонаправленные, многонаправленные и широковещательные запросы процедур. Это может быть сделано так же, как описано в 23.2.1.1; то есть, аргумент пункта **to** операции **call**, для однонаправленных запросов - адрес одного объекта получения (или может быть опущен в случае непосредственных связей); для многонаправленных запросов - списка адресов ряда

приемников; и, для широковещательных запросов, ключевого слова **all component**. В случае непосредственных связей, пункт **to** может быть опущен, потому что объект получения уникально идентифицируется структурой системы.
Обработка ответов и исключений для блокированной или неблокированной однонаправленной операции **call** была объяснена в 23.3.1.1 23.3.1.4. многонаправленная или широковещательная операция **call** могут приводить к нескольким ответам и исключениям от различных партнеров коммуникации.
В случае многонаправленной и широковещательной операции **call** процедуры неблокирования, все исключения, которые могут быть вызваны различными партнерами коммуникации, могут быть обработаны в последующих выражениях **catch**, **alt** или **interleave**.
В случае многонаправленной и всеобщей операции **call** процедуры блокирования, существуют два варианта: В первом случае, только один ответ или исключение обрабатывается в подпрограмме обработки ответа и исключения операции **call**. Затем дальнейшие ответы и исключения могут быть обработаны в последующем выражениии **alt** или **interleave**. Во втором случае несколько ответов или исключений обрабатываются при помощи повторных выражений в одном или более блоке выражений и деклараций части обработки ответа и исключений операции call: выполнение повторного утверждения вызывает повторную оценку самой сущности вызова.

ПРИМЕЧАНИЕ. – Во втором случае, пользователь должен обращаться с множеством повторений.

ПРИМЕР 1:

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to (MyPeerOne, MyPeerTwo) {// многонаправленный вызов MyProc
    // Обрабатывает ответ от MyPeerOne
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
    if (first) { first := false; repeat; }
    :
    }
    // Обрабатывает ответ от MyPeerTwo
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
    if (first) { first := false; repeat; }
    :
    }
    [] MyPort.catch(timeout) {            // исключение таймаута после 20мс
    setverdict(fail);
    stop;
    }
}

alt {
  [] MyPort.getreply(MyProc:{?, ?}) {  // Обрабатывает все другие ответы всеобщего вызова
    repeat
  }
}
```

В случае многонаправленной и широковещательной операции **call** процедуры блокирования, где используется ключевое слово **nowait** , все ответы и исключения должны быть обработаны в последующих выражениях **alt** или **interleave**.

ПРИМЕР 2:

```
MyPort.call(MyProc:{5,MyVar}) to (MyPeer1, MyPeer2) nowait;  // Многонапрвленный вызов MyProc

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { }  // Обрабатывает ответ MyPeer1
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { }  // Обрабатывает ответ MyPeer2
}
```

23.3.2 Операция Getcall

23.3.2.0 Общие положения

Операция **getcall** используется для указания, что тестовый компонент принимает вызов от SUT или другого тестового компонента. Операция **getcall** используется только в портах на базе процедур (или смешанных), а сигнатура вызова процедуры, который следует принять, должна быть включена в список разрешенных входящих процедур в определении типа порта.

Операция **getcall** удаляет первый (верхний) вызов из очереди входящего порта тогда и только тогда, когда выполнены критерии сопоставления, связанные с операцией **getcall**. Эти критерии сопоставления относятся к сигнатуре вызова, который должен быть обработан, и к партнеру по связи. Критерии сопоставления для сигнатуры могут быть либо определены "инлайн", либо выделены из шаблона сигнатуры.

В случае соединений типа "один ко многим" операция **getcall** может быть ограничена каким-либо определенным партнером по связи. Это ограничение должно быть обозначено использованием ключевого слова **from**.

ПРИМЕР 1:

```
  MyPort.getcall(MyProc: MyProcTemplate(5, MyVar)); // MyPort принимает вызов в MyProc
My.Port.getcall (MyProc : {5, MyVar}) from MyPeer; // MyPort принимает вызов в MyProc от MyPeer
```

Аргумент сигнатуры операции **getcall** не должен использоваться для передачи имен переменных для параметров **in** и **inout**. Назначение значения параметра **in** и **inout** переменным должно быть сделано в части назначения операции **getcall**. Это позволяет использовать шаблоны подписи в операциях **getcall** так же, как шаблоны для типов.

(Факультативная) часть назначения операции **getcall** включает назначение значения параметра **in** и **inout** переменным и поиск адреса компонента запроса. Значение части назначения не должно использоваться с операцией getcall . Ключевое слово **param** используется, чтобы восстановить значение параметра call.

Ключевое слово **sender** используется в случаях, когда требуется запросить адрес передатчика (например, для адресации ответа или особого состояния вызывающей стороне в конфигурации "один ко многим").

ПРИМЕР 2:

```
  MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
  // Значение параметра in or inout MyProc назначены MyPar1Var и MyPar2Var.
  MyPort.getcall (MyProc : {5, MyVar}) -> sender MySenderVar;
  //  Принимает вызов MyProc в MyPort с  параметрами in или inout 5 и MyVar.
  // Адрес вызывающего абонента восстановлен и сохранен в MySenderVar.
Следующие примеры getcall  показывают возможности использовать соответствующие признаки и
опустить дополнительные части, которые могут быть незначительными для тестовой спецификации.
MyPort.getcall (MyProc : {5, Var}) -> param (MyVar1, MyVar2) sender
MySenderVar;
  MyPort.getcall (MyProc : {5, ?}) -> param (MyVar1, MyVar2);

  MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// Значение первого параметра inout не важно или не используется
// Следующие примеры поясняют возможности присвоения значений параметров in и inout
// переменным. Следующая сигнатура предназначена для вызываемой процедуры.

signature MyProc2 (in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall (MyProc2 : {?, ?, 3, -, ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
  param (MyVarIn1, MyVarIn2, MyVarIn3, -, -, MyVarInout1);
// Параметры  A, B и C присвоены переменным MyVarA, MyVarB, и MyVarE.
// Параметры out D  не требуется учитывать

MyPort.getcall (MyProc2 : { ?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);

// Альтернативная нотация для присвоения значения параметров in и inout переменным.
// Отметим, что имена в списке присвоений ссылаются на имена, использованные в
// сигнатуре из MyProc2.

MyPort.getcall (MyProc2 : {1, 2, 3, -,  *}) -> param (MyVarE:= E);
// Для дальнейшего выполнения тестового примера необходимо только значение параметра
// inout.
```

23.3.2.1 Прием любого вызова

Операция **getcall** без списка аргументов для критериев сопоставления сигнатуры будет удалять первый вызов из очереди во входящем порту (если таковая имеется), если выполняются все другие критерии сопоставления. Параметры вызовов, принятых с помощью *AcceptAnyCall*, не присваиваются переменной.

ПРИМЕР:

```
MyPort.getcall;         // Удаляет вызов из верхнего уровня очереди в MyPort.
MyPort.getcall from MyPartner; //  Удаляет вызов от MyPartner  из порта MyPort
MyPort.getcall -> sender MySenderVar;
// Удаляет вызов из порта MyPort  и адрес вызывающего объекта
```

23.2.4.2 Прием в любом порту

Для обозначения операции **getcall** в любом порту используется ключевое слово **any**. Например:

ПРИМЕР:

```
any port.getcall (MyProc)
```

23.3.3 Операция Ответа

Операция **reply** используется для ответа на ранее принятый вызов в соответствии с сигнатурой процедуры. Операция **reply** должна использоваться только для порта, основанного на процедуре (или смешанного). Определение типа порта должно включать название процедуры, которой принадлежит операция **reply**.

ПРИМЕЧАНИЕ. – Отношение между принятым вызовом и операцией **reply** не может всегда быть проверено статически. Для проверки можно определить операцию **reply** без соответствующей операции **getcall**.

Часть значения операции reply состоит из ссылки сигнатуры с соответствующим списком фактических параметров и (факультативно) возвращаемым значением. Сигнатура может или быть определена в форме шаблона сигнатуры, или может быть определена как in-line. Все параметры out и inout сигнатуры должны иметь определенную величину; то есть, не разрешено использование механизмов соответствия, таких как *AnyValue* .

Ответы на одну или более операций **call** могут быть отправлены одному, нескольким или всем подобным объектам, связанным с адресуемым портом. Это может быть определено таким же образом, как описано в 23.2.1.1. Это означает, что аргумент выражения **to** операции **reply** для однонаправленных ответов является адресом единственного объекта получения; для многонаправленных ответов состоит из списка адресов набора приемников; и для широковещательных ответов состоит из ключевых слов **component all**.

В случае непосредственных связей, выражение **to** может быть опущено, так как что объект получения уникально идентифицируется структурой системы.

Если вызывающему абоненту должно возвращаться значение, то это должно быть явно заявлено с помощью ключевого слова **value** .

ПРИМЕР:

```
MyPort.reply(MyProc2:{ - ,5});          // Отвечает на принятый вызов от MyProc2.

MyPort.reply(MyProc2:{ - ,5}) to MyPeer; // Отвечает на принятый вызов от MyProc2 из MyPeer

MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2); // Многонаправленный ответ MyPeer1 и
MyPeer2

MyPort.reply(MyProc2:{ - ,5}) to all component;   // Всеобщий ответ всем объектам
,соединенным
                                                  // с MyPort

MyPort.reply(MyProc3:{5,MyVar} value 20);// Отвечает на принятый вызов MyProc3.
```

23.3.4 Операция Getreply

23.3.4.0 Общие положения

Операция **getreply** используется для обработки ответов на ранее вызванные процедуры. Операция **getreply** используется только в порту на базе процедур (или смешанном). Определение типа порта должно включать название процедуры, которой принадлежит операция **getreply** .

Операция **getreply** должна удалить верхний ответ из поступающей очереди порта в том и только том случае, когда выполнены критерии соответствия, связанные с операцией **getreply**. Эти критерии соответствия зависят от сигнатуры процедуры, которая должна быть обработана, и партнера коммуникации. Соответствующие критерии для сигнатуры могут или быть определены в процессе или получены из ее шаблона.

Противопоставление принятой величины возвращения может быть определено при использовании ключевого слова **value** .

Операция **getreply** может быть ограничена определенными партнерами коммуникации в случае связей "один ко многим". Это ограничение должно быть обозначено при использовании ключевого слова **from** .

ПРИМЕР 1:

```
MyPort.getreply(MyProc:{5, ?} value 20); // Прием ответа MyProc с двумя out или
                                 // inout и возврат значения 20
```

```
MyPort.getreply(MyProc2:{ - , 5}) from MyPeer; // Прием ответа MyProc2 от MyPeer
```

Аргумент сигнатуры операции **getreply** не должен использоваться в именах переменной для параметров **inout** и **out**. Назначение величин параметров **inout** и **out** переменным должно быть сделано в части назначения операции **getreply**. Это позволяет использовать шаблоны сигнатур в операциях **getreply** таким же образом, как используются шаблоны для типов.

(Факультативная) часть назначения операции **getreply** включает назначение значений параметров **inout** и **out** переменным и поиску адреса отправителя ответа. Для получения возвращаемого значения используется ключевое слово **value**, а ключевое слово **param** используется, чтобы восстановить значение параметра ответа. Когда требуется восстанавливать адрес отправителя используется ключевое слово **sender**.

ПРИМЕР 2:

```
 MyPort.getreply(MyProc1:{?, ?} value ?) -> value  MyRetValue param(MyPar1,MyPar2);
// Возвращенное значениеприсвоено переменной MyRetValue и значения
// двух out или inout параметров присвоены переменным MyPar1 и MyPar2.

 MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender
MySender;
 // Для дальнейшего выполнения теста не учитывается значение первого параметра и
 // адрес компонента-отправителя извлекается и сохраняется в переменной MySender.

// Следующие примеры описывают некоторые возможности назначения значений параметра out и  inout
//переменным. Для процедуры, которая должна быть вызвана, принята следующая сигнатура
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 Получение любого ответа

Операция **getreply** без списка аргументов для критериев сопоставления сигнатуры удаляет сообщение **reply** из верхнего уровня очереди во входящем порту (если таковая имеется), если выполняются все остальные критерии сопоставления. Параметры или возвращаемые значения ответов, принятых с помощью *GetAnyReply*, не присваиваются к переменной. Если *GetAnyReply* использована в части ответа и обработки исключения операции **call**, то она должна только рассматривать ответы процедуры, выванной операцией **call**.

ПРИМЕР:

```
MyPort.getreply;// Удаляет верхний ответ из MyPort.

MyPort.getreply from MyPeer; // Удаляет верхний ответ принятый от MyPeer по MyPort.

MyPort.getreply -> sender MySenderVar;  // Удаляет верхний ответ принятый по MyPort и
восстанавливает
                          // адрес объекта-отправителя
```

23.3.4.2 Чтение ответа в любом порту

Для чтения ответа в любом порту используется ключевое слово **any port**.

ПРИМЕР:

```
any port.getreply (MyProc)
```

23.3.5 Операция raise

Операция **raise** используется для генерации исключения. Исключение должно быть вызвано только в порту, основанном на процедуре (или смешаном). Исключение – реакция на принятую процедуру вызова, результат которого приводит к возникновению исключительного, особого события. Тип исключения должен быть определен в сигнатуре вызванной процедуры. Определение типа порта должно включать в его список допущенных запросов процедуры название процедуры, которой принадлежит исключение.

ПРИМЕЧАНИЕ. – Отношение между принятым вызовом и операцией raise не может всегда проверяться статически. Для проверки, можно определить операцию raise без соответствующей операции getcall.

Часть, отвечающая за значение операции **raise**, состоит из ссылки сигнатуры, сопровождаемой значением исключения.

Исключения определены как типы. Поэтому значение исключения может либо являться производным шаблона, или же являться результатом выражения (которое, конечно, может быть явной величиной). Факультативное поле типа в спецификации значения операции **raise** должна быть использована в случаях, когда это необходимо, чтобы избежать любой неоднозначности типа посылаемого значения.

Исключения одной или более операциям **call** могут быть высланы в один, несколько или всем подобным объектам, связанным с портом, к которому обращаются. Это может быть определено так же, как описано в 23.2.1.1. Это означает, что аргумент пункта **to** операции **raise** является, для однонаправленных исключений адресом единственного объекта получения; для исключений многонаправленных является списком адресов набора приемников; и для широковещательных исключений – состоит из ключевых слов **all component**.

В случае непосредственных связей элемент **to** может быть опущен, потому что объект получения уникально идентифицирован структурой системы.

ПРИМЕР:

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
// Возбуждение исключения со значением, являющимся результатом арифметического выражения
// на MyPort

MyPort.raise(MyProc, integer:5});// Возбуждение исключения с целым значением 5 для MyProc

MyPort.raise(MySignature, "My string") to MyPartner;
// Возбуждение исключения со значением "My string" на MyPort для MySignature и
// посылка его к MyPartner

MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
// Возбуждение исключения со значением "My string" на MyPort для MySignature и посылка его к
MyPartnerOne и
// MyPartnerTwo (то есть – многонаправленная связь)

MyPort.raise(MySignature, "My string") to all component;
// Возбуждение исключения со значением "My string" на MyPort для MySignature и посылка его
// всем объектам, связанным с MyPort (то есть – всеобщая связь)
```

23.3.6 Операция Catch

23.3.6.0 Общие положения

Операция **catch** используется, чтобы получить исключения вызванные тестовым компонентом или SUT в качестве реакции на вызов процедуры. Операция **catch** используется только в портах на базе процедур (или смешанных). Тип полученного исключения указывается в сигнатуре процедуры, которая вызвала исключение. Исключения определены как типы и таким образом могут рассматриваться как сообщения; например, могут использоваться шаблоны чтобы отделить различные значения для того же самого типа исключения.

Операция **catch** удаляет верхнее исключение из соответствующей поступающей очереди порта тогда и только тогда, когда оно удовлетворяет всем критериям соответствия, связанным с операцией **catch**. Не должно происходить никакой привязки входящих величин к элементам выражения или к шаблону. Назначение величин исключения переменным должно быть сделано в части назначения операции **catch**.

Операция **catch** может быть ограничена для определенного партнера по коммуникации в случае связей "один ко многим". Это ограничение должно быть обозначено при использовании ключевого слова **from** .

ПРИМЕР 1:

```
MyPort.catch(MyProc, integer: MyVar); // Захватывает целое исключение значения
MyPort.catch(MyProc, MyVar);     //Альтернатива предыдущему примеру.
MyPort.catch(MyProc, A<B);     // Захватывает булево исключениеMyPort.catch(MyProc, MyType:{5,
MyVar}); // Определение шаблона величины исключения на лету.
MyPort.catch(MyProc, charstring:"Hello")from MyPeer; // Захват исключения"Hello" из MyPeer
```

(Факультативная) часть назначения операции **catch** включает назначение величины исключения и поиск адреса компонента запроса. Ключевое слово **value** используется, чтобы восстановить величину исключения, а ключевое слово **sender** - когда требуется восстанавливать адрес отправителя.

ПРИМЕР 2:

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// Захватывает исключение Catches an exception от MyPartner и присваивает его значение MyVar.
MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Захватывает исключение, присваивает его значение MyVarTwo и восстанавливаеть
// адрес отправителя.
```

Операция **catch** может быть частью ответа и процедуры обработки исключения операции **call** или использоваться, чтобы определить альтернативу в выражении **alt**. Если операция **catch** используется в части принятия операции **call**, информация о названии порта и ссылке сигнатуры, указывающая на процедуру, которая вызвала исключение, является избыточной, потому что эта информация следует из операции **call**. Однако по причинам удобочитаемости (например, в случае сложных выражений **call**) эта информация должна быть повторена.

23.3.6.1 Исключения таймаута

Имеется одно особое исключение **timeout**, которое может быть получена операцией **catch**. Исключение **timeout** является вариантом действия на случай, когда вызванная процедура не отвечает и не порождает исключения в пределах заранее определенного времени.

ПРИМЕР:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {    //исключение таймаута после 20 мс
    setverdict(fail);
    stop;
     }
}
```

Захват исключений **timeout** ограничивается частью "обработка исключения" вызова. Для операции **catch**, которая обрабатывает исключение **timeout**, не разрешены дальнейшие критерии сопоставления (включая часть **from**) и часть "присвоение".

22.2.6.2 Захват любого исключения

Операция **catch** без списка аргументов позволяет фиксировать любое действительное исключение. В наиболее общем случае не используется ключевое слово **from** . Значения исключения, принятые *CatchAnyException*, не должны назначаться переменной. Если *CatchAnyException* будет использоваться в части ответа и обработки исключения операции **call**, то оно должно рассматривать только исключения, вызванные в соответствии с процедурой, вызванной операцией **call**. *CatchAnyException* будет также захватывать исключение **timeout** .

ПРИМЕР:

```
MyPort.catch;
MyPort.catch from MyPartner;
MyPort.catch -> sender MySenderVar;
```

23.3.6.3 Захват в любом порту

Для захвата исключения в любом порту используется ключевое слово **any**.

ПРИМЕР:

```
any port.catch (timeout)
```

23.4 Операция Check

23.4.0 Общие положения

Операция **check** является обобщенной операцией, которая позволяет считывать доступную информацию верхнего элемента в очереди *входящего* порта на базе сообщений или на базе процедур, не удаляя этот верхний элемент из очереди. Операция **check** должна обрабатывать значения определенного типа в портах на базе сообщений и различать вызовы, подлежащие приему, особые состояния, подлежащие фиксации, и ответы от предыдущих вызовов в портах на базе процедур.

Операции приема **receive, getcall, getreply** и **check** вместе с их частями "сопоставление" и "присвоение" используются операцией **check** для определения состояния, которое необходимо проверить, и для выделения значения или значений его параметров, если требуется.

Проверяться должен *верхний* элемент в очереди входящего порта (посмотреть *внутрь* очереди невозможно). Если очередь пуста, то операция **check** будет неуспешной. Если очередь не пуста, то берется копия верхнего элемента и над этой копией выполняется операция приема, указанная в операции **check**. Операция **check**

будет неуспешной, если принимающая операция окажется неуспешной, то есть не будут выполнены критерии сопоставления. В этом случае удаляется *копия* верхнего элемента очереди, а выполнение теста продолжается обычным образом, то есть определяется следующие выражение или альтернатива для операции проверки. Операция **check** будет успешной, если принимающая операция окажется успешной.

Неправильное использование операции **check**, например проверка особого состояния в порту на базе сообщений, приводит к ошибке тестового примера.

ПРИМЕЧАНИЕ. – В большинстве случаев правильность использования операции проверки может быть проверена статически, то есть до/в течении трансляции.

ПРИМЕР:

```
MyPort1.check(receive(5)); // Проверяет целое сообщение значения 5.

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Проверяет вызов MyProc в порту MyPort2 от MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Проверяет ответ процедуры MyProc в MyPort2 где возвращенное значение 20 и
// значения из двух параметров out или inout есть  5 и значение MyVar.

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 Проверка любой операции

Операция **check** без списка аргументов позволяет проверку, есть ли в очереди входящего порта что-то, ожидающее обработки. Проверка любой операции позволяет различать разные передатчики (в случае соединений типа "один ко многим") с помощью раздела **from** и запрашивать передатчик с помощью краткой части "присвоение" с разделом **sender**. В случае смешанных портов **check** проверяет любую операцию, как основанную на сообщении, так и основанную на процедуре, входной очереди смешанного порта. Если **check** какой-нибудь операции соответствует обеим входным очередям смешанного порта, информации, связанной с основанной на процедуре очередью, нужно уделить первостепенное внимание, то есть, возвращенной в результате **check** какой-либо операции. Например, если основанное на сообщении и основанные на процедуре входные очереди смешанного порта не пусты, и информация отправителя должна быть восстановлена посредством **check** любой операции, то должен быть выдан отправитель запроса, ответа или исключения в основанной на процедуре входной очереди.

ПРИМЕЧАНИЕ. – Информация, связанная с основанной на сообщении входной очередью смешанного порта, может быть легко восстановлена при использовании операции **check** в комбинации с **receive** любой операции, например.

```
MyPort.check(receive) -> sender Mysender.
```

ПРИМЕР:

```
MyPort.check;

MyPort.check (from MyPartner);

MyPort.check (-> sender MySenderVar);
```

23.5 Проверка любых портов

Чтобы **проверить** любой порт, используйте ключевые слова **any port**. В случае **проверки** любой операции порта без аргумента входные очереди смешанных портов должны быть проверены, как определено в 23.4.1.

ПРИМЕР:

```
any port.check;
```

23.5 Управление портами связи

23.5.0 Общие положения

В TTCN-3 имеются следующие операции для управления портами на базе сообщений, на базе процедур и смешанными портами:

- `clear`: удаление содержимого из очереди входящего порта;
- `start`: удаление содержания очереди входящего порта и разрешение посылки и получения операций в порту;
- `stop`: запрещение посылки и отказ операциям получения согласовываться в порту;
- `halt`: немедленное запрещение операций посылок в порту и запрещение операций получения согласовываться с новыми сообщениями/запросами/ответами/исключениями, которые входят в очередь порта после того, как операция приостановки была выполнена. Записи, которые уже находятся в очереди, могут быть все еще обработаны.

23.5.1 Операция очистки порта

Операция **`clear`** удаляет содержимое *входящей* очереди в упомянутом порту. Если очередь в порту уже пуста, то эта операция не будет иметь каких-либо последствий.

ПРИМЕР:

```
MyPort.clear; // освобождает порт MyPort
```

23.5.2 Портовая операция Start

Если порт определен в качестве порта, позволяющего операции приема, такие, как **`receive, getcall`** и т. п., то операция **`start`** освобождает входящую очередь поименованного порта и запускает опрос трафика, проходящего через этого порт. Если порт определен в качестве порта, позволяющего операции передачи, то такие операции, как **`send, call, raise`** и т. п., также разрешается выполнять в этом порту.

ПРИМЕР:

```
MyPort.start; // запускает MyPort
```

По умолчанию все порты компонента должны запускаться без особой команды при создании компонента. Портовая операция **`start`** вызовет рестарт порта с удалением всех сообщений, ожидающих очереди во входящей очереди.

23.5.3 Портовая операция Stop

Если порт определен в качестве порта, позволяющего операции приема, такие, как **`receive`** и **`getcall`**, то операция **`stop`** останавливает опрос упомянутого порта. Если порт определен в качестве порта, позволяющего операции передачи, то получивший **`stop`** порт запрещает выполнение таких операций, как **`send, call, raise`** и т. п.

ПРИМЕР 1:

```
MyPort.stop;   // Останавливает MyPort
```

ПРИМЕЧАНИЕ. – Для прекращения режима ожидания сообщений в порту, необходимо, чтобы все полученные операции, определенные до операции остановки, были полностью выполнены – только после этого работа порта будет приостановлена.

ПРИМЕР 2:

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                  // полученная величина расшифрована, сравнена на
                                  // величина сохранена в переменной RecPDU
MyPort.stop;                      // Не определены принятые операции, после выполнения
                                  // операции остановки(если порт не перезапущен
                                  // последующей стартовой операцией ).
MyPort.receive (MyTemplate2);     // Эта операция не согласуется и заблокируется
                                  // (предполагается, что ничего
                                  //по умолчанию не активировано)
```

23.5.4 Операция порта halt

Если порт позволяет такие операции приема, как **receive**, **trigger** и **getcall,** то операция **halt** запрещает операциям приема сменяться сообщениями и процедурой вызова элементов, которые входят в очередь порта после выполнения операции **halt** в этом порту. Сообщения и элементы процедур вызова, которые уже были в очереди перед операцией **halt**, все еще могут быть обработаны операциями приема. Если порт позволяет операции посылки, то **halt** порта немедленно запрещает выполнять операции посылки, такие как **send**, **call**, **raise**, и т. д. Последующие операции **halt** не имеют никакого значения для состояния порта или его очереди.

ПРИМЕЧАНИЕ 1. – Когда операция выполнена, операция порта halt фактически помещает маркер после последнего входа в приемной очереди,. Записи перед маркером могут обрабатываться обычным образом. После того, как все записи в очереди перед маркером были обработаны, состояние порта эквивалентно остановленному состоянию.

ПРИМЕЧАНИЕ 2. – Если операция порта stop выполнена на остановленном порту прежде, чем все записи в очереди перед маркером были обработаны, дальнейшие операции приема немедленно отвергаются (то есть, маркер фактически перемещен в вершину очереди).

ПРИМЕЧАНИЕ 3. – Операция порта start на остановленном порту очищает все записи в очереди независимо от того прибыли они прежде или после выполнения операции порта halt. Она также удаляет маркер.

ПРИМЕЧАНИЕ 4. – Операция порта clear на остановленном порту очищает все записи в очереди независимо от того прибыли они прежде или после выполнения операции порта halt. . Это также фактически помещает маркер в верхнюю часть очереди.

ПРИМЕР:

```
MyPort.halt;                        // С этого момента запрещены посылки из Myport;
                                    // обработка сообщений в очереди все еще возможна.
MyPort.receive (MyTemplate1);       // Если сообщение уже было в очереди перед операцией halt
                                    // и оно соответствует MyTemplate1, то оно обрабатывается;
                                    // иначе блокируются операции приема.
```

23.6 Использование Any и All с портами

Ключевые слова **any** и **all** могут использоваться с операциями конфигурации и связи, как показано в таблице 18.

Таблица 18/Z.140 – Any и All с портами

Операция	Разрешено		Пример
	any	**all**	
receive, **trigger**, **getcall**, **getreply**, **catch**, **check**	да		**any port.receive**
connect / **map**			
start, stop, clear, halt		да	**all port.start**

24 Таймерные операции

24.0 Общие положения

TTCN-3 поддерживает ряд таймерных операций. Эти операции могут использоваться в тестовых примерах, функциях, altstep и управлении модулем.

Предполагается, что каждая единица области TTCN-3, в которой объявлены таймеры, поддерживает свой собственный *список работающих таймеров* и *список таймаута*, то есть, список всех таймеров, которые фактически работают и список всех переполненных таймеров. Списки таймаутов являются частью мгновенного снимка, взятого, когда тестовый случай был выполнен. Список таймаута обновляется, если таймер запущен в рассматриваемой области, остановлен, переполнен или если выполнена операция **timeout** .

ПРИМЕЧАНИЕ 1. – Список работающих таймеров и список таймаута - только концептуальные списки и не ограничивают применение таймеров. Могут также использоваться другие структуры данных, такие как набор, где доступ к событиям таймаута не ограничен, например, порядком, в котором произошли события таймаута .

ПРИМЕЧАНИЕ 2. – Предполагается, что для каждого тестового компонента существуют специальный список работающих таймеров и список таймаута , которые оперируют началом/остановкой таймера и событиями таймаута таймеров, объявленных в соответствующем определении типа компонента.

Когда таймер переполняется (концептуально немедленно перед обработкой мгновенного снимка ряда альтернативных событий), событие таймаута помещается в список таймаута области наблюдения, в которой был объявлен таймер. Таймер немедленно становится бездействующим. Может появиться только один вход для любого конкретного таймера в списке таймаутов наблюдаемого элемента, в которой таймер в некий момент был объявлен .

Все работающие таймеры должны быть автоматически удалены, если компонент будет явно или неявно остановлен.

Таблица 19/Z.140 – Обзор таймерных операций TTCN-3

Таймерные операции	
Команда	**Связанное ключевое слово или символ**
Запустить таймер	**start**
Остановить таймер	**stop**
Считать истекшее время	**read**
Проверить, работает ли таймер	**running**
Событие тайм-аута	**timeout**

24.1 Таймерная операция Start

Таймерная операция **start** используется для указания на то, что необходимо запустить таймер. Значения таймеров должны быть не отрицательными числами типа **float** (то есть больше или равны 0.0**)**. Когда таймер запущен, его название добавлено в список работающих таймеров (для данной контекстной единицы).

ПРИМЕР:

```
MyTimer1.start;         // MyTimer1 запущен с выдержкой по умолчанию.
MyTimer2.start (20E-3);// MyTimer2 запущен с выдержкой 20 мс.
// Элементы множеств таймера могут также быть запущены в цикле, например таймер t_Mytimer [5];
var float v_timerValues [5];
   for (var integer i := 0; i<=4; i:=i+1)
     { v_timerValues [i] := 1.0 }

   for (var integer i := 0; i<=4; i:=i+1)
     {t_Mytimer [i].start ( v_timerValues [i])}
```

Дополнительный параметр "значение таймера" используется, когда не дана выдержка по умолчанию или когда желательно отменить значение по умолчанию, указанное в объявлении таймера. Когда выдержка таймера отменяется, новое значение применяется только к текущему экземпляру таймера, любые последующие операции **start** для этого таймера, которые не указывают выдержку, должны использовать выдержку по умолчанию.

Старт таймера со значением 0.0 означает немедленное переполнение таймера. Запуская таймер с отрицательным значением, например, величина таймера - результат выражения, или без указанного величины таймера, должна вызвать ошибку времени выполнения.

Счетчик таймера считает от значения с плавающей запятой "нуль" (0,0) до максимума, установленного параметром "выдержка".

Операция **start** может быть применена к действующему таймеру, в этом случае таймер остановлен и перезапущен. Любая запись в списке таймаута для этого таймера должна быть удалена из списка таймаута.

24.2 Таймерная операция Stop

Операция **stop** используется для остановки работающего таймера и для удаления его из списка работающих таймеров. Остановленный таймер становится пассивным, а его истекшее время устанавливается в значение с плавающей запятой "нуль" (0,0).

Остановка пассивного таймера является действительной операцией, хотя она не дает какого-либо эффекта. Остановка переполненного таймера приводит к тому, что запись для этого таймера в списке таймаута должна быть удаленной. Может использоваться ключевое слово **all** , чтобы остановить все таймеры, которые видны в области наблюдения, в которой вызвали операцию **stop**.

ПРИМЕР:

```
MyTimer1.stop;   // останавливает MyTimer1
all timer.stop;  // останавливает все работающие таймеры
```

24.3 Таймерная операция Read

Операция **read** используется для запроса времени, которое прошло с момента запуска указанного таймера . Возвращенное значение должно иметь тип **float.**

ПРИМЕР:

```
var float Myvar;
MyVar := MyTimer1.read;  // присвоить переменной MyVar значение времени, которое
                         // прошло с момента запуска MyTimer1
```

Применение операции **read** к пассивному таймеру, то есть к таймеру, не перечисленному в списке работающих таймеров, вернет **float** значение "нуль"(0.0).

24.4 Таймерная операция Running

Таймерная операция **running** используется для проверки, занесен или нет таймер в список работающих таймеров для данной единицы контекста (то есть, что он был запущен и что его выдержка не истекла, а он не был остановлен). Эта операция выдает значение **true**, если таймер в списке, а в остальных случаях – значение **false**.

ПРИМЕР:

```
if (MyTimer1.running) { …  }
```

24.5 Операция Timeout

Операция **timeout** позволяет проверять переполнение таймера, или всех таймеров, в единице контекста тестового компонента или контроля за модулем, в котором была вызвана операция timeout.

Когда операция **timeout** обработана, в том случае, если указано название таймера, то производится поиск в списке согласно правилам поиска TTCN-3. Если обнаруживается событие таймаута, соответствующее названию таймера, то это событие удаляется из таймаут-списка, и операция **timeout** успешно завершена. **Timeout** не должен использоваться в булевом выражении, но он может использоваться, чтобы определить альтернативу в выражении **alt** или как автономное выражение в описании поведения. В последнем случае операция **timeout,** рассматривается как краткая форма выражения **alt** с единственной альтернативой, то есть, она имеет семантику блокирования, и поэтому обеспечивает способность пассивного ожидания таймаута таймера (ов).

ПРИМЕР 1:

```
MyTimer1.timeout;     // проверяет тайм-аут ранее запущенного таймера
                  // MyTimer1
```

Ключевое слово **any** вместе с операцией **timeout** (а не явно названный таймер) будет успешно выполнена в том случае, если список таймаутов не пуст.

ПРИМЕР 2:

```
any timer.timeout;   // проверяет тайм-аут любого ранее запущенного таймера
```

23.6 Сводка использования Any и All с таймерами

Ключевые слова **any** и **all** могут использоваться с таймерными операциями, как показано в таблице 20.

Таблица 20/Z.140 – Any и All с таймерами

Операция	Разрешено		Пример
	any	**all**	
Start			
stop		да	**all timer.stop**

read			
running	да		**if (any timer.running) { … }**
timeout	да		**any timer.timeout**

25 Операции тестового вердикта

25.0 Общие положения

Операции вердикта (см. таблицу 19) позволяют устанавливать и запрашивать вердикты с помощью операций **setverdict** и **getverdict** соответственно. Эти операции используются только в тестовых примерах, altstep и функциях.

Таблица 21/Z.140 – Обзор операций тестового вердикта TTCN-3

Операции тестового вердикта	
Команда	**Соответствующее ключевое слово или символ**
Установить местный вердикт	**setverdict**
Прочитать местный вердикт	**getverdict**

Каждый тестовый компонент в активной конфигурации должен поддерживать свой местный вердикт. Местный вердикт является таким объектом, который создается для каждого тестового компонента во время его создания. Он используется для слежения за индивидуальным вердиктом каждого тестового компонента (то есть в MTC и во всех до единого PTC).

25.1 Вердикт тестового примера

Кроме того, имеется глобальный вердикт тестового примера, экземпляр которого создается и обрабатывается тестовой системой, и который обновляется при завершении выполнения каждого тестового компонента (то есть MTC и всех PTC). Этот вердикт не доступен для операций **getverdict** и **setverdict**. Значение этого вердикта должно выдаваться тестовым примером, когда он завершает выполнение. Если выданный вердикт не сохраняется явно в управляющей части (например, путем присвоения переменной), то он теряется.

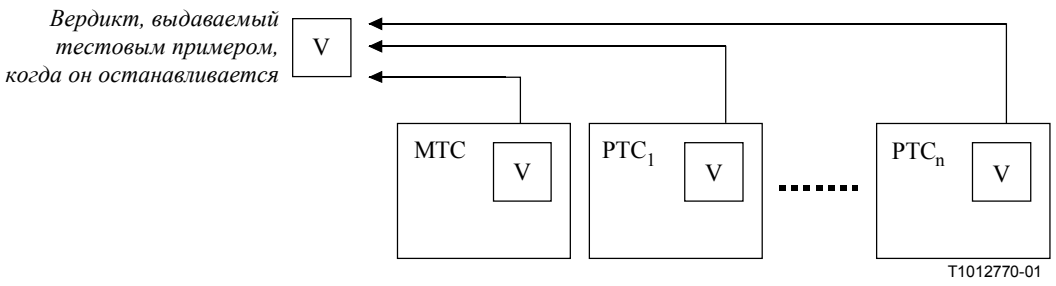


Рисунок 14/Z.140 – Иллюстрация отношений между вердиктами

ПРИМЕЧАНИЕ. – TTCN-3 не определяет реальные механизмы, которые производят обновление местного вердикта и вердикта тестового примера. Эти механизмы зависят от реализации.

25.2 Значения вердиктов и правила перезаписи

25.2.0 Общие положения

Вердикт может иметь пять различных значений: **pass, fail, inconc, none** и **error**, то есть различающихся значений для **verdicttype** (см. п. 6.1).

ПРИМЕЧАНИЕ. – Значение **inconc** означает неокончательный вердикт.

Операция **setverdict** используется только со значениями **pass, fail, inconc** и **none**.

ПРИМЕР 1:

```
setverdict (pass);
getverdict (inconc);
```

Значение местного вердикта может быть вызвано с помощью операции **getverdict**.

ПРИМЕР 2:

```
MyResult := getverdict; // где MyResult – это переменная типа verdicttype
```

Во время реализации тестового компонента создается его объект "местный вердикт", который устанавливается в значение **none**.

Когда значение локального вердикта изменяется (то есть используется операция **setverdict**), результат этого изменения должен подчиняться правилам перезаписи, перечисленным в таблице 22. Вердикт тестового примера обновляется неявно при окончании какого-либо тестового компонента. Результат такой неявной операции также подчиняется правилам перезаписи, перечисленным в таблице 22.

Таблица 22/Z.140 – Правила перезаписи для вердикта

Текущее значение вердикта	Новое присвоенное значение вердикта			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

ПРИМЕР 3:

```
 :
setverdict(pass);     // локальный вердикт устанавливается как pass

 :
 setverdict(fail);     //  до выполнения этой строки, в результате
 :               // чего значение местного вердикта будет
 :               // перезаписано в fail. Когда PTC остановится, вердикт
 :               // тестового примера будет установлен в fail.
```

25.2.1 Вердикт Error

Вердикт **error** является особым, так как он устанавливается тестовой системой для индикации появления ошибки тестового примера (то есть во время выполнения). Он не должен устанавливаться операцией **setverdict** и не будет возвращен операцией **getverdict.** Никакое другое значение вердикта не может заменить вердикт **error**. Это означает, что вердикт **error** может быть только результатом операции тестового примера **execute**.

26 Внешние действия

В некоторых ситуациях тестирования, некоторый интерфейс (ы) SUT могут отсутствовать или являться неизвестными априори (например, интерфейс управления), однако может оказаться необходимо стимулировать

SUT на выполнение определенных действий (например, послать сообщение тестовой системе). Кроме того, определенные действия могут требоваться от испытательного персонала (например, изменить условия окружающей среды тестирования, такие как температура, напряжение подводимой энергии, и т. д.). Необходимое действие может быть описано как строковое выражение; то есть, использование буквенных последовательностей, переменных и параметров строкового типа, и т.д., а также в виде комбинации указанных выше вариантов.

ПРИМЕР:

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString); // Неформальное описание
                                                   // внешних воздействий
```

Нет никакой спецификации того, что необходимо сделать над SUT или самой SUT, чтобы вызвать это действие, только неофициальное описание самого необходимого действия.
Внешние воздействия могут использоваться в тестовых случаях, функциях, altstep и в управлении модулем.

27 Часть управления модулем

27.0 Общие положения

Тестовые примеры определены в части определений модуля, в то время как часть контроля за модулем управляет их выполнением. Все переменные (если есть), определенные в части управления модуля, должны попасть в тестовый пример посредством параметризации, если они должны использоваться в определении поведения того тестового примера; то есть, TTCN-3 не поддерживает глобальные переменные любого вида.

В начале каждого тестового случая должна быть перезагружена тестовая конфигурация. Это означает, что все компоненты и порты, вводимые операциями create, connect и т.д., были разрушены в предыдущем тестовом случае, когда тот тестовый случай был остановлен (следовательно не 'видимы' в новом тестовом случае).

27.1 Выполнение тестовых примеров

Тестовый пример вызывают, используя выражение **execute** . Как результат выполнения тестового примера, будет возвращено и может быть приписано переменной для дальнейшей обработки заключение тестового примера либо **none**, **pass, inconc, fail** либо **error**.
Факультативно, выражение **execute** позволяет контролировать тестовый пример посредством продолжительности работы таймера (см. 27.5).

ПРИМЕР:

```
execute(MyTestCase1());      // выполняет MyTestCase1 без записи выдаваемого
                         // тестового вердикта и без контроля времени

MyVerdict := execute(MyTestCase2()); // выполняет MyTestCase2 и записывает
                             // результирующий вердикт в переменную
                             // MyVerdict

MyVerdict := execute(MyTestCase3(),5E-3);// выполняет MyTestCase3 и записывает
                       // результирующий вердикт в переменную MyVerdict. Если
                       // тестовый пример не закончится в пределах 5 мс, то
                       // MyVerdict даст значение "error"
```

27.2 Окончание тестовых примеров

Тестовый пример заканчивается с окончанием МТС. При окончания МТС (явно или неявно) все работающие параллельные тестовые компоненты должны быть удалены тестовой системой.

ПРИМЕЧАНИЕ 1. – Конкретные механизмы для остановки всех РТС зависят от инструментальных решений и поэтому не рассматриваются в данной Рекомендации.

Окончательный вердикт тестового примера вычисляется на основе окончательных местных вердиктов различных тестовых компонентов согласно правилам, определенным в разделе 25. Реальный местный вердикт тестового компонента становится его окончательным местным вердиктом, когда тестовый компонент остановился сам или когда остановился при помощи другого тестового компонента или их остановила тестовая система.

ПРИМЕЧАНИЕ 2. – Чтобы избежать кольцевых состояний при вычислении тестовых вердиктов из-за задержанных остановок разных PTC, MTC должен убедиться, что все PTC остановлены (посредством команды **done** или **killed**), прежде чем остановиться самому.

27.3 Управление выполнением тестовых примеров

Выражения программы, ограниченные определенными в таблицах 11 и 12, могут использоваться в части управления модуля, чтобы определить такие характеристики, как порядок, в котором тестовые примеры должны быть выполнены или сколько раз должен быть выполнен тестовый пример.

ПРИМЕР:

```
module MyTestSuite  ()  {
:
control  {
    :
    // Выполнить этот тест 10 раз     count:=0;
    while (count < 10)
    {    execute (MySimpleTestCase1());
     count := count+1;
     }
    }
}
```

Если программные команды не используются, то по умолчанию тестовые примеры выполняются последовательно в порядке, в котором они появляются в управляющей части модуля.

ПРИМЕЧАНИЕ. – Это не исключает возможности, что определенные инструменты могут пожелать отменить этот порядок по умолчанию, чтобы позволить пользователю или инструменту выбрать какой-либо другой порядок выполнения.

Для управления выполнением тестовых примеров может также использоваться выбор и отсев (см. 27.4)

27.4 Выбор тестового примера

Существуют различные способы для выбора и отсева тестовых примеров в TTCN-3. Например, для выбора тестовых примеров, которые следует выполнить, и для отмены выбора могут использоваться булевы выражения. Сюда, безусловно, входит и использование функций, которые выдают булево значение.

ПРИМЕР 1:

```
module MyTestSuite () {
     :
    control   {
      :
     if (MySelectionExpression1())       {
          execute(MySimpleTestCase1());
          execute(MySimpleTestCase2());
          execute(MySimpleTestCase3());
     }
     if (MySelectionExpression2())       {
          execute(MySimpleTestCase4());
          execute(MySimpleTestCase5());
          execute(MySimpleTestCase6());
     }
      :
    }
}
```

Другой метод выполнения тестовых примеров в виде группы заключается в сборе их в какой-либо функции и выполнении этой функции из управления модулем.

ПРИМЕР 2:

```
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
```

```
}
:
control
{ if (MySelectionExpression1()) { MyTestCaseGroup1(); }
  if (MySelectionExpression2()) { MyTestCaseGroup2(); }
  :
}
```

Поскольку тестовый пример возвращает единственное значение типа **verdicttype**, возможно также управлять порядком выполнения тестового примера в зависимости от результата тестового примера. Использование **verdicttype** TTCN-3 - другой способ выбора тестового примера.

ПРИМЕР 3:

```
if ( execute (MySimpleTestCase()) == pass )
     { execute (MyGoOnTestCase()) }
else
     { execute (MyErrorRecoveryTestCase()) };
```

27.5 Использование таймеров в управлении

Таймеры могут использоваться для того, чтобы управлять выполнением тестового примера. Это может быть сделано, используя явный таймаут в выражении **execute** . Если тестовый пример не закончится в пределах этой продолжительности, то результатом выполнения тестового примера является вердикт ошибки, и тестовая система должна закончить этот тестовый пример. Таймер, используемый для контроля за тестовым примером, является таймером системы и его не следует объявлять или запускать.

ПРИМЕР 1:

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// Где возвращенное заключение будет error если MyTestCase не завершит выполнение
// в пределах 7 мс
```

Можно также использовать таймерные операции для явного контроля выполнения тестового примера.

ПРИМЕР 2:

```
// Пример использования операции со считающим таймером
while (T1.running or x<10) // Здесь T1 – предыдущий запущенный таймер
{   execute(MyTestCase());
    x := x+1;
}

// Пример использования операций start и timeout

timer T1 := 1.0;
:
execute(MyTestCase1());
T1.start;
T1.timeout; // Пауза перед выполнением следующего тестового примера
execute(MyTestCase2());
```

28 Определение атрибутов

28.0 Общие положения

Атрибуты связываются с элементами языка TTCN-3 посредством команды **with**. Синтаксис для аргумента команды **with** (то есть реальные атрибуты) определяется как нефиксированная текстовая цепочка.

Имеется четыре вида атрибутов:

a) **display:** позволяет определять атрибуты отображения, относящиеся к конкретному формату представления;

b) **encode:** позволяет ссылаться на конкретные правила кодирования;

c) **variant:** позволяет ссылаться на конкретные варианты кодирования;

d) **extension:** позволяет описывать атрибуты, определяемые пользователем.

28.1 Атрибуты отображения

Все элементы языка TTCN-3 могут иметь атрибуты **display**, чтобы указывать, как конкретные элементы языка должны отображаться, например, в табличном формате.

Специальные строки для атрибутов, относящиеся к атрибутам отображения для формата табличного (аттестационного) представления, можно найти в Рекомендации МСЭ-Т Z.141 [1].

Специальные цепочки для атрибутов, относящиеся к атрибутам отображения для формата графического представления, можно найти в Рекомендации МСЭ-Т Z.142 [2].

Другие атрибуты **display** могут определяться пользователем.

> ПРИМЕЧАНИЕ. – Так как определяемые пользователем атрибуты не стандартизованы, эти атрибуты могут отличаться в зависимости от инструментов, либо могут даже не поддерживаться.

28.2 Кодирование значений

28.2.0 Общие положения

Правила кодирования определяют, как конкретное значение, шаблон и т. п. должны быть закодированы и переданы через порт связи, а также то, как принятые сигналы должны быть декодированы. В TTCN-3 нет безусловного (по умолчанию) механизма кодирования. Это означает, что правила кодирования или директивы по кодированию определяются некоторым внешним по отношению к TTCN-3 способом.

В TTCN-3 общие и частные правила кодирования могут быть описаны путем использования атрибутов **encode** и **variant.**

28.2.1 Атрибуты кодирования

Атрибут **encode** позволяет связать некоторое указанное правило или директиву кодирования с определением TTCN-3.

Способ определения реальных правил кодирования (например, проза, функции и т. п.) выходит за рамки настоящей Рекомендации. Если конкретные правила не указаны, то кодирование должно определяться отдельной реализацией.

В большинстве случаев атрибуты кодирования будут использоваться иерархически. Верхним уровнем является модуль в целом, следующим уровнем – группа, а самым нижним уровнем – отдельный тип или определение:

a) **module:** кодирование применяется ко всем типам, определенным в модуле, включая типы TTCN-3 (встроенные типы);

b) **group:** кодирование применяется к группе определений типов, определяемых пользователем;

c) **type or definition:** кодирование применяется к отдельному типу или определению, определяемому пользователем;

d) **field**: кодирование применяется к полю в типе **record** или **set** или шаблону.

ПРИМЕР:

```
module MyTTCNmodule
{   :
import from MySecondModule with {
          encode "MyRule 1"
          }
                 // Экземпляры MyRecord будут кодироваться согласно правилу MyRule 1
  :
type charstring MyType;// Обычно кодируется согласно определенному
                // правилу глобального кодирования
  :
  group MyRecords

  {   :
      type record MyPDU1

      {
       integer    field1,   // field1 будет кодироваться согласно Rule 3
       boolean    field2,   // field2 будет кодироваться согласно Rule 3
```

```
    Mytype     field3    // field3 будет кодироваться согласно Rule 2
    }
    with {encode (field1, field2) "Rule 3"}
    :
}

with {encode "Global encoding rule"
}
     :


}
   with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.2 Атрибуты примера

Чтобы определить обработку указанной в настоящее время схемы кодирования вместо ее замены, должен использоваться атрибут **variant** . Атрибут **variant** отличается от других атрибутов тем, что они близко связаны с атрибутами кодирования. Поэтому, для атрибутов варианта применяются дополнительные правила перезаписи (см. 28.5.1).

ПРИМЕР:

```
module MyTTCNmodule1
{   :
    type charstring MyType; // Обычно кодируется согласно «правилу глобального кодирования»
    :
    group MyRecords
    {    :
     type record MyPDU1
     {
         integer field1,    // field1 будет кодироваться согласно Rule 2
         Mytype  field3     // field3 будет кодироваться согласно Rule 2
                      // использую любой возможный формат кодирования длины
     }
     with { variant (field1) "length form 3" }
     :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.3 Специальные последовательности

Следующие последовательности - предопределенные (стандартизированные) признаки **variant** для простых основных типов (см. E.2.1):

a) "8 битов" и "8 битов без знака", означает, что относился к целому числу и зарегистрированному типу, что целое значение или целое число, связанные с пересчетом, должны быть обработаны, как будто они представлены как 8-битные (один байт) в пределах системы.

b) "16 битов" и "16 битов без знака", означает, что относился к целому числу и зарегистрированному типу, что целое значение или целое число, связанные с пересчетом, должны быть обработаны, как будто они представлены как 16-битные (два байта) в пределах системы.

c) "32 бита" и "32 бита без знака", означает, что относился к целому числу и зарегистрированному типу, что целое значение или целое число, связанные с пересчетом, должны быть обработаны, как будто они представлены как 32 битах (четыре байта) в пределах системы.

d) "64 бита" и "64 бита без знака", означает, что относился к целому числу и зарегистрированному типу, что целое значение или целое число, связанные с пересчетом, должны быть обработаны, как будто они представлены как 64 битах (восемь байтов) в пределах системы.

e) "IEEE754 float","IEEE754 double", "IEEE754 extended float" и "IEEE754 extended double" означает, что относится к типу с плавающей запятой, что величина должна быть закодирована и расшифрована согласно стандартному IEEE 754 (см. библиографию).

Следующие последовательности – предопределенные (стандартизированные) признаки **variant** для **charstring** и **universal charstring** (см. E.2.2):

a) "UTF-8" значит, в применении к типу универсальная строка символов, что каждый символ значения должен быть индивидуально закодирован и расшифрован согласно Формату Преобразования UCS 8 (UTF-8) как определено в Приложении R ISO/IEC 10646 [10].

b) "UCS-2" значит, в применении к типу универсальная строка символов, что каждый символ значения должен быть индивидуально закодирован и расшифрован согласно форме представления кодирования UCS-2 (см. 14.1 из ISO/IEC 10646 [10]).

c) "UTF-16" значит, в применении к типу универсальная строка символов, что каждый символ значения должен быть индивидуально закодирован и расшифрован согласно Формату Преобразования UCS 16 (UTF-16) как определено в Приложении Q ISO/IEC 10646 [10].

d) "на 8 битов" значит, в применении к типам строка символов и универсальная строка символов, что каждый символ значения должен быть индивидуально закодирован и расшифрован согласно закодированному представлению, как это определено в ISO/IEC 8859 (8-битовое кодирование).

Следующая последовательность – предопределенный (стандартизированный) признак **variant** для структурированных типов (см. E.2.3):

a) "IDL:fixed FORMAL/01-12-01 v.2.6" означает, в применении к типу записи, что величина должна быть обработана как IDL десятичное значение с фиксированной запятой (см. библиографию).

Эти атрибуты варианта могут использоваться в комбинации с более общими кодовыми признаками, определенными на более высоком уровне. Например, **universal charstring**, определенный с атрибутом **variant**, приписывает "UTF-8" в пределах модуля, у которого непосредственно есть глобальный признак кодирования "BER:1997" (см. 12.2/Z.146 [6]), заставит каждый символ значения в пределах последовательности сначала быть закодированным по правилам UTF-8, и затем эта величина UTF-8 будет закодирована по более глобальным правилам BER .

28.2.4 Неправильные кодирования

Если желательно указать неправильные кодирования, то они указываются в источнике, который способен обрабатывать ссылки и является внешним по отношению к модулю, таким же способом, каким даются ссылки на действительные правила кодирования.

28.3 Атрибуты расширения

Все элементы языка TTCN-3 могут иметь атрибуты **extension**, определяемые пользователем.

> ПРИМЕЧАНИЕ. – Так как определяемые пользователем атрибуты не стандартизованы, эти атрибуты могут по-разному интерпретироваться в инструментах, предоставленных разными поставщиками, либо даже не поддерживаются.

28.4 Контекст атрибутов

Команда **with** может связывать атрибуты с отдельными элементами языка. Можно также связывать атрибуты с несколькими элементами языка, например, список полей структурированного типа с выражением атрибута, связанного с определением единственного типа, или же путем связывания команды **with** с единицей окружающего контекста или с группой элементов языка.

ПРИМЕР:

```
// MyPDU1 будет отображаться как PDU
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2 будет отображаться как PDU с атрибутом расширения, свойственного приложению, MyRule
type record MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}

// Следующая группа определений …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
```

```
}
with {display "PDU"} // Все типы группы MyPDUs будут отображаться как PDU

// is identical to
group MyPDUs {
    type record MyPDU3 { … } with { display "PDU"}
    type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 Правила перезаписи для атрибутов

Определение признака в единице контекста более низкого уровня перепишет общее определение признака в единицу контекста более высокого уровня. Дополнительные правила перезаписи для атрибутов варианта определены в 28.5.1.

ПРИМЕР 1:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// В следующем , MyRecordA закодирован согласно RuleA , а не согласно RuleB
type record MyRecordB
{
     :
    field MyRecordA
} with { encode "RuleB" }
```

Выражение **with**, которое помещено в области другого выражения **with**, должно отвергнуть наиболее удаленный **with**. Это должно также относиться к использованию выражения **with** с группой . Нужно быть аккуратным, когда схема переписывания используется совместно со ссылкой на единственные определения. Общим правилом является то, что атрибуты должны быть назначены и переписаны согласно порядку их возникновения.

```
// Пример использования схемы перезаписи с выражением with
 group MyPDUs
 {
     type record MyPDU1 { … }
     type record MyPDU2 { … }

     group MySpecialPDUs
     {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
 }
 with {extension "MySpecialRule"} // MyPDU3 и MyPDU4 будут иметь атрибут расширения,
                 // свойственного приложению MySpecialRule
}
with
{
 display "PDU";       // Все типы группы MyPDUs будут отображаться как PDU и
 extension "MyRule"; // (если не перезаписаны) иметь атрибут расширения MyRule
}

// идентичный …
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" }
 }

}
```

Атрибут определения в нижней области может быть переписан в верхнюю использую команду **override**

ПРИМЕР 2:

```
type record MyRecordA
{
    :
} with {encode "RuleA"}

// Ниже показано, что MyRecordA кодируется согласно RuleA, а не RuleB
type record MyRecordB

    field MyRecordA
} with {encode "RuleB"}
```

Директива замены принуждает все содержащиеся типы во всех низких контекстах принудительно перейти к указанному атрибуту.

28.5.1 Дополнительные правила перезаписи для атрибутов варианта

Атрибут **variant** всегда связывается с атрибутом **encode**. Принимая во внимание, что вариант кодирования может измениться, кодирование не должно изменяться без переписывания всех текущих атрибутов варианта. Поэтому, для атрибутов варианта применены следующие правила перезаписи:

атрибут **variant** переписывает текущий атрибут **variant** согласно правилам, определенным в 28.5;

атрибут **encoding**, то, который переписывает текущий атрибут **encoding** согласно правилам, определенным в 28.5, также переписывает соответствующий текущий атрибут **variant**; то есть, если никакой новый атрибут **variant** не предоставлен, то текущий атрибут **variant** перестает действовать;

атрибут **encoding**, который изменяет текущий атрибут **encoding** импортированного языкового элемента согласно правилам, определенным в 28.6, также изменяет соответствующий текущий атрибут **variant**; то есть, если никакой новый атрибут **variant** не предоставлен, то текущий атрибут **variant** перестает действовать.

ПРИМЕР:

```
module MyVariantEncodingModule {
    :
    type charstring MyCharString; // Обычно закодирован согласно "Encoding 1"
    :
    group MyVariantsOne {
     :
     type record MyPDUone
     {
         integer field1, // field1 будет закодирован согласно только "Encoding 2".
                         // "Encoding 2" перезаписывает "Encoding 1" и вариант 'Variant 1'
         Mytype  field3  // field3 будет закодировано согласно "Encoding 1" с
                         // вариантом "Variant 1".
     }
     with { encoding (field1) "Encoding 2" }
     :
    }
    with { variant "Variant 1" }

    group MyVariantsTwo
    {    :
     type record MyPDUtwo
     {
         integer field1, // field1 будет закодировано согласно "Encoding 3"
                         // использую вариант кодирования 'Variant 3'
         Mytype  field3  // field3 будет закодировано согласно "Encoding 3"
                         // использую вариант кодирования "Variant 2"
     }
     with { variant (field1) "Variant 3" }
     :
    }
    with { encode "Encoding 3"; variant 'Variant 2'}

}
with { encode "Encoding 1" }
```

28.6 Изменение атрибутов импортированных элементов языка

Как правило, элемент языка импортируется вместе с его атрибутами. В некоторых случаях может оказаться необходимым изменить эти атрибуты при импортировании элемента языка; например, некоторый тип может отображаться в одном модуле как ASP, а затем он импортируется другим модулем, в котором он должен отображаться как PDU. Для таких случаев разрешается изменять атрибут по команде импорта.

ПРИМЕР:

```
import from MyModule {
type MyType
}
with { display "ASP" }      // MyType будет отображаться как ASP.

import from MyModule {
group MyGroup
}
with {

display "ASP";           // Все типы по умолчанию будут отображаться на PDU.
extension "MyRule"
}
```

Приложение A
Язык BNF и статическая семантика

A.1 Язык BNF для TTCN-3

A.1.0 Общие положения

В данном Приложении определяется синтаксис TTCN-3 с использованием расширенной формы Бэкуса-Наура (впредь называемый просто BNF).

A.1.1 Соглашения для описания синтаксиса

В таблице A.1 определяется метанотация, применяемая при описании грамматики расширенной BNF для TTCN-3.

Таблица A.1/Z.140 – Синтаксическая метанотация

::=	определяется как
abc xyz	за abc следует xyz
|	альтернатива
[abc]	0 или 1 экземпляр abc
{abc}	0 или более экземпляров abc
{abc}+	1 или более экземпляров abc
(...)	текстовая группа
Abc	нетерминальный символ abc
"abc"	терминальный символ abc

A.1.2 Символы окончания команды

Как правило, все конструкции языка TTCN-3 (то есть определения, объявления, команды и операции) заканчиваются точкой с запятой (;). Точка с запятой не обязательны, когда языковая конструкция заканчивается правой фигурной скобкой (}) или когда следующим символом является правая фигурная скобка (}); то есть, языковая конструкция является последней командой в блоке команд, операций и деклараций.

A.1.3 Идентификаторы

Идентификаторы TTCN-3 чувствительны к состоянию регистра и могут содержать только строчные буквы (a–z), прописные буквы (A–Z) и цифры (0–9). Разрешается также использовать символ подчеркивания (_). Идентификатор должен начинаться с буквы (то есть не с цифры и не с символа подчеркивания).

A.1.4 Комментарии

Комментарии, записанные не ограниченным какими-либо правилами текстом, могут появляться в любом месте спецификации TTCN-3.

Блочные комментарии должны открываться парой символов /* и закрываться парой символов */.

ПРИМЕР 1:

```
/* Это блочный комментарий,
распоженный на двух строках */
```

Блочные комментарии не должны быть вложенными.

```
/* Это не является /* законным */ комментарием */
```

Строчные комментарии должны открываться парой символов // и закрываться *<переводом строки>*.

ПРИМЕР 2:

```
// Это - строчный комментарий,
// расположенный на двух строках
```

Строчные комментарии могут следовать за программными командами TTCN-3, но они не должны быть вложены в команду.

ПРИМЕР 3:

```
// Нижеследующее – незаконно
const // Это MyConst integer MyConst : = 1;

// Нижеследующее – законно
const integer MyConst : = 1;  // Это MyConst
```

A.1.5 Терминальные символы TTCN-3

Терминальные символы TTCN-3 и зарезервированные слова перечислены в таблицах A.2 и A.3.

Таблица A.2/Z.140 – Список специальных терминальных символов TTCN-3

Символы начала/конца блока	**{ }**
Символы начала/конца списка	**( )**
Символы альтернативы	**[ ]**
К символу (в диапазоне)	**. .**
Блочные комментарии и строчные комментарии	**/* */ //**
Символ окончания строки/команды	**;**
Символы арифметических операций	**+ / -**
Символ оператора сцепления цепочек	**&**
Символы оператора эквивалентности	**!= == >= <=**
Символы оболочки цепочки	**" '**
Символы подстановки/сопоставления	**? ***
Символ присвоения	**: =**
Присвоение операции связи	**->**
Значения **bitstring, hexstring** и **octetstring**	**B H O**
Показатель в числе с плавающей запятой	**E**

Предопределенные идентификаторы функций, определенные в Таблице 10 и описанные в Приложении C, также должны рассматриваться как зарезервированные слова.

Таблица A.3/Z.140 – Список терминальных символов TTCN-3, являющихся зарезервированными словами

action	**fail**	**noblock**	**select**
activate	**false**	**none**	**self**
address	**float**	**not**	**send**
alive	**for**	**not4b**	**sender**
all	**from**	**nowait**	**set**
alt	**function**	**null**	**setverdict**
altstep			**signature**
and	**getverdict**	**octetstring**	**start**
and4b	**getcall**	**of**	**stop**
any	**getreply**	**omit**	**subset**
anytype	**goto**	**on**	**superset**
	group	**optional**	**system**
bitstring		**or**	
boolean	**hexstring**	**or4b**	**template**
		out	**testcase**

case	**if**	**override**	**timeout**
call	**ifpresent**		**timer**
catch	**import**	**param**	**to**
char	**in**	**pass**	**trigger**
charstring	**inconc**	**pattern**	**true**
check	**infinity**	**port**	**type**
clear	**inout**	**procedure**	
complement	**integer**		**union**
component	**interleave**	**raise**	**universal**
connect		**read**	**unmap**
const	**kill**	**receive**	
control	**killed**	**record**	**value**
create			**valueof**
	label	**rem**	**var**
deactivate	**language**	**repeat**	**variant**
default	**length**	**reply**	**verdicttype**
disconnect	**log**	**return**	
display		**running**	**while**
do	**map**	**runs**	**with**
done	**match**		
	message		**xor**
else	**mixed**		**xor4b**
encode	**mod**		
enumerated	**modifies**		
error	**module**		
except	**modulepar**		
exception	**mtc**		
execute			
extends			
extension			
external			

Терминальные символы TTCN-3, перечисленные в Таблице A.3, не должны использоваться в качестве идентификаторов в модуле TTCN-3. Они должны быть написаны строчными буквами.

A.1.6 Продукты BNF синтаксиса TTCN-3

A.1.6.0 Модуль TTCN-3

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                   "{"
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   "}"
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - LanguageSpec может  быть пропущен, только если модуль, на который
ссылаются, содержит примечание TTCN-3 */
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 Часть определений модуля

A.1.6.1.0 Общие положения

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
```

```
                                TemplateDef |
                                ModuleParDef |
                                FunctionDef |
                                SignatureDef |
                                TestcaseDef |
                                AltstepDef |
                                ImportDef |
                                GroupDef |
                                ExtFunctionDef |
                                ExtConstDef) [WithStatement]
```

A.1.6.1.1 Определения Typedef

```
      12. TypeDef ::= TypeDefKeyword TypeDefBody
      13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
      14. TypeDefKeyword ::= "type"
      15. StructuredTypeDef ::= RecordDef |
                                UnionDef |
                                SetDef |
                                RecordOfDef |
                                SetOfDef |
                                EnumDef |
                                PortDef |
                                ComponentDef
      16. RecordDef ::= RecordKeyword StructDefBody
      17. RecordKeyword ::= "record"
      18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                            "{" [StructFieldDef {"," StructFieldDef}] "}"
      19. StructTypeIdentifier ::= Identifier
      20. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
      21. StructDefFormalPar ::= FormalValuePar
/* СТАТИЧЕСКАЯ СЕМАНТИКА - FormalValuePar должно быть сведено ко входным параметрам */
      22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                             [OptionalKeyword]
      23. NestedTypeDef ::= NestedRecordDef |
                            NestedUnionDef |
                            NestedSetDef |
                            NestedRecordOfDef |
                            NestedSetOfDef |
                            NestedEnumDef
      24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
      25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef} "}"
      26. NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
      27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
      28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
      29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
      30. StructFieldIdentifier ::= Identifier
      31. OptionalKeyword ::= "optional"
      32. UnionDef ::= UnionKeyword UnionDefBody
      33. UnionKeyword ::= "union"
      34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                           "{" UnionFieldDef {"," UnionFieldDef} "}"
      35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
      36. SetDef ::= SetKeyword StructDefBody
      37. SetKeyword ::= "set"
      38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
      39. OfKeyword ::= "of"
      40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier | AddressKeyword)
      [SubTypeSpec]
      41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
      42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                      "{" EnumerationList "}"
      43. EnumKeyword ::= "enumerated"
      44. EnumTypeIdentifier ::= Identifier
      45. EnumerationList ::= Enumeration {"," Enumeration}
      46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
      47. EnumerationIdentifier ::= Identifier
      48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
      49. SubTypeIdentifier ::= Identifier
      50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
/* СТАТИЧЕСКАЯ СЕМАНТИКА - AllowedValues должен иметь тот же самый тип как и поле подтипа */
      51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch ")"
      52. ValueOrRange ::= RangeDef | ConstantExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Производная RangeDef должна  использоваться только с integer, charstring,
universal charstring или основанном на float типами */
/* СТАТИЧЕСКАЯ СЕМАНТИКА - При порождении подтипа charstring или universal charstring диапазон и
величины не должны быть смешаны в одном и том же SubTypeSpec */
      53. RangeDef ::= LowerBound ".." UpperBound
      54. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
```

```
/* СТАТИЧЕСКАЯ СЕМАНТИКА - StringLength должен  использоваться только с типами String или в рамках
set of и record of. SingleConstExpression и UpperBound должны выражаться  в неотрицательных целых
значениях  (в случае UpperBound - включая бесконечность) */
      55.   LengthKeyword ::= "length"
      56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
      57. PortDef ::= PortKeyword PortDefBody
      58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
      59. PortKeyword ::= "port"
      60. PortTypeIdentifier ::= Identifier
      61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
      62. MessageAttribs ::= MessageKeyword
                             "{" {MessageList [SemiColon]}+ "}"
      63. MessageList ::= Direction AllOrTypeList
      64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
      65. MessageKeyword ::= "message"
      66. AllOrTypeList ::= AllKeyword | TypeList
/* ПРИМЕЧАНИЕ. – Не рекомендуется  использовать AllKeyword в определениях порта  */
      67. AllKeyword ::= "all"
      68. TypeList ::= Type {"," Type}
      69. ProcedureAttribs ::= ProcedureKeyword
                               "{" {ProcedureList  [SemiColon]}+ "}"
      70. ProcedureKeyword ::= "procedure"
      71. ProcedureList ::= Direction AllOrSignatureList
      72. AllOrSignatureList ::= AllKeyword | SignatureList
      73. SignatureList ::= Signature {"," Signature}
      74. MixedAttribs ::= MixedKeyword
                           "{" {MixedList [SemiColon]}+ "}"
      75. MixedKeyword ::= "mixed"
      76. MixedList ::= Direction ProcOrTypeList
      77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
      78. ProcOrType ::= Signature | Type
      79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                           [ExtendsKeyword ComponentType {"," ComponentType}]
                           "{" [ComponentDefList] "}"
      80. ComponentKeyword ::= "component"
      81. ExtendsKeyword ::= "extends"
      82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
      83. ComponentTypeIdentifier ::= Identifier
      84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
      85. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
      86. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
      87. PortElement ::= PortIdentifier [ArrayDef]
      88. PortIdentifier ::= Identifier
```

A.1.6.1.2 Определения констант

```
      89. ConstDef ::= ConstKeyword Type ConstList
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Тип должен соответствовать правилам, данными в пункте 9.*/
      90. ConstList ::= SingleConstDef  {"," SingleConstDef}
      91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Величина ConstantExpression должна иметь тот же самый тип, как и тип,
установленный  для констант */
      92. ConstKeyword ::= "const"
      93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Определения шаблона

```
      94. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef] AssignmentChar TemplateBody
      95. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
      96. TemplateKeyword ::= "template"
      97. TemplateIdentifier ::= Identifier
      98. DerivedDef ::= ModifiesKeyword TemplateRef
      99. ModifiesKeyword ::= "modifies"
      100. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
      101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/* STATIC SEMANTICS - FormalValuePar должно быть сведено ко входным параметрам */
      102. TemplateBody ::= (SimpleSpec | FieldSpecList | ArrayValueOrAttrib) |
      [ExtraMatchingAttributes]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - В пределах TeplateBody ArrayValueOrAttrib может использоваться для
множества, record, record of и set of типов. */
      103. SimpleSpec ::= SingleValueOrAttrib
      104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
      105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
      106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА – В пределах FieldReference ArrayOrBitRef может использоватья для record of
и set of шаблонов/полей шаблонов только в модифицированных шаблонах */
      107. StructFieldRef ::= StructFieldIdentifier| PredefinedType | TypeReference
/* СТАТИЧЕСКАЯ СЕМАНТИКА - PredefinedType и TypeReference дожны использоваться только для некого
обозначения значения. PredefinedType не должно быть AnyTypeKeyword.*/
```

```
      108. ParRef ::= SignatureParIdentifier
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SignatureParIdentifier должен быть формальным Идентификатором параметра из
связанного определения сигнатуры */
      109. SignatureParIdentifier ::= ValueParIdentifier
      110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ArrayRef может факультативно использоваться для типов массивов и SEQUENCE
OF и SET OF из ASN.1 и record of и set of из TTCN-3. Такое же обозначение используется для битовых
ссылок в типе bitstring внутри ASN.1 или TTCN-3 */
      111. FieldOrBitNumber ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SingleExpression превратится в значение целого типа */
      112. SingleValueOrAttrib ::= MatchingSymbol |
                                   SingleExpression |
                                   TemplateRefWithParList
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Для того, чтобы сослаться на переменные в текущей области,
VariableIdentifier (доступный посредством singleExpression) может использоваться только  в in-line
определениях */
      113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
      114. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
      115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch | TemplateBody
      116. NotUsedSymbol ::= Dash
      117. MatchingSymbol ::= Complement |
                              AnyValue |
                              AnyOrOmit |
                              ValueOrAttribList |
                              Range |
                              BitStringMatch |
                              HexStringMatch |
                              OctetStringMatch |
                              CharStringMatch |
                              SubsetMatch |
                              SupersetMatch
      118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch  IfPresentMatch
      119. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
      120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
      121. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
      122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
      123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
      124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
      125. CharStringMatch ::= PatternKeyword Cstring
      126. PatternKeyword ::= "pattern"
      127. Complement ::= ComplementKeyword ValueList
      128. ComplementKeyword ::= "complement"
      129. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
      130. SubsetMatch ::= SubsetKeyword ValueList
/* СТАТИЧЕСКАЯ СЕМАНТИКА  - Соответствующее подмножество , должно использоваться с типом только set
of */
      131. SubsetKeyword ::= "subset"
      132. SupersetMatch ::= SupersetKeyword ValueList
/* СТАТИЧЕСКАЯ СЕМАНТИКА  - Соответствующий расширенный набор должен использоваться только с типом
set of */
      133. SupersetKeyword ::= "superset"
      134. PermutationMatch ::= PermutationKeyword PermutationList
      135. PermutationKeyword ::= "permutation"
      136. PermutationList ::= "(" TemplateBody { "," TemplateBody } ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Ограничения на содержание TemplateBody даны в пункте B.1.3.3 */
      137. AnyValue ::= "?"
      138. AnyOrOmit ::= "*"
      139. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
      140. LengthMatch ::= StringLength
      141. IfPresentMatch ::= IfPresentKeyword
      142. IfPresentKeyword ::= "ifpresent"
      143. Range ::= "(" LowerBound ".." UpperBound ")"
      144. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
      145. UpperBound ::= SingleConstExpression | InfinityKeyword
/* СТАТИЧЕСКАЯ СЕМАНТИКА - LowerBound и UpperBound должны определяться как тип  integer, charstring,
universal charstring или float. В случае, если LowerBound или UpperBound определяются типом
charstring или universal charstring, может присутствовать только SingleConstExpression, и длина
строки должна быть равна 1*/
      146. InfinityKeyword ::= "infinity"
      147. TemplateInstance ::= InLineTemplate
      148. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList]
      |
                                   TemplateParIdentifier
      149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
      150. InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                              TemplateBody
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Область типа может  быть опущена, только когда тип  неявно однозначен */
      151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
      152. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
```

```
      153. TemplateActualPar ::= TemplateInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Когда соответствующий формальный параметр не будет иметь типа шаблона,
порождённое TemplateInstance должно разрешиться в один или более SingleExpressions */
      154. TemplateOps ::= MatchOp | ValueofOp
      155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Тип величины, возвращенной  выражением, должен быть тем же, что и тип
шаблона, а каждое поле шаблона должно разрешиться к единственной величине */
      156. MatchKeyword ::= "match"
      157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
      158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Определения функции

```
      159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                           "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                           StatementBlock
      160. FunctionKeyword ::= "function"
      161. FunctionIdentifier ::= Identifier
      162. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
      163. FunctionFormalPar ::= FormalValuePar |
                                 FormalTimerPar |
                                 FormalTemplatePar |
                                 FormalPortPar
      164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Использование ключевого слова шаблона должно соответствовать ограничениям
пункта 16.1.0 */
      165. ReturnKeyword ::= "return"
      166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
      167. RunsKeyword ::= "runs"
      168. OnKeyword ::= "on"
      169. MTCKeyword ::= "mtc"
      170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
      171. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
      172. FunctionStatementOrDef ::= FunctionLocalDef |
                                     FunctionLocalInst |
                                     FunctionStatement
      173. FunctionLocalInst ::= VarInstance | TimerInstance
      174. FunctionLocalDef ::= ConstDef | TemplateDef
      175. FunctionStatement ::= ConfigurationStatements |
                                 TimerStatements |
                                 CommunicationStatements |
                                 BasicStatements |
                                 BehaviourStatements |
                                 VerdictStatements |
                                 SUTStatements
      176. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
      177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
                           PreDefFunctionIdentifier
      178. PreDefFunctionIdentifier ::= Identifier
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Идентификатор будет одним из предопределенных идентификаторов функции
TTCN-3 из Приложения C */
      179. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
      180. FunctionActualPar ::= TimerRef |
                                 TemplateInstance |
                                 Port |
                                 ComponentRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если соответствующий формальный параметр не будет иметь тип шаблона,
продукция TemplateInstance должно разрешиться к одному или более SingleExpressions, то есть
эквивалентно продукции Выражения */
```

A.1.6.1.5 Определения сигнатур

```
      181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                            "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                            [ExceptionSpec]
      182. SignatureKeyword ::= "signature"
      183. SignatureIdentifier ::= Identifier
      184. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
      185. SignatureFormalPar ::=  FormalValuePar
      186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
      187. ExceptionKeyword ::= "exception"
      188. ExceptionTypeList ::= Type {"," Type}
      189. NoBlockKeyword ::= "noblock"
      190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Определения тестового примера

```
      191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                           "("[TestcaseFormalParList] ")" ConfigSpec
```

```
                        StatementBlock
      192. TestcaseKeyword ::= "testcase"
      193. TestcaseIdentifier ::= Identifier
      194. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
      195. TestcaseFormalPar ::= FormalValuePar |
                                 FormalTemplatePar
      196. ConfigSpec ::= RunsOnSpec [SystemSpec]
      197. SystemSpec ::= SystemKeyword ComponentType
      198. SystemKeyword ::= "system"
      199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")"
                               ["," TimerValue] ")"
      200. ExecuteKeyword ::= "execute"
      201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
      202. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
      203. TestcaseActualPar ::= TemplateInstance
      /* СТАТИЧЕСКАЯ СЕМАНТИКА - Если соответствующий формальный параметр не будет иметь типа
      шаблона, продукция TemplateInstance должна разрешиться к одному или более SingleExpressions, то
      есть эквивалентна продукции Выражения */
```

A.1.6.1.7 Определения Altstep

```
      204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                          "("[AltstepFormalParList] ")" [RunsOnSpec]
                          "{" AltstepLocalDefList AltGuardList "}"
      205. AltstepKeyword ::= "altstep"
      206. AltstepIdentifier ::= Identifier
      207. AltstepFormalParList ::= FunctionFormalParList
/* СТАТИЧЕСКАЯ СЕМАНТИКА - altstep,  активизированые по умолчанию, должны  содержаться в параметрах,
параметрах порта, или параметрах таймера */
/* СТАТИЧЕСКАЯ СЕМАНТИКА - altstep, вызваные только как альтернатива в выражении alt или как
автономное выражение в описании поведения TTCN-3, могут иметь параметры in, out и inout */
      208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
      209. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef | TemplateDef
/*СТАТИЧЕСКАЯ СЕМАНТИКА - AltstepLocalDef должен соответствовать ограничениям, указанным в пункте
16.2.2.1 */
      210. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - все экземпляры таймера в FunctionActualParList должны быть объявлены как
компоненты локальных таймеров (см. также производство ComponentElementDef)*/
      211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Определения импорта

```
      212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
      213. ImportKeyword ::= "import"
      214. AllWithExcepts ::= AllKeyword [ExceptsDef]
      215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
      216. ExceptKeyword ::= "except"
      217. ExceptSpec ::= {ExceptElement [SemiColon]}
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Любой из  порожденных компонентов (ExceptGroupSpec, ExceptTypeDefSpec и
т.д.) может присутствовать только однажды в производстве ExceptSpec */
      218. ExceptElement ::= ExceptGroupSpec |
                             ExceptTypeDefSpec |
                             ExceptTemplateSpec |
                             ExceptConstSpec |
                             ExceptTestcaseSpec |
                             ExceptAltstepSpec |
                             ExceptFunctionSpec |
                             ExceptSignatureSpec |
                             ExceptModuleParSpec
      219. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
      220. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
      221. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
      222. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
      223. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
      224. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
      225. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
      226. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
      227. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
      228. ImportSpec ::= {ImportElement [SemiColon]}
      229. ImportElement ::= ImportGroupSpec |
                             ImportTypeDefSpec |
                             ImportTemplateSpec |
                             ImportConstSpec |
                             ImportTestcaseSpec |
                             ImportAltstepSpec |
                             ImportFunctionSpec |
                             ImportSignatureSpec |
                             ImportModuleParSpec
      230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
/* ПРИМЕЧАНИЕ. - не рекомендуется использовать RecursiveKeyword  */
```

```
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {","
FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                          EnumTypeIdentifier |
                          PortTypeIdentifier |
                          ComponentTypeIdentifier |
                          SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
247. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword (SignatureRefList | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword (ModuleParRefList | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 Определения группы

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 Определения внешней функции

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")" [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 Определения внешней константы

```
271. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Тип должен соответствовать правилам, приведенным в пункте 9.*/
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Определение параметров модуля

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar | ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Величина ConstantExpression должна иметь тот же самый тип, как и тип,
установленный для Параметра */
277. ModuleParType ::= Type
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Тип не должен быть компонентным типом, типом по умолчанию или anytype. Тип
может превратиться в адресный тип только в том случае, если определение для типа адреса  определено в
пределах модуля */
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar ConstantExpression]
                       {","ModuleParIdentifier [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Часть управления

A.1.6.2.0 Общие положения

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Ограничения на использование выражений в части управления даны в Таблице
11 */
```

A.1.6.2.1 Создание экземпляра переменной

```
286. VarInstance ::= VarKeyword ((Type VarList) | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 Создание экземпляра таймера

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Если Выражение приводит к SingleExpression, оно должно привести к величине
типа float. Выражение должно только привести к CompoundExpression при инициализации  назначения
величины таймера по умолчанию для массива таймеров */
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier) {ArrayOrBitRef}
```

A.1.6.2.3 Операции компонента

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword ["(" SingleExpression ")"] [AliveKeyword]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Ограничения на SingleExpression смотреть в пункте 22.1 */
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
```

```
      323. ComponentRef ::= ComponentOrDefaultReference | SystemOp | SelfOp | MTCOp
      324. DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
      325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                           AllConnectionsSpec |
                                           AllPortsSpec |
                                           AllCompsAllPortsSpec]
      326. AllConnectionsSpec ::= "(" PortRef ")"
      327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
      328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword PortKeyword ")"
      329. DisconnectKeyword ::= "disconnect"
      330. MapStatement ::= MapKeyword SingleConnectionSpec
      331. MapKeyword ::= "map"
      332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
      333. UnmapKeyword ::= "unmap"
      334. StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword "(" FunctionInstance ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - экземпляр Функции может  иметь только входные параметрах */
/* СТАТИЧЕСКАЯ СЕМАНТИКА - экземпляр Функции не должно иметь параметров таймера */
      335. StartKeyword ::= "start"
      336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral Dot StopKeyword) |
                               (AllKeyword ComponentKeyword Dot StopKeyword)
      337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference | MTCOp | SelfOp
      338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral Dot KillKeyword) |
                               (AllKeyword ComponentKeyword Dot KillKeyword)
      339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Переменная, связанная с VariableRef или типом возвращения, связанным с
FunctionInstance, должна иметь компонентный тип, если используется в выражениях конфигурации, и
должен иметь тип по умолчанию, если используется в деактивированном выражении. */
      340. KillKeyword ::= "kill"
```

A.1.6.2.4 Операции порта

```
      341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
      342. CommunicationStatements ::= SendStatement |
                                       CallStatement |
                                       ReplyStatement |
                                       RaiseStatement |
                                       ReceiveStatement |
                                       TriggerStatement |
                                       GetCallStatement |
                                       GetReplyStatement |
                                       CatchStatement |
                                       CheckStatement |
                                       ClearStatement |
                                       StartStatement |
                                       StopStatement
      343. SendStatement ::= Port Dot PortSendOp
      344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
      345. SendOpKeyword ::= "send"
      346. SendParameter ::= TemplateInstance
      347. ToClause ::= ToKeyword AddressRef |
                        AddressRefList |
                        AllKeyword ComponentKeyword
/* СТАТИЧЕСКАЯ СЕМАНТИКА - AddressRef не должен содержать  механизм соответствия */
      348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
      349. ToKeyword ::= "to"
      350. AddressRef ::= TemplateInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - TemplateInstance должен иметь адресный или компонентный тип */
      351. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
      352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
      353. CallOpKeyword ::= "call"
      354. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - только выходные параметры могут быть опущены или определены с
соответствующим атрибутом */
      355. CallTimerValue ::= TimerValue | NowaitKeyword
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Значение должно иметь  тип float */
      356. NowaitKeyword ::= "nowait"
      357. PortCallBody ::= "{" CallBodyStatementList "}"
      358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
      359. CallBodyStatement ::= CallBodyGuard  StatementBlock
      360. CallBodyGuard ::= AltGuardChar CallBodyOps
      361. CallBodyOps ::= GetReplyStatement | CatchStatement
      362. ReplyStatement ::= Port Dot PortReplyOp
      363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
      364. ReplyKeyword ::= "reply"
      365. ReplyValue ::= ValueKeyword Expression
      366. RaiseStatement ::= Port Dot PortRaiseOp
      367. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
      368. RaiseKeyword ::= "raise"
      369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
      370. PortOrAny ::= Port | AnyKeyword PortKeyword
```

```
      371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - вариант PortRedirect может  присутствовать только если также присутствует
вариант ReceiveParameter */
      372. ReceiveOpKeyword ::= "receive"
      373. ReceiveParameter ::= TemplateInstance
      374. FromClause ::= FromKeyword AddressRef
      375. FromKeyword ::= "from"
      376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
      377. PortRedirectSymbol ::= "->"
      378. ValueSpec ::= ValueKeyword VariableRef
      379. ValueKeyword ::= "value"
      380. SenderSpec ::= SenderKeyword VariableRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Variable ref должен иметь тип компонент или адрес */
      381. SenderKeyword ::= "sender"
      382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
      383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - вариант PortRedirect может  присутствовать, только если также присутствует
вариант ReceiveParameter */
      384. TriggerOpKeyword ::= "trigger"
      385. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
      386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                             [PortRedirectWithParam]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - вариант PortRedirect может  присутствовать, только если также присутствует
вариант ReceiveParameter */
      387. GetCallOpKeyword ::= "getcall"
      388. PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
      389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                                     SenderSpec
      390. ParamSpec ::= ParamKeyword ParamAssignmentList
      391. ParamKeyword ::= "param"
      392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
      393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
      394. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* СТАТИЧЕСКАЯ СЕМАНТИКА - parameterIdentifiers должен быть из соответствующего определения сигнатуры
*/
      395. ParameterIdentifier ::= ValueParIdentifier
      396. VariableList ::= VariableEntry  {"," VariableEntry}
      397. VariableEntry ::= VariableRef | NotUsedSymbol
      398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
      399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
                              [FromClause] [PortRedirectWithValueAndParam]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - вариант PortRedirectWithParam может  присутствовать только если также
присутствует вариант ReceiveParameter */
      400. PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
      401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                             RedirectWithParamSpec
      402. GetReplyOpKeyword ::= "getreply"
      403. ValueMatchSpec ::= ValueKeyword TemplateInstance
      404. CheckStatement ::= PortOrAny Dot PortCheckOp
      405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
      406. CheckOpKeyword ::= "check"
      407. CheckParameter ::= CheckPortOpsPresent | FromClausePresent | RedirectPresent
      408. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
      409. RedirectPresent ::= PortRedirectSymbol SenderSpec
      410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
      411. CatchStatement ::= PortOrAny Dot PortCatchOp
      412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - вариант PortRedirect может  присутствовать, только если вариант
CatchOpParameter также присутствует */
      413. CatchOpKeyword ::= "catch"
      414. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
      415. ClearStatement ::= PortOrAll Dot PortClearOp
      416. PortOrAll ::= Port | AllKeyword PortKeyword
      417. PortClearOp ::= ClearOpKeyword
      418. ClearOpKeyword ::= "clear"
      419. StartStatement ::= PortOrAll Dot PortStartOp
      420. PortStartOp ::= StartKeyword
      421. StopStatement ::= PortOrAll Dot PortStopOp
      422. PortStopOp ::= StopKeyword
      423. StopKeyword ::= "stop"
      424. AnyKeyword ::= "any"
```

A.1.6.2.5 Операции таймера

```
      425. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
      426. TimerOps ::= ReadTimerOp | RunningTimerOp
      427. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
      428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
      429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
      430. ReadTimerOp ::= TimerRef Dot ReadKeyword
```

```
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Тип

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                       EnumTypeIdentifier |
                       SubTypeIdentifier |
                       ComponentTypeIdentifier
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ArrayBounds приведет  к неотрицательной величине целого типа */
```

A.1.6.4 Значение

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
                         OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier |
                        ModuleParIdentifier ) |
```

```
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" "B"
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" "H"
486. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* ССЫЛКА - символ определен соответствующим типом CharacterString. Для
charstring символ из набора символов определен в ITU-T Rec. T.50. Для универсального charstring
символ из любого набора символов определен в ISO/IEC 10646 */
491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
493. AlphaNum ::= Alpha | Num
494. UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
| "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
495. LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
| "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
496. ExtendedAlphaNum ::= /* ССЫЛКА - графический символ из BASIC LATIN или из LATIN-1
SUPPLEMENT наборов символов  определен в ISO/IEC 10646 (символы от знака (0,0,0,32), до знака
(0 0 0 126), от знака (0 0 0 161), до знака (0 0 0 172) и от знака (0 0 0 174), до знака (0 0 0
255))  */
497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 Параметризация

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type
ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
509. TimerParIdentifier ::= Identifier
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           TemplateKeyword Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 Выражение With

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА - DefOrFieldRef должен обращаться к определению или области, которые
находится в пределах модуля, группы или определения, с которым связано выражение with  */
526. DefinitionRef ::= StructTypeIdentifier |
                       EnumTypeIdentifier |
                       PortTypeIdentifier |
                       ComponentTypeIdentifier |
                       SubTypeIdentifier |
                       ConstIdentifier |
                       TemplateIdentifier |
                       AltstepIdentifier |
                       TestcaseIdentifier |
                       FunctionIdentifier |
                       SignatureIdentifier |
```

```
                        VarIdentifier |
                        TimerIdentifier |
                        PortIdentifier |
                        ModuleParIdentifier |
                        FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{" GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{" TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{" TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{" ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{" AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{" TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{" FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{" SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{" ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 Выражение Behaviour

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
/* СТАТИЧЕСКАЯ СЕМАНТИКА - TestcaseInstance нельзя вызвать из пределов существующего выполняемого
тестового примера выполнения или цепи функции, вызванной из тестового примера, то есть экземпляры
тестовых примеров могут  создаваться только из части управления или из функций, непосредственно
вызванных из части управления  */
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ActivateOp нельзя вызвать из пределов части управления модуля */
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SingleExpression должен привести к величине типа verdict */
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Чтобы присвоить значение error  не должен использоваться SetLocalVerdict
*/
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ] {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
537. ActionText ::= FreeText | Expression
/*СТАТИЧЕСКАЯ СЕМАНТИКА – Выражение должно иметь базовый тип charstring или universal charstring */
538. ReturnStatement ::= ReturnKeyword  [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] | GuardOp
     StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
/*СТАТИЧЕСКАЯ СЕМАНТИКА - BooleanExpression должно соответствовать ограничениям пункта 20.1.2 */
545. GuardOp ::= TimeoutStatement |
                 ReceiveStatement |
                 TriggerStatement |
                 GetCallStatement |
                 CatchStatement |
                 CheckStatement |
                 GetReplyStatement |
                 DoneStatement |
                 KilledStatement
/* СТАТИЧЕСКАЯ СЕМАНТИКА - GuardOp, используемый в пределах части управления модуля, должен
содержать только timeoutStatement  */
546. InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
549. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
/* СТАТИЧЕСКАЯ СЕМАНТИКА - StatementBlock, не может содержать выражения loop, goto, activate,
deactivate, stop, return или вызовы функций  */
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
```

```
      558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
      559. ActivateKeyword ::= "activate"
      560. DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference ")"]
      561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Основные выражения

```
      562. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct |
                               SelectCaseConstruct
      563. Expression ::= SingleExpression | CompoundExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Выражение не должно содержать в пределах части управления модуля
Конфигурацию,  операцию активации или операцию verdict */
      564. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Внутри CompoundExpression ArrayExpression могут быть использованы типы
Arrays, record, record of и set of. */
      565. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
      566. FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
      567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
      568. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
      569. NotUsedOrExpression ::= Expression | NotUsedSymbol
      570. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
      571. SingleConstExpression ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SingleConstExpression не должен содержать Переменные или параметры Модуля
и должен привести к постоянной Величине во  время компиляции */
      572. BooleanExpression ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - BooleanExpression должен привести к Величине Булевого типа */
      573. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - В пределах CompoundConstExpression ArrayConstExpression может
использоваться для  типов Arrays, record, record of и set of.  */
      574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec}
      "}"
      575. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
      576. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
      577. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
      578. Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Выражение, стоящее в правой части операции присваивания, нужно оценить
явной Величиной типа, совместимого с типом, стоящим в левой части, для переменных величин, и должно
оцениваться явной Величиной, шаблоном (буквенный или экземпляр шаблона) или механизмом сопоставления,
совместимым с типом, стоящим в левой части, для переменных шаблона. */
      579. SingleExpression ::= XorExpression { "or" XorExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один XorExpression будет существовать, то XorExpressions
должны быть выражены в определенных величинах совместимых типов */
      580. XorExpression ::= AndExpression { "xor" AndExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один AndExpression будет существовать, то AndExpressions
должны быть выражены в определенных величинах совместимых типов */
      581. AndExpression ::= NotExpression { "and" NotExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один NotExpression будет существовать, то NotExpressions
должны быть выражены в определенных величинах совместимых типов */
      582. NotExpression ::= [ "not" ] EqualExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Операнды  оператора not должны иметь булевый тип (TTCN или ASN.1) или
производные Булевого типа. */
      583. EqualExpression ::= RelExpression { EqualOp RelExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один RelExpression будет существовать, то RelExpressions
должны быть выражены в определенных величинах совместимых типов */
      584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если оба  ShiftExpressions существуют, то каждое ShiftExpression должно
быть выражено определенным целым числом, значением Enumerated или float (эти значения могут
относиться к TTCN или являться значениями ASN.1), или являться производными этих типов  */
      585. ShiftExpression ::= BitOrExpression { ShiftOp BitOrExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Каждый Результат должен разрешиться к определенной Величине. Если больше
чем один Результат будет существовать, то правый операнд должен быть типа целого числа  или
производного и если изменение op будет '<<' или '>>' , то левый операнд должен привести или к
bitstring, hexstring или к типу octetstring или производным этих типов. Если изменение op будет '<'
или '>' тогда, то левый операнд должен иметь тип bitstring, hexstring, charstring или universal
charstring или производные этих типов  */
      586. BitOrExpression ::= BitXorExpression { "or4b" BitXorExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один BitXorExpression будет существовать, то
BitXorExpressions должны быть выражены в определенных величинах совместимых типов */
      587. BitXorExpression ::= BitAndExpression { "xor4b" BitAndExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один BitAndExpression будет существовать, то
BitAndExpressions должны быть выражены в определенных величинах совместимых типов */
      588. BitAndExpression ::= BitNotExpression { "and4b" BitNotExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если больше чем один BitNotExpression будет существовать, то
BitNotExpressions должны быть выражены в определенных величинах совместимых типов */
      589. BitNotExpression ::= [ "not4b" ] AddExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Если оператор not4b  существует, то операнд должен иметь тип bitstring,
octetstring или hexstring или производные этих типов. */
      590. AddExpression ::= MulExpression { AddOp MulExpression }
```

```
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Каждый MulExpression должен привести к определенной Величине. Если
существует больше чем один MulExpression, и AddOp приводит к StringOp тогда, то MulExpressions должен
привести к тому же самому типу, который должен быть bitstring, hexstring, octetstring, charstring или
universal charstring или производные этих типов. Если существует больше чем один MulExpression, и
AddOp не приводит к StringOp, то MulExpression должен привести к integer или float типу  или к
производным этих типов.*/
      591. MulExpression ::= UnaryExpression { MultiplyOp UnaryExpression }
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Каждый UnaryExpression должен привести к определенной Величине. Если
больше чем один UnaryExpression будет существовать тогда, то UnaryExpressions  должен привести к
integer или float типу  или к производным этих типов. */
      592. UnaryExpression ::= [ UnaryOp ] Primary
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Primary приведется к определенной Величине к integer или float типу  или к
производным этих типов .*/
      593. Primary ::= OpCall | Value | "(" SingleExpression ")"
      594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | TypeDefIdentifier))
                                      | ArrayOrBitRef }+
/* СТАТИЧЕСКАЯ СЕМАНТИКА  - TypeDefIdentifier должен использоваться, только если тип VarInstance или
ReferencedValue, в котором используется ExtendedFieldReference, относится к типу anytype.*/
      595. OpCall ::= ConfigurationOps |
                      VerdictOps |
                      TimerOps |
                      TestcaseInstance |
                      FunctionInstance |
                      TemplateOps |
                      ActivateOp
      596. AddOp ::= "+" | "-" | StringOp
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Операнды операторов "+" или "-" должны иметь тип integer или float или
являться производными от integer или float (то есть подтип).*/
      597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Операнды операторов "*", "/", rem или mod должны иметь тип integer или
float или являться производными от integer или float (то есть подтип).*/
      598. UnaryOp ::= "+" | "-"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Операнды операторов "+" или "-" должны иметь тип integer или float или
являться производными от integer или float (то есть подтип).*/
      599. RelOp ::= "<" | ">" | ">=" | "<="
/* СТАТИЧЕСКАЯ СЕМАНТИКА - старшинство операторов определено в Таблице 7 */
      600. EqualOp ::= "==" | "!="
      601. StringOp ::= "&"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Операнды оператора string  должны относиться к bitstring, hexstring,
octetstring или являться символьной строкой */
      602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
      603. LogStatement ::= LogKeyword "(" LogItem { "," LogItem } ")"
      604. LogKeyword ::= "log"
      605. LogItem ::= FreeText | TemplateInstance
      606. LoopConstruct ::= ForStatement |
                            WhileStatement |
                            DoWhileStatement
      607. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                            StatementBlock
      608. ForKeyword ::= "for"
      609. Initial ::= VarInstance | Assignment
      610. Final ::= BooleanExpression
      611. Step ::= Assignment
      612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                              StatementBlock
      613. WhileKeyword ::= "while"
      614. DoWhileStatement ::= DoKeyword StatementBlock
                                WhileKeyword "(" BooleanExpression ")"
      615. DoKeyword ::= "do"
      616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                                    StatementBlock
                                    {ElseIfClause}[ElseClause]
      617. IfKeyword ::= "if"
      618. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
      619. ElseKeyword ::= "else"
      620. ElseClause ::= ElseKeyword  StatementBlock
      621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
      622. SelectKeyword ::= "select"
      623. SelectCaseBody ::= "{" { SelectCase }+ "}"
      624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {"," TemplateInstance } ')' |
      ElseKeyword )
                        StatementBlock
      625. CaseKeyword ::= "case"
```

A.1.6.9 Прочие продукты

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

Приложение B

Сопоставление входящих значений

B.1 Механизмы сопоставления шаблона

B.1.1 Общие положения

В данном Приложении определяются механизмы сопоставления, которые могут использоваться в шаблонах TTCN-3 (и только в шаблонах).

B.1.1 Специфичные значения сопоставления

Специфичные значения являются основным механизмом сопоставления в шаблонах TTCN-3. Специфичными значениями в шаблонах являются выражения, которые не содержат механизмов сопоставления или символов подстановки. Если не указано другое, поле шаблона соответствует значению входящего поля тогда и только тогда, когда входящее поле имеет точно такое же значение, как значение, к которому сводится выражение в шаблоне.

ПРИМЕР:

```
// Дано определение типа сообщения

type record MyMessageType
{
   integer fileld1,
   charstring  field2,
   boolean fiel3 optional,
   integer [4]  field4
   }
// Шаблоном сообщения, использующим специфичные значения, может быть
template MyMessageType MyTemplate : =
{
   field1 : = 3 + 2,    // специфическое значение типа integer
   field2 : = "My string",  // специфическое значение типа charstring
   field3 : = true, // специфическое значение типа boolean
   field4 : = {1, 2, 3} // специфическое значение массива integer
}
```

B.1.1.1 Пропуск значений

Ключевое слово omit означает, что отсутствует дополнительная область шаблона. Это может использоваться на величинах всех типов, при условии, что область шаблона является факультативной.

ПРИМЕР:

```
template Mymessage MyTemplate:=
{   :
    :
    field3 := omit,   // исключает это поле
    :
}
```

B.1.2 Механизмы сопоставления вместо значений

B.1.2.0 Общие положения

Могут использоваться следующие механизмы соответствия вместо явных величин.

B.1.2.1 Список значений

Списки значений определяют перечни приемлемых входящих значений. Могут использоваться значения всех типов. Поле шаблона, в котором используется список значений, соответствует входящему полю тогда и только тогда, когда значение входящего поля соответствует какому-либо значению из списка значений. Каждое значение в списке значений должно иметь тип, объявленный для поля шаблона, в котором используется этот механизм.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{
   field1 : = (2, 4, 6),    // список значений integer
   field2 : = ("String1", "String2"),     // список значений charstring
   :
   :
}
```

B.1.2.2 Дополнительный список значений

Ключевое слово **complement** обозначает список таких значений, которые не будут приниматься в качестве входящих значений (то есть этот список является дополнением к списку значений). Могут использоваться все значения всех типов.

Каждое значение в таком списке должно иметь тип, объявленный для поля шаблона, в котором используется дополнение. Поле шаблона, в котором используется дополнение, соответствует входящему полю тогда и только тогда, когда входящее поле не соответствует никакому значению из перечисленных в этом списке значений. Список значений, безусловно, может быть и списком с одним значением.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{
   complement (1, 3, 5),    // список неприемлемых значений integer
   :
   field3 not (true)          // сопоставление даст false
   :
}
```

B.1.2.3 Любое значение

Символ сопоставления "?" (AnyValue) используется для указания, что любое действительное входящее значение приемлемо. Он может использоваться со значениями всех типов. Поле шаблона, в котором используется механизм any value, соответствует входящему полю тогда и только тогда, когда входящее поле сводится к одиночному элементу указанного типа.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{
   field1 : = ?,    // будет соответствовать любому integer
   field2 : = ?,    // будет соответствовать любому непустому значению charstring
   field3 : = ?,    // будет соответствовать true или false
   field4 : = ?   // будет соответствовать любой последовательности integer
}
```

B.1.2.4 Любое значение или ничего

Символ сопоставления "*" (AnyValueOrNone) используется для указания, что любое действительное входящее значение, включая пропуск этого значения, приемлемо. Он может использоваться со значениями всех типов, если поле шаблона объявлено факультативным.

Поле шаблона, в котором используется этот символ, соответствует входящему полю тогда и только тогда, когда входящее поле сводится к любому элементу указанного типа или входящее поле отсутствует.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{  :
   field3 : = *,   // будет соответствовать true или false, или "пропущенное поле"
   :
}
```

B.1.2.5 Диапазон значений

Диапазон обозначает ограниченную область приемлемых значений. Если он используется для значений типа **integer** или **float** (и подтипов **integer** или **float**). Граничным значением может быть:

a) бесконечность или минус бесконечность;

b) выражение, которое сводится к конкретному значению **integer**.

Нижняя граница ставится с левой стороны диапазона, а верхняя граница – с правой стороны. Нижняя граница должна быть меньше верхней границы. Поле шаблона, в котором используется диапазон, соответствует входящему полю тогда и только тогда, когда значение входящего поля равно одному из значений диапазона.

Когда в шаблонах или областях шаблона используется типы **charstring** или **universal charstring**, то при оценке границ мы должны получать действительные позиции для символа в кодовой таблице(ах) наборов символов типа (например, данное положение не должно быть пустым). Пустые позиции между нижней и верхней границей не рассматриваются как действительные значения для указанного диапазона.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{
   field1 : = (1 . . 6),    // диапазон типа integer
   :
   :
   :
}
// другими записями для field1 могли быть (-∞…8) и (12…∞)
```

B.1.2.6 SuperSet

Супермножество (SuperSet) – это операция соответствия, которая должна использоваться только для величин типов **set of**. Супермножество обозначается ключевым словом **superset**. Поле, которое использует супермножество, соответствует согласованному входящему полю тогда и только тогда, когда входящее поле содержит по крайней мере все элементы, определенные в пределах супермножества, а также может содержать дополнительные. Аргумент супермножество должен иметь тип, объявленный для поля, в котором используется механизм супермножество.

ПРИМЕР:

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// любая последовательность сочетаний целых чисел, которая содержит по крайней мере одно
// появление чисел 1, 2 и 3 в любом порядке и месте
```

B.1.2.7 SubSet

Подмножество является операцией соответствия, которая должна использоваться только для величин типов **set of**. Подмножество обозначается ключевым словом **subset.** Поле, которое использует подмножество, соответствует соответствующему входному полю тогда и только тогда, когда входящее поле содержит только элементы, определенные в пределах подмножества, и может содержать меньше. Аргумент подмножество должен иметь тип, объявленный в поле, в котором используется механизм подмножества.

ПРИМЕР:

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// любая последовательность сочетаний целых чисел, которая содержит ноль или одно
// появление чисел 1, 2 и 3 в любом порядке и месте
```

B.1.3 Механизмы сопоставления внутри значений

B.1.3.0 Общие положения

Следующие механизмы сопоставления могут использоваться в явных значениях типа **strings, records, records of, sets, sets of** и массивах

B.1.3.1 Любой элемент

Символ сопоставления "?" (*AnyElement*) используется для указания на то, что он заменяет одиночные элементы в цепочке (за исключением цепочки знаков), в **record of**, **set of** или в массиве. Он используется только в рамках значений типов **string**, **record of**, **set of** и массивов.

ПРИМЕР:

```
template Mymessage MyTemplate:=
{  :
   field2 : = "abcxyz",
```

```
    field3 : = '10???'B, / / где каждый "?" может быть либо 0, либо 1
    field4 : = {1, ?, 3} / / где "?" может быть любым значением integer
}
```

ПРИМЕЧАНИЕ. – Символ "?" в field4 может интерпретироваться как *AnyValue* в виде значения integer либо как *AnyElement* внутри **record of, set of** или массива. Так как обе интерпретации ведут к одному и тому же соответствию, проблемы не возникает.

B.1.3.1.1 Использование символа подстановки с одним знаком

Если требуется выразить символ подстановки "?" в цепочке знаков, то это следует делать с использованием комбинации знаков (см. п. B.1.5). Например, "abcdxyz", "abccxyz", "abcxxyz" и т. д. будут соответствовать **pattern** "abc?xyz". Однако "abcxyz", "abcdefxyz" и т. д. не будут соответствовать.

B.1.3.2 Любое число элементов или их отсутствие

Символ сопоставления "*" (AnyElementsOrNone) используется для указания, что он заменяет нулевое или любое число последовательных элементов цепочки (кроме цепочек знаков), или **record of**, **set of**, или массива. Символ "*" соответствует максимально длинной последовательности элементов согласно комбинации, определяемой символами, окружающими "*". Например:

ПРИМЕР:

```
template Mymessage MyTemplate : =
{  :
   field2 : = "abcxyz",
   field3 : = '10*11'B, // Где "*" может быть любой последовательностью битов
       // (возможно, пустой)
   field4 : = {*, 2, 3} // Где "*" может быть любым значением integer
       // или пропущенным значением
}
var charstring MyStrings [4];
MyPCO.receive (MyStrings : {"abyz", *, "abc"});
```

Если символ "*" появляется на высшем уровне внутри цепочки, типов **record of**, **set of** или массива, то его следует интерпретировать как *AnyElementOrNone*.

ПРИМЕЧАНИЕ. – Это правило препятствует возможной в противном случае интерпретации "*" как *AnyValueOrNone*, которое заменяет элемент внутри цепочки, типов **record of, set of** или массива.

B.1.3.2.1 Использование нескольких символов подстановки знаков

Если требуется выразить символ подстановки "*" в цепочке знаков, то это следует делать с использованием комбинации знаков (см. п. C.1.5). Например, "abcxyz", "abcdefxyz", "abcabcxyz" и т. д. будут соответствовать **pattern** "abc*xyz".

B.1.3.3 Перестановка

Перестановка - операция соответствия, которая должна использоваться на величинах только типов **record of**. Перестановка обозначена ключевым словом **permutation**. Выражения и AnyElement и AnyElementsOrNone могут быть элементами перестановки. Каждый элемент, перечисленный в перестановке, должен иметь тип, повторяющий тип **record of**.

Перестановка вместо единственного элемента означает, что любой ряд элементов является приемлемым, если он содержит те же самые элементы как список величины в перестановке, даже, возможно, в другом порядке. Если и перестановка и AnyElementsOrNone будут использоваться в значении, то они должны быть оценены совместно.

AnyElementsOrNone, используемый в перестановке, заменяет либо ни один, либо любой ряд элементов в пределах сегмента значения записи, соответствующей перестановке. AnyElementsOrNone, используемый в перестановке, должен быть оценен в последнюю очередь (когда все другие элементы списка перестановки уже сопоставлены элементу в аттестованном списке).

ПРИМЕЧАНИЕ 1. – *AnyElementsOrNone*, используемый в перестановке, имеет различные последствия, в отличии от *AnyElementsOrNone*, используемого вместе с перестановкой, поскольку в последнем *AnyElementsOrNone* заменяет только последующие элементы . Например, {**permutation**(1,2, *)} является эквивалентным логическим элементом ({*, 1, *, 2, *}, {*, 2, *, 1, *}), в то время как {**permutation**(1,2), *} является эквивалентным логическим элементом ({1,2}, {2,1}, *).

К ПРИМЕЧАНИЕ 2. – Когда *AnyElementsOrNone* используется вместе с перестановкой, атрибут продолжительности может применяться к *AnyElementsOrNone*, чтобы ограничить ряд элементов, которому соответствует *AnyElementsOrNone* (см. также B.1.4.1). Напротив, никакой атрибут продолжительности не должен быть добавлен к *AnyElementsOrNone*, используемому в перестановке (но может быть применен вместо этого к целой перестановке).

ПРИМЕР:

```
type record of integer MySequenceOfType;
```

```
   template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
   // соответствует любой из следующих последовательностей 4 целых чисел: 1,2,3,5; 1,3,2,5;
2,1,3,5;
   // 2,3,1,5; 3,1,2,5; or 3,2,1,5

   template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
   // соответствует любой последовательности 4 целых чисел, которая заканчивается 5 и содержит 1
и 2   // по крайней мере однажды в других позициях

   template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
   // соответствует  любой последовательности начинающейся с целых чисел 1,2,3; 1,3,2; 2,1,3;
2,3,1; 3,1,2 or 3,2,1

   template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
   // соответствует  любой последовательности кончающейся  целыми числами 1,2,3; 1,3,2; 2,1,3;
2,3,1; 3,1,2 or 3,2,1

   template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
   // соответствует  любой последовательности  целых чисел, содержащих любую из следующих
подстрок в любой позиции:
   // 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

   template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
   // соответствует  любой последовательности  целых чисел, которая заканчивается 5 и содержащий
   // 1 и 2 по крайней мере однажды в других позициях

   template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
   // соответствует  любой последовательности   трех - восьми начинающихся  с целых чисел 1,2,3;
   // 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

   template MySequenceOfType MyTemplate9 := { permutation ( 1, 2, *) length (3..5), 5 };
   // соответствует  любой последовательности    четырех - шести целых чисел, которая
   // заканчивается 5 и содержит 1 и 2 по крайней мере однажды в другой позиции
```

B.1.4 Сопоставление атрибутов значений

B.1.4.0 Общие положения

Следующие атрибуты могут быть связаны с соответствующими механизмами.

B.1.4.1 Ограничения длины

Атрибут ограничения длины используется для ограничения длины значений цепочки и числа элементов в структуре **set of** , **record of** или массива. Он используется только как атрибут в следующих механизмах: *AnyValue, AnyValueOrNone, AnyElement* и *AnyElementOrNone* (но не внутри перестановки), перестановке, супермножество или подмножество. Он может использоваться также в сочетании механизмом соответствия и с атрибутом **ifpresent**. Синтаксис **length** можно найти в пп. 6.2.3 и 6.3.3.

ПРИМЕЧАНИЕ. – Когда соответствующие механизмы и дополнения и ограничения продолжительности используются для поля шаблона или шаблона, ограничения, подразумеваемые ими, должны применяться независимо к полю шаблона или шаблона.

Единицы длины в случае значений цепочки должны интерпретироваться согласно таблице 4. Для типов **set of**, **record of** и множества единица длины будет копированного типа. Границы должны обозначаться выражениями, которые сводятся к конкретным неотрицательным значениям **integer**. Альтернативно может использоваться ключевое слово **infinity** в качестве значения для верхней границы, чтобы указать на отсутствие верхнего предела длины.

Определения длины для шаблона не должны противоречить длине для ограничений (если таковая имеется) соответствующего типа. Поле шаблона, в котором **length** используется в качестве атрибута какого-либо символа, соответствует входящему полю тогда и только тогда, когда входящее поле соответствует как этому символу, так и связанному с ним атрибуту. Атрибут **length** соответствует, если длина входящего поля больше или равна указанной нижней границе и меньше или равна верхней границе. В случае одиночного значения длины атрибут **length** соответствует, если только длина принимаемого поля точно равна указанному значению.

Можно использовать ограничение продолжительности вместе со специальной величиной **omit**; однако в этом случае атрибут продолжительности не имеет никакого эффекта (то есть, с **omit** он – избыточен). При наличии AnyValueOrNone и ifpresent он содержит ограничение на входящее значение, если оно имеется.

ПРИМЕР:

```
template Mymessage MyTemplate : =
{
  field1 : = complement ({4,5},{1,4,8,9}) length (1 . . 6),  // любое значение, содержащее  1,
2, 3, 4,
           // 5 или 6 элементов приняты, если оно не {4,5}
           // или {1,4,8,9})
  field2 : = "ab*ab" length (13)          // Максимальная длина цепочки
           // AnyElementOrNone составляет 9 знаков
```

B.1.4.2 Индикатор IfPresent

Индикатор **ifpresent** указывает, что сопоставление может производиться, если присутствует (то есть не опущено) какое-либо факультативное поле. Этот атрибут может использоваться со всеми механизмами сопоставления при условии, что тип объявлен факультативным.

Поле шаблона, в котором используется **ifpresent**, соответствует входящему полю тогда и только тогда, когда входящее поле соответствует согласно примененному механизму сопоставления либо входящее поле отсутствует.

ПРИМЕР:

```
template Mymessage:MyTemplate:=
{   :
    field2 : = "abcd" ifpresent,  // сопоставляется "abcd", если не опущено
    :
:
}
```

ПРИМЕЧАНИЕ. – Смысл *AnyValueOrName* точно такой же, как у ? **ifpresent**.

B.1.5 Сопоставление комбинации знаков

B.1.5.0 Общие положения

Комбинации знаков могут использоваться в шаблонах для определения формата требуемой цепочки знаков, подлежащей приему. Комбинации знаков могут использоваться для сопоставления значений charstring и universal charstring. Кроме буквенных знаков в комбинациях знаков допускается использование мета-знаков (то есть ? и * в пределах комбинации знаков подразумевает соответствие любого символа и любого числа любому символу соответственно).

ПРИМЕР 1:

```
template charstring MyTemplate : = pattern "ab??xyz*0";
```

Этот шаблон будет соответствовать любой цепочке знаков, которая содержит знаки «ab», за которыми следуют любые два знака, за которыми в свою очередь следуют знаки «xyz», а затем любое число любых знаков (включая любое число «0») перед заключительным знаком «0».

Если потребуется считать любой метазнак обычным знаком, то перед ним должен стоять метазнак '\'.

ПРИМЕР 2:

```
template charstring MyTemplate : = pattern "ab?\?xyz*";
```

Этот шаблон будет соответствовать любой цепочке знаков, которая содержит знаки 'ab', за которыми следуют любой знак, за которыми в свою очередь следуют знаки '?xyz', а затем любое число любых знаков.

Список метазнаков для шаблонов TTCN-3 приведен в таблице B.1. Метазнак не может содержать пробелов, за исключением пробела, следующего за концом строки знака перед или в выражении набора.

Таблица B.1/Z.140	Список метазнаков шаблона TTCN-3
Метазнак	**Описание**
?	Соответствует любому знаку (см. Примечание 1 и 2)
*	Соответствует любому знаку ноль или больше раз; будет соответствовать самому длинному числу знаков (см. пример 1 выше) (см. Примечания 1 и 2)
\	Приводит к тому ,что следующий метазнак интерпретируется как символьная константа (см. Примечание 3). Предшествуя символу без определенного метазнака значение '\' и символу вместе соответствует знаку после '\' (см. Примечание 4)
[ ]	Соответствует любому знаку в пределах указанного набора (см. B.1.5.1 для более точной информации)
-	Имеет метазнак, означающий в паре квадратных скобок ("["и"]") только, кроме первых и последних положений в пределах скобки. Позволяет определять диапазон знаков (см. B.1.5.1 для более точной информации)
^	Имеет метазнак, означающий как первый символ после открывающейся квадратной скобки в паре квадратных скобок ("["и"]") и соответствует любому знаку, комплементарному набору знаков после этого метазнака (см. B.1.5.1 для более точной информации)
\q{group,plane,row,cell}	Соответствует универсальному знаку, определенному четверкой
{reference}	Вставляет справочную определенную пользователем последовательность и интерпретирует это как регулярное выражение (см. B.1.5.2 для более точной информации)
\N{reference}	Соответствует любому символу в пределах набора знаков, где набор определен по соответствующему определению (см. B.1.5.4 для более точной информации)
\d	Соответствует любой цифре (эквивалентный [0-9])
\w	Соответствует любому алфавитно-цифровому знаку (эквивалентный [0-9a-zA-Z])
\t	Соответствует C0 знаку управления HT (9) (см. ISO/IEC 6429 [11])
\n	Соответствует любому из следующих за C0 знаков управления : LF (10), VT (11), И СЛЕДУЮЩИЕ (12), CR (13) (см. ISO/IEC 6429 [11]) (совместно вызванные знаки новой строки)
\r	Соответствует C0 знаку управления CR (см. ISO/IEC 6429 [11])
\s	Соответствует любому знаку управления из следующих за C0: HT(9), LF(10), VT(11), FF(12), CR(13), SP(32) (см. ISO/IEC 6429 [11], ITU-T Rec. T.50 [9]) (совместно вызванные знаки пробела)
\b	Соответствует границе слова (любой графический символ кроме SP или DEL предшествуют или сопровождается любым из знаков пробел или новая строка)
\"	Соответствует знаку двойных кавычек
""	Соответствует знаку двойных кавычек
|	Используется, чтобы обозначить два альтернативных выражения
( )	Используется, чтобы сгруппировать выражение
#(n, m)	Соответствует предыдущему выражению по крайней мере n раз, но не больше, чем м. раз (постфикс) (см. B.1.5.3 для большего количества деталей)
#n	Соответствует предыдущему выражению точно n раз (где n - единственная цифра) (постфикс); то же самое как # (n)
+	Соответствует предыдущему выражению один или несколько раз (постфикс); то же самое как # (1),
ПРИМЕЧАНИЕ 1. – Метазнаки ? и * могут соответствовать любым знакам набора символов основных типов шаблона или области шаблона, в которой они используются (то есть, не рассматривая примененные ограничения типа). Однако нельзя забывать, что операции приема требуют проверки типа полученного сообщения прежде, чем попытаться соответствовать этому. Поэтому принятые величины, не соответствующие спецификации подтипа области шаблона или шаблона, никогда не предоставляются для сочетания	
ПРИМЕЧАНИЕ 2. – В некоторых других языках/указаниях ? и * имеют различные значения как метазнаки. Однако в TTCN эти символы традиционно используются для того, чтобы соответствовать, как определено в этой таблице.	
ПРИМЕЧАНИЕ 3. – Следовательно, символ наклонной черты влево может быть подобран парой символов наклонной черты влево без места между ними (\\), например, образец "\\d" будет соответствовать последовательности "\d"; открывающиеся или закрывающиеся квадратные скобки соответствовать "\["и" \] "соответственно, и т.д.	
ПРИМЕЧАНИЕ 4. –Такое использование метазнака "\" не рекомендуется, поскольку дальнейшие метазнаки могут быть определены позже.	

B.1.5.1 Выражение set (набор)

Список символов, расположенных между парой "[" и "]", соответствует любому единственному символу в этом списке. Выражение набора ограничено символами "[" "]" . В дополнение к символьной константе можно описать зоны символа, используя дефис "-" как разделитель. Зона состоит из символа непосредственно перед разделителем, символа сразу после него и всех символов с кодом символа между двух граничащих символов. Символ дефиса "-" внутри списка, но не предшествуя или после символа, теряет свое специальное значение.

Выражение набора может также быть инвертировано, помещая знак вставки "^" символ как первый символ после открывающейся квадратной скобки. Отрицание имеет приоритет по зонам символа. Поэтому дефис "-" сразу после знака вставки отрицания "^" должен быть обработан как буквенный символ.

Запрещены пустые списки и пустые инвертированные списки. Поэтому заключительная квадратная скобка "]" сразу после открывающейся квадратной скобки "[" или знак вставки после открывающейся квадратной скобки "[" и немедленно сопровождаемый заключительная квадратная скобка "]" должны быть обработаны как буквенные символы.

Все метазнаки, кроме упомянутых ниже, теряют свое специальное значение в списке:

- "]" не находится в начальной позиции и не немедленно после "^";
- "-" не в первой или последней позиции в списке;
- "^" в первой позиции в списке кроме тех случаев, когда немедленно сопровождается заключительной квадратной скобкой;
- "\", \d, \t, \w, \r, \n, \s и \b;
- \q {группа, самолет, ряд, ячейка};
- **\N** {ссылка}.

ПРИМЕЧАНИЕ 1. – Вложенные списки не разрешены (например в образце "[ab [r-z]]" второе "[", обозначает символьную константу "[", первое "]" закрывает список, и второе "]" вызывает ошибку, поскольку нет никакой связанной вводной скобки в образце).

ПРИМЕЧАНИЕ 2. – Чтобы включить символ знака вставки "^", поместите его куда-нибудь кроме первой позиции или поставьте перед ним наклонную черту (backslash). Чтобы включить литерал дефис "-", поместите его сначала или в конец списка, или поставьте перед ним наклонную черту (backslash). Чтобы включать заключительную квадратную скобку "]", поместите ее сначала или поставьте перед ним ему наклонную черту (backslash). Если первый символ в списке - знак вставки "^", то символы "-" и "]" соответствуют себе, если они сразу следуют за этем знаком вставки.

ПРИМЕР:

```
template charstring RegExp1:= pattern '[a-z]';  // это будет соответствовать любому символу
//от a до z

template charstring RegExp2:= pattern '[^a-z]';  // это будет соответствовать любому символу
//кроме  символов от a до z
template charstring RegExp3:= pattern '[AC-E][0-9][0-9][0-9]YKE';
// RegExp3 будет соответствовать последовательности, которая начинается с буквы A или букв
// между C и E (но,  например, не B) и имеет три цифры и буквы YKE
```

B.1.5.2 Выражение Ссылки

В дополнение к непосредственным строковым значениям также возможно в пределах образца использовать ссылки на существующие шаблоны, константы, переменные или параметры модуля. Ссылка, заключенная в пределах "{" "}", символы и ссылка должна свестись к одному из типов строки символов. Содержание шаблонов констант или переменных, на которые ссылаются, должно быть обработано как регулярное выражение. Каждое выражение должно быть разыменовано только однажды.

ПРИМЕР:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern '{MyString}';
```

Этот шаблон соответствовал бы любой строке символов, которая состоит из символов 'ab', сопровождаемый любым символом. В действительности, любая строка символов после ключевого слова pattern или явно или ссылкой будет интерпретироваться по правилам, определенным в этом пункте.

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

Этот шаблон соответствовал бы любой строке символов, которая состоит из символов 'ab', сопровождаемый любым символом, сопровождаемым символами 'de', сопровождаемый символом с group=1, plane=1, row=13 и cell=7 в ISO/IEC 10646 .

Если выражение ссылки обращается к шаблону, постоянному или переменному, который содержит одно или более выражений ссылки, то ссылки в отнесенном шаблоне, постоянном или переменном, должны рекурсивно быть разименованы прежде, чем вставить их содержание в относящийся образец.

ПРИМЕР:

```
const charstring MyConst2 := pattern "ab";
template charstring RegExp1 := pattern "{MyConst2}";
   // соответствует строке "ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
   // соответствует строке "abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
   // соответствует строке "cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
  // соответствует только строке "{RegExp1}" (то есть не будет обработан как выражение ссылки
   // к шаблону RegExp1),
```

B.1.5.3 Выражение соответствия n раз

Для указания на то, что предыдущее выражение должно быть сопоставлено неоднократно, должен использоваться один из следующих синтаксисов : "# (n, m)", "# (n,)", "# (m)", "# (n)", "#n" или "+". Форма "# (n, m)" определяет, что предыдущее выражение должно быть подобрано по крайней мере n раз, но не больше чем m раз. Постфикс метазнака "# (n,)" определяет, что предыдущее выражение должно быть сопоставлено по крайней мере n раз, в то время как "# (m)" указывает, что предыдущее выражение должно быть сопоставлено не больше чем m раз. Метазнаки (постфиксы) "# (n)" и "#n" определяют, что предыдущее выражение должно быть сопоставлено точно n раз (они эквивалентны "# (n, n)"). В форме "#n", n должен быть единственной цифрой. Постфикс метазнака "+" обозначает, что предыдущее выражение должно быть сопоставлено по крайней мере в 1 раз (эквивалентно "# (1),").

ПРИМЕР:

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)'; // соответствует по крайней мере 9, но
                                                       // не больше, чем 11 символам от a до z
template charstring RegExp5a:= pattern '[a-z]#(9)';       // соответствует точно 9
                                                          // символам от a до z
template charstring RegExp5b:= pattern '[a-z]#9';         // соответствует точно 9
                                                          // символам от a до z
template charstring RegExp6:= pattern '[a-z]#(9, )';      // соответствует по крайней мере 9
                                                          // символам от a до z
template charstring RegExp7:= pattern '[a-z]#(, 11)';     // соответствует не более 11
                                                          // символам от a до z
template charstring RegExp8:= pattern '[a-z]+';           // соответствует по крайней мере 1
                                                          // символу от a до z
```

B.1.5.4 Соответствие набора символов, на который ссылаются

Примечание формы "\N {reference}", где reference обозначает шаблон длиной в один знак, постоянную, переменную или параметр модуля, соответствует символу в значении, на которую ссылаются, или шаблон.

Ссылаясь на шаблон, постоянную, переменную или параметр модуля, который имеет длину не 1, должно вызвать ошибку.

Примечание формы "\N {typereference}", где "typereference" - ссылка на типы charstring или universal charstring, соответствует любому символу набора символов, обозначенного типом, на который ссылаются.

ПРИМЕЧАНИЕ 1. – Случаи, когда набор символов, на который ссылаются, не истинное подмножество величин, позволенных по определению типа области шаблона или шаблона, для которой используется образец символа, нельзя рассматривать как ошибка (но, например, соответствие никогда не может происходить, если два набора не накладываются).

ПРИМЕЧАНИЕ 2. – \N {**charstring**} эквивалентна ? , когда последний применен к области шаблона или шаблону типа **charstring,** и \N {**universal, charstring**} эквивалентен ? , когда последний применен к области шаблона или шаблону типа **universal charstring** (но вызывает ошибку если применен к области шаблона или шаблону типа **charstring**).

ПРИМЕР:

```
type charstring MyCharRange ('a'..'z');
type charstring MyCharList ('a', 'z');
const MyCharRange myCharR := 'r';
```

```
template charstring myTempPatt1 := pattern '\N { myCharR }';
// myTempPatt1 должен соответствовать только 'r'

template charstring myTempPatt2 := pattern '\N { MyCharRange }';
// myTempPatt2 должен соответствовать любой строке, содержащей один знак от a до z

template MyCharRange myTempPatt3 := pattern '\N { MyCharList }';
// myTempPatt3 должен соответствовать только 'a' и 'r'

template MyCharList myTempPatt4 := pattern '\N { MyCharRange }';
// myTempPatt4 должен соответствовать только 'a' и 'r'
```

B.1.5.5 Правила совместимости типа для образцов

Для образцов (см. B.1.5.2) и наборов символов (см. B.1.5.4), на которые ссылаются , применяются определенные правила совместимости типа: тип, на который ссылаются, шаблон, постоянная, переменная или параметр модуля типа **charstring**, могут всегда использоваться в спецификации образца области шаблона или шаблона типа **universal charstring**; тип, на который ссылаются, шаблон или величина типа **universal charstring** может использоваться в спецификации образца области шаблона или шаблона типа **charstring**, если все символы, используемые в шаблоне, на который ссылаются, или величине, и наборе символов, разрешенном типом, на который ссылаются, имеет их соответствующие символы в типе charstring (см. определение соответствующих символов в 6.7.1).

Приложение C

Предопределенные функции TTCN-3

В данном Приложении описываются предопределенные функции TTCN-3.

C.0 Общая процедура обработки исключений

Ошибочные ситуации (например, входной параметр вне позволенного диапазона, входной параметр имеет неправильный тип, входная величина содержит неподходящий символ, и т.д.), для которого никакое явное правило обработки исключений не определено в соответствующих пунктах этого приложения, вызовет ошибку во время компилирования или во время работы TTCN-3. Какие ошибочные ситуации вызовут ошибку во время компилирования, а какие - ошибку во времени выполнения – зависит от выбора реализации инструментария.

C.1 Целое число в знак

```
int2char (integer value) return charstring
```

Эта функция преобразует значение **integer** в диапазоне от 0 до 127 (8-битовое кодирование) в charstring значение, длиной в один знак. Значение **integer** описывает 8-битовое кодирование знака.

C.2 Знак в целое число

```
char2int (charstring  value) return integer
```

Эта функция преобразует charstring значение, длинной в один знак, в значение **integer** в диапазоне от 0 до 127 . Целое значение описывает 8-битное кодирование знака.

C.3 Целое число в универсальный знак

```
int2unichar (integer value) return universal charstring
```

Эта функция преобразует значение **integer** в диапазоне от 0 до 2 147 483 647 (32-битное кодирование) в **universal** charstring значение, длиной в один знак. Целое значение описывает 32-битное кодирование знака.

Результатом функции будет –1, если значение аргумента отрицательно или превышает 268435455.

C.4 Универсальный знак в целое число

```
unichar2int (universal charstring  value) return integer
```

Эта функция преобразует **universal charstring** значение, длиной в один знак, в значение **integer** в диапазоне от 0 до 2 147 483 647. Значение integer описывает 32-битовое кодирование знака.

C.5 Цепочка битов в целое число

```
bit2int (bitstring value) return integer
```

Эта функция преобразует одиночное значение **bitstring** в одиночное значение **integer**.

При этом преобразование **bitstring** следует рассматривать как значение **integer** с положительным основанием 2. Самый правый бит является младшим разрядом, а самый левый бит – старшим. Биты 0 и 1 представляют десятичные значения "0" и "1" соответственно.

C.6 Шестнадцатеричная цепочка в целое число

```
hex2int (hexstring value) return integer
```

Эта функция преобразует одиночное значение **hexstring** в одиночное значение **integer**.

При этом преобразовании **hexstring** следует рассматривать как значение **integer** с положительным основанием 16. Самая правая шестнадцатеричная цифра является младшим разрядом, а самая левая – старшим. Шестнадцатеричные цифры от 0 до F представляют десятичные значения от 0 до 15 соответственно.

C.7 Цепочки октетов в целое число

```
oct2int (octetstring value) return integer
```

Эта функция преобразует одиночное значение **octetstring** в одиночное значение **integer**.

При этом преобразовании **octetstring** следует рассматривать как значение **integer** с положительным основанием 16. Самая правая шестнадцатеричная цифра является младшим разрядом, а самая левая – старшим. Число выданных шестнадцатеричных цифр должно быть кратно 2, так как один октет состоит из двух шестнадцатеричных цифр. Шестнадцатеричные цифры 0…F представляют десятичные значения 0…15 соответственно.

C.8 Цепочка знаков в целое число

```
str2int (charstring value) return integer
```

Эта функция преобразует **charstring**, представляющую значение **integer**, в эквивалентное **integer**.

ПРИМЕР:

```
str2int ("66")   // будет выдавать integer со значением 66
str2int ("-66")  // будет выдавать integer со значением -66
str2int ("abc")  // сгенерирует ошибку компилятора или тестового варианта
str2int ("0")    // будет выдавать integer со значением 0
```

C.9 Целое число в цепочку битов

```
int2bit (in integer value, in integer length) return bitstring
```

Эта функция преобразует одиночное значение **integer** в одиночное значение **bitstring**. Полученная цепочка имеет длину **length**.

При этом преобразовании **bitstring** следует рассматривать как положительное значение **integer** с основанием 2. Самый правый бит является младшим разрядом, а самый левый – старшим. Биты 0 и 1 представляют десятичные значения 0 и 1 соответственно. Если преобразование выдает значение с числом битов, которое меньше указанного в параметре **length**, то **bitstream** должна заполняться слева нулями.

C.10 Целое число в шестнадцатеричную цепочку

```
int2bit (in integer value, in integer length) return hexstring
```

Эта функция преобразует одиночное значение **integer** в одиночное значение **hexstring**. Полученная цепочка содержит число шестнадцатеричных цифр, равное **length**.

При этом преобразовании **hexstring** следует рассматривать как положительное значение **integer** с основанием 16. Самая правая шестнадцатеричная цифра является самым младшим разрядом, а самая левая – старшим. Шестнадцатеричные цифры от 0 до F представляют десятичные значения от 0 до 15 соответственно. Если преобразование выдает значение с числом шестнадцатеричных цифр, которое меньше указанного в параметре **length**, то hexstring должна заполняться слева нулями.

C.11 Целое число в цепочку октетов

```
int2bit (in integer value, in integer length) return octetstring
```

Эта функция преобразует одиночное значение **integer** в одиночное значение **octetstring**. Полученная цепочка содержит число октетов **length**.

При этом преобразовании **octetstring** следует рассматривать как положительное значение **integer** с основанием 16. Самая правая шестнадцатеричная цифра является самым младшим разрядом, а самая левая – старшим. Число выданных шестнадцатеричных цифр должно быть кратно 2, так как один октет состоит из двух шестнадцатеричных цифр. Шестнадцатеричные цифры от 0 до F представляют десятичные значения от 0 до 15 соответственно. Если преобразование выдает значение с числом шестнадцатеричных цифр, которое меньше указанного в параметре **length**, то **hexstring** должна заполняться слева нулями.

C.12 Целое число в цепочку знаков

```
int2str (integer value) return charstring
```

Эта функция преобразует значение **integer** в его цепочечный эквивалент (основание выдаваемой цепочки всегда будет десятичным).

ПРИМЕР:

```
int2str ("66")    //будет выдавать charstring со значением 66
int2str ("-66")   //будет выдавать charstring со значением -66
int2str ("0")  //будет выдавать charstring со значением 0
```

C.13 Длина для типа "цепочка"

```
lengthof (any_string_type value) return integer
```

Эта функция выдает длину значения, которое имеет тип **bitstring**, **hexstring**, **octetstring** или цепочки любых знаков. Единицы длины для каждого типа цепочки определены в таблице 4.

Длина **universal charstring** должна быть вычислена, считая каждый символ объединения и символ слога хангула (включая наполнители) самостоятельно (см. ISO/IEC 10646 [10], и пункты 23 и 24).

ПРИМЕР:

```
lengthof ('010'B) // выдает 3
lengthof ('F3'H)  // выдает 2
lengthof ('F2'O)  // выдает 1
lengthof (universal charstring : "Length_of_Example")// выдает 17
```

C.14 Число элементов в структурированном значении

```
sizeof (structured_type value) return integer
```

Эта функция выдает реальное число элементов в типе **record, record of, set, set of, template** или в массиве.

Эта функция выдает реальное число элементов параметра модуля, постоянных, переменных или шаблонов типа **record, record of, set, set of** или множества (см. Примечание). В случае **record of** и **set** of величин, шаблонов или множества, фактическая величина, которая будет возвращена, является последовательным номером последнего определенного элемента (индекс этого элемента плюс 1).

ПРИМЕЧАНИЕ. – Вычисляются только элементы объекта TTCN-3, который является параметром функции; то есть, никакие элементы вложенных типов/величин не принимаются во внимание при определении величины возвращения.

ПРИМЕР:

```
// Дано
type record MyPDU
   {   boolean field1  optional,
       integer field2
       };

   template MyPDU  MyTemplate
       { field1  omit,
         field2 5
       };

   var integer numElements;

   // тогда

   numElements := sizeof(MyTemplate);    // returns 1

   // Дано
   type record length(0..10) of integer MyList;
   var MyList MyRecordVar;
   MyRecordVar := { 0, 1, omit, 2, omit };
   // тогда
   numElements := sizeof(MyRecordVar);
   // возвращает 4 безотносительно тому, что элемент MyRecordVar [2] неопределен
```

C.15 Функция IsPresent

```
ispresent (any_type value) return boolean
```

Эта функция, разрешенная только для типов **record** и **set** ввыдает значение **true** (ИСТИНА) тогда и только тогда, когда значение указанного поля присутствует в реальном экземпляре указанного объекта данных. Аргументом в **ispresent** должна быть ссылка на поле типа **record** и **set**.

```
// Дано
type record MyRecord
   {   boolean field1 optional,
       integer field2
   }
// и дано, что MyPDU является шаблоном типа MyRecord,
// а received_PDU также относится к типу MyRecord,
// тогда
MyPort.receive (MyPDU) - > value received PDU
ispresent (received_PDU.field1)
// выдает true, если field1 присутствует в реальном экземпляре MyPDU
```

C.16 Функция IsChosen

```
ischosen (any_type value) return boolean
```

Эта функция выдает значение **true** тогда и только тогда, когда ссылка на объект данных определяет тот вариант типа **union**, который реально выбран для заданного объекта данных.

ПРИМЕР:

```
/ / Дано
type union MyUnion
   {   PDU_type1   p1,
       PDU_type2   p2,
       PDU_type p3
   }
/ / и дано, что MyPDU является шаблоном типа MyUnion,
/ / а received_PDU также относится к типу MyUnion,
/ / тогда
MyPort.receive (MyPDU) - > value received_PDU
ischosen (received_PDU.p2)
/ / выдает true, если реальный экземпляр MyPDU переносит PDU типа PDU_type2
```

C.17 Функция Regexp

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
  character_string_type
```

Эта функция возвращает подпоследовательность входной строки символов `instr`, которая является содержанием n-ной группы, соответствующего выражения. Во входной последовательности `instr` может быть любой тип строки символов. Возвращаемый тип строки символов есть корневой тип `instr`. Выражение является комбинацией символов, как описано в B.1.5. Число группы, которая будет возвращена, определяется с помощью `groupno`, который должен быть положительным целым числом. Номера групп назначаются по порядку появления вводной скобки группы и считаются, начинаясь от 0 с шагом 1. Если никакая подпоследовательность, выполняющая все условия (то есть, образец и число группы), не найдена в пределах входной последовательности, то выдается пустая последовательность.

ПРИМЕР:

```
// Дано
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)'

// Тогда выражение
var charstring mystring := regexp(myinput, mypattern,1)
//выдаст значение «17».
```

C.18 Bitstring к charstring

```
bit2str (bitstring value) return charstring
```

Эта функция преобразовывает единственную величину **`bitstring`** в единственный **`charstring`**. Получающийся **`charstring`** имеет ту же самую длину как **`bitstring`** и содержит только символы '0' и '1'.

Для этого преобразования **`bitstring`** должен быть преобразован в **`charstring`**. Каждый бит **`bitstring`** преобразован в символ '0' или '1' в зависимости от величины 0 или 1 из бита. Последовательный порядок символов в получающемся **`charstring`** - тот же самый как и порядок битов в **`bitstring`**.

ПРИМЕР:

```
bit2str ('1110101'B) will return "1110101"
```

C.19 Hexstring к charstring

```
hex2str (hexstring value) return charstring
```

Эта функция преобразовывает единственную величину **`hexstring`** в единственный **`charstring`**. Получающийся **`charstring`** имеет ту же самую длину как **`hexstring`** и содержит только символы от "0" до "9" и от "A" до "F".

Для этого преобразования **`hexstring`** должен быть преобразован в **`charstring`**. Каждая шестнадцатиричная цифра **`hexstring`** преобразована в символ от "0" до "9" и от "A" до "F" в зависимости от шестнадцатиричной величины "0" до "9" и "A" до "F". Последовательный порядок символов в получающемся **`charstring`** - тот же самый, как и порядок цифр в **`hexstring`**.

ПРИМЕР:

```
hex2str ('AB801'H) will return "AB801"
```

C.20 Octetstring к строке символов

```
oct2str (octetstring invalue) return charstring
```

Эта функция преобразовывает **`octetstring`** `invalue` в **`charstring`**, представляющий последовательность, эквивалентную входной величине. У получающегося **`charstring`** должна быть та же самая длина как поступающий **`octetstring`**.

Для этого преобразования каждая шестнадцатиричная цифра `invalue` преобразована в символ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'B', 'C', 'D', 'E' или 'F', повторяя величину шестнадцатиричной цифры. Последовательный порядок символов в получающемся **charstring** - тот же самый, как и порядок шестнадцатиричных цифр в **octetstring**.

ПРИМЕР:

```
oct2str ('4469707379'O) = "4469707379"
```

C.21 Строка символов в octetstring

```
str2oct (charstring invalue) return octetstring
```

Эта функция преобразовывает последовательность типа **charstring** в **octetstring**. Последовательность invalue должна содержать символы четного числа, и каждый должен быть только графическим символом, одним из '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'b', 'c', 'd', 'e' 'f', 'B', 'C', 'D', 'E' или 'F'. У получающегося **octetstring** будет та же самая длина как у входного **charstring**.

ПРИМЕР:

```
str2oct ("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
```

C.22 Bitstring в hexstring

```
bit2hex (bitstring value) return hexstring
```

Эта функция преобразовывает единственную величину **bitstring** в единственный **hexstring**. Получающийся **hexstring** представляет ту же самую величину как и **bitstring**.

Для этого преобразования битовая строка должна быть преобразована в шестнадцатиричную строку, где битовая строка разделена на группы четырех битов, начинающихся с самого правого бита. Каждая группа четырех битов преобразована в шестнадцатиричную цифру следующим образом:

'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, и '1111'B → 'F'H.

Когда крайняя левая группа битов содержит меньше чем 4 бита, эта группа переполнена на '0'B слева, пока это не содержит точно 4 бита и преобразована впоследствии. Последовательный порядок цифр в получающей шестнадцатиричной строке - тот же самый, как и порядок групп 4 битов в битовой строке.

ПРИМЕР:

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 Hexstring в octetstring

```
hex2oct (hexstring value) return octetstring
```

Эта функция преобразовывает единственную величину **hexstring** в единственный **octetstring**. Получающийся **octetstring** представляет ту же самую величину что и **hexstring**.

Для этого преобразования **hexstring** должен быть преобразован в **octetstring**, где **octetstring** содержит ту же самую последовательность шестнадцатиричных цифр как **hexstring**, когда длина **hexstring** модуля 2 0. Иначе, получающийся **octetstring** содержит 0 в крайней левой шестнадцатиричной цифре, сопровождаемый той же самой последовательностью шестнадцатиричных цифр как в **hexstring**.

ПРИМЕР:

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 Bitstring в octetstring

```
bit2oct (bitstring value) return octetstring
```

Эта функция преобразовывает единственную величину **bitstring** в единственный **octetstring**. Получающийся **octetstring** представляет ту же самую величину как **bitstring**.

Преобразования придерживается следующего: bit2oct (величина) =hex2oct (bit2hex (величина)).

ПРИМЕР:

```
bit2oct ('111010111'B) = '01D7'O
```

C.25 Hexstring в bitstring

```
hex2bit (hexstring value) return bitstring
```

Эта функция преобразовывает единственную величину **hexstring** в единственный **bitstring**. Получающийся **bitstring** представляет ту же самую величину как **hexstring**.

С целью этого преобразования **hexstring** должен быть преобразован в **bitstring**, где шестнадцатиричные цифры **hexstring** преобразованы в группах битов следующим образом:

'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, и 'F'H → '1111'B.

Последовательный порядок групп 4 битов в получающемся **bitstring** - то же самое как порядок шестнадцатиричных цифр в **hexstring**.

ПРИМЕР:

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 Octetstring в hexstring

```
oct2hex (octetstring value) return hexstring
```

Эта функция преобразовывает единственную величину **octetstring** в единственный **hexstring**. Получающийся **hexstring** представляет ту же самую величину как **octetstring**.

С целью этого преобразования **octetstring** должен быть преобразован в **hexstring**, содержащий ту же самую последовательность шестнадцатиричных цифр как **octetstring**.

ПРИМЕР:

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 Octetstring в bitstring

```
oct2bit (octetstring value) return bitstring
```

Эта функция преобразовывает единственную величину **octetstring** в единственный **bitstring**. Получающийся **bitstring** представляет ту же самую величину как **octetstring**.

Преобразование придержится следующего: oct2bit (величина) =hex2bit (oct2hex (величина)).

ПРИМЕР:

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.28 Integer в float

```
int2float (integer value) return float
```

Эта функция преобразовывает величину **integer** в величину **float** .

ПРИМЕР:

```
int2float(4) = 4.0
```

C.29 Float в integer

```
float2int (float value) return integer
```

Эта функция преобразовывает величину **float** в величину **integer**, удаляя фракционную часть аргумента и возвращая получающееся **integer** .

ПРИМЕР:

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 Функция генератора случайных чисел

```
rnd ([float seed]) return float
```

Функция **rnd** возвращает (псевдо) случайное число меньше 1, но больше или равный 0. Генератор случайных чисел инициализируется посредством дополнительного случайного значения. Впоследствии, если никакое новое случайного значения не будет предоставлено, то последнее произведенное число будет использоваться как случайное значение для следующего случайного числа. Без предыдущей инициализации величина, вычисленная из системного времени, будет использоваться как величина случайного значения, когда **rnd** будет использоваться в первый раз.

ПРИМЕЧАНИЕ. – Каждый раз, когда функция **rnd** инициализируется с той же самой величиной случайного значения, она должна повторить ту же самую последовательность случайных чисел.

Чтобы получить случайное целое число в данном диапазоне, может использоваться следующая формула:

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
  // Здесь, upperbound и lowerbound указывают на высший и низший номер в диапазоне
```

C.31 Функция Substring

```
substr (any_string_type value, in integer index, in integer returncount) return input_string_type
```

Эта функция возвращает подпоследовательность из величины, которая имеет тип **bitstring**, **hexstring**, **octetstring**, или любую строку символов. Тип подпоследовательности – тип исходной входной величины. Отправная точка подпоследовательности для возврата определена вторым в параметре (индекс). Индексация запусков от ноля. Третий входной параметр определяет длину подпоследовательности, которая будет возвращена. Единицы длины как определены в Таблице 4.

ПРИМЕР:

```
substr ('00100110'B, 3, 4)       // выдает '0011'B

substr ('ABCDEF'H, 2, 3)         // выдает 'CDE'H

substr ('01AB23CD'O, 1, 2)       // выдает 'AB23'O

substr ("My name is JJ", 11, 2)  // выдает "JJ"
```

C.32 Число элементов в структурированном типе

```
sizeoftype(any_type value) return integer
```

Эта функция возвращает заявленный ряд элементов параметра модуля, постоянного, переменного или **template record of** или **set of** типа или множества (см. Примечание). Эта функция должна быть применена к величинам типов с ограничением длины. Фактическое число, которое будет возвращено, является последовательным числом последнего элемента не смотря на то, определена ли его величина или нет (то есть, верхний индекс длины определения типа, на котором базируется параметр функции , плюс 1).

ПРИМЕЧАНИЕ. – Вычисляются только элементы объекта TTCN-3, которые являются параметром функции; то есть, никакие элементы вложенных типов/величин не принимаются во внимание при определении величины возвращения.

ПРИМЕР:

```
// Дано
type record of integer MyPDU1;
type set length(1..8) of integer MyPDU2;
type record length(10) of integer MyPDU3;

var MyPDU1 MyRecordOfVar1;
var MyPDU2 MyRecordOfVar2;
var MyPDU3 MyRecordOfVar3;

var integer numElements;
```

```
// тогда
numElements := sizeoftype(MyRecordOfVar1);   // выдает ошибку, так как MyPDU1 не ограничен
numElements := sizeoftype(MyRecordOfVar2);   // выдает 8
numElements := sizeoftype(MyRecordOfVar3);   // выдает 10
```

C.33 Строка символов во float

```
str2float (charstring value) return float
```

Эта функция преобразовывает **charstring**, включающий число с плавающей запятой в величину **float**. Формат числа в **charstring** должен следовать правилам в 6.1.0 со следующими исключениями:

- разрешены ведущие нули ,
- разрешен ведущий '+' признак перед положительными величинами ,
- разрешен '-0.0' .

ПРИМЕР:

```
str2float('12345.6') //  то же самое, что и str2float('123.456E+02')
```

C.34 Функция Replace

```
replace (in any_string_type str, in integer ind, in integer len, in any_string_type repl)
return any_string_type
```

Эта функция заменяет подпоследовательность величины **str** в индексе **ind** длины, **len** с последовательностью величины **repl**, и возвращает получающуюся последовательность. **str** не должен быть изменен. Если **len - 0**, последовательность **repl** вставлена. Если **ind - 0**, **repl** вставлен в начале **str**. Если **ind - lengthof (str), repl** вставлен в конце **str. str** и **repl** должены быть одного типа последовательности и должены иметь тип, основанный на **bitstring**, **hexstring, octetstring,** или любую строку символов. Возвращенная последовательность имеет тот же самый тип как **str** и **repl**. Надо отметить, что индексация в последовательностях начинается с нуля.

Следующие ошибочные случаи приведут к ошибке во время компиляции или во время выполнения:

- **str** или **repl** не последовательности;
- **str** и **repl** разного типа;
- **ind** меньше **0** или больше **lengthof(str);**
- **len** меньше **0** или больше **lengthof(str);**
- **ind+len** больше **lengthof(str).**

ПРИМЕР:

```
replace ('00000110'B, 1, 3, '111'B)    // возвращает '01110110'B

replace ('ABCDEF'H, 0, 2, '123'H)    // возвращает '123CDEF'H

replace ('01AB23CD'O, 2, 1, 'FF96'O) // возвращает '01ABFF96CD'O

replace ("My name is JJ", 11, 1, "xx") // возвращает "My name is xxJ"

replace ("My name is JJ", 11, 0, "xx") // возвращает "My name is xxJJ"

replace ("My name is JJ", 2, 2, "x") // возвращает "Myxame is JJ",

replace ("My name is JJ", 12, 2, "xx") // выводит ошибку тестового варианта

replace ("My name is JJ", 13, 2, "xx") // выводит ошибку тестового варианта

replace ("My name is JJ", 13, 0, "xx") // возвращает "My name is JJxx"
```

C.35 Octetstring в строку символов

```
oct2char (octetstring invalue) return charstring
```

Эта функция преобразовывает octetstring invalue в charstring. Входной параметр invalue не должен содержать октеты величины выше чем 7F. У получающегося charstring должна быть та же самая длина как и у входного octetstring. Октеты интерпретируются как МСЭ-Т Rec. T.50 [9] кодекса (согласно IRV) и получающиеся символы приложены к возвращенной величине.

ПРИМЕР:

```
oct2char ('4469707379'O) = "Dipsy"
```

ПРИМЕЧАНИЕ. – Возвращенная строка символов может содержать неграфические символы, которые не могут быть представлены между двойными кавычками.

C.36 Строка символов в octetstring

```
char2oct (charstring invalue) return octetstring
```

Эта функция преобразовывает charstring invalue в octetstring. Каждый октет octetstring будет содержать МСЭ-Т Rec. T.50 [9] кодекса (согласно IRV) соответствующих символов invalue.

ПРИМЕР:

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

Приложение D (для информации)

Утратило значение

Примечание. – Содержание этого приложения перемещено в МСЭ-Т Рек. Z.146 [6].

Приложение E (для информации)

Библиотека используемых типов

E.1 Ограничения

Имена типов, добавленных к этой библиотеке, должны быть уникальными в пределах всего языка и в пределах библиотеки (то есть, не должно быть ни одного из названий, определенных в Приложении C). Имена, определенные в этой библиотеке, не должны использоваться пользователями TTCN-3 как идентификаторы определений, отличных от данных в этом приложении.

ПРИМЕЧАНИЕ. – Таким образом, тип определений, данных в этом приложении, может быть повторен в модулях TTCN-3, но никакой тип, отличный от того, что определен в этом приложении, не может быть определен с одним из идентификаторов, используемых в этом приложении.

E.2 Типы, используемые TTCN-3

E.2.1 Используемые простые основные типы

E.2.1.0 Знаковые и беззнаковые однобайтные целые числа

Эти типы поддерживают величины целого числа диапазона от-128 до 127 для знакового типа, и от 0 до 255 для типа беззнакового. Символика величины для этих типов - та же самая как символика величины для типа целого числа. Значения этих типов должны быть закодированы и декодированы как они были представлены в единственном байте в пределах системы, независимо от фактической используемой формы представления.

ПРИМЕЧАНИЕ. – Кодирование величин этих типов может быть тем же самым или может отличаться друг от друга и от кодирования типа целого числа (основной тип этих полезных типов) в зависимости от фактических используемых правил кодирования. Детали правил кодирования вне области этой Рекомендации.

Определения типа для этих типов:

```
type integer byte     (-128 .. 127)     with { variant "8 bit" };
type integer unsignedbyte (0 .. 255)    with { variant "unsigned 8 bit" };
```

E.2.1.1 Знаковые и беззнаковые короткие целые числа

Эти типы поддерживают величины целого числа диапазона от-32 768 - 32 767 для знакового типа, и от 0 до 65 535 для типа беззнакового. Символика величины для этих типов - та же самая как символика величины для типа целого числа. Величины этих типов должны быть закодированы и расшифрованы как они были представлены в двух байтах в пределах системы, независимо от фактической используемой формы представления.

ПРИМЕЧАНИЕ. – Кодирование величин этих типов может быть тем же самым или может отличаться друг от друга и от кодирования типа целого числа (основной тип этих полезных типов) в зависимости от фактических используемых правил кодирования. Детали правил кодирования вне области этой Рекомендации.

Определения типа для этих типов:

```
type integer short        (-32768 .. 32767)   with { variant "16 bit" };

type integer unsignedshort  (0 .. 65535)      with { variant "unsigned 16 bit" };
```

E.2.1.2 Знаковые и беззнаковые длинные целые числа

Эти типы поддерживают величины целого числа диапазона от -2 147 483 648 к 2 147 483 647 для знакового типа, и от 0 до 4 294 967 295 для типа беззнакового. Символика величины для этих типов - та же самая как символика величины для типа целого числа. Величины этих типов должны быть закодированы и расшифрованы как они были представлены на четырех байтах в пределах системы, независимо от фактической используемой формы представления.

ПРИМЕЧАНИЕ. – Кодирование величин этих типов может быть тем же самым или может отличаться друг от друга и от кодирования типа целого числа (основной тип этих полезных типов) в зависимости от фактических используемых правил кодирования. Детали правил кодирования вне области этой Рекомендации.

Определения типа для этих типов:

```
type integer long(-2147483648 .. 2147483647)
              with { variant "32 bit" };
type integer unsignedlong(0..4294967295)
              with { variant "unsigned 32 bit" };
```

E.2.1.3 Знаковые и беззнаковые longlong целые

Эти типы поддерживают величины целого числа диапазона от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 для знакового типа, и от 0 до 18 446 744 073 709 551 615 для типа беззнакового. Символика величины для этих типов - та же самая как символика величины для типа целого числа. Величины этих типов должны быть закодированы и расшифрованы как они были представлены на восьми байтах в пределах системы, независимо от фактической используемой формы представления.

ПРИМЕЧАНИЕ. – Кодирование величин этих типов может быть тем же самым или может отличаться друг от друга и от кодирования типа целого числа (основной тип этих полезных типов) в зависимости от фактических используемых правил кодирования. Детали правил кодирования вне области этой Рекомендации.

Определения типа для этих типов:

```
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 floats

Эти типы поддерживают Стандартные 754 ANSI/IEEE (см. Библиографию) для двойной арифметики с плавающей запятой. Тип IEEE 754 float поддерживает числа с плавающей запятой по основанию 10, экспонента размера 8, мантисса размера 23 и бит знака. Тип IEEE 754 double поддерживает числа с плавающей запятой с основой 10, экспонента размера 11, мантисса размера 52 и бит знака. Тип IEEE 754 `extfloat` поддерживает числа с плавающей запятой с основой 10, минимальная экспонента размера 11, минимальная мантисса размера 32 и бит знака. Тип IEEE 754 `extdouble` поддерживает числа с плавающей запятой с основой 10, минимальная экспонента размера 15, минимальная мантисса размера 64 и бит знака.

Величины этих типов должны быть закодированы и расшифрованы согласно определения IEEE 754 . Символика величины для этих типов - то же самое как символика величины для типа float (основание 10).

ПРИМЕЧАНИЕ. – Точное кодирование величин этого типа зависит от фактических используемых правил кодирования. Детали правил кодирования правил вне области этой Рекомендации.

Определения типа для этих типов:

```
typefloat   IEEE754float     with { variant "IEEE754 float" };
typefloat   IEEE754double    with { variant "IEEE754 double" };
typefloat   IEEE754extfloat  with { variant "IEEE754 extended float" };
typefloat   IEEE754extdouble     with { variant "IEEE754 extended double" };
```

E.2.2 Используемые типы строки символов

E.2.2.0 UTF-8 строка символов "utf8string"

Этот тип поддерживает весь набор символов TTCN-3 типа **`universal charstring`** (см. параграф d 6.1.1). Его характерные значения - ноль, один, или другие символы из этого набора. Величины этого типа должны быть полностью (например, каждый символ величины индивидуально) быть закодированными и расшифрованными согласно Формату Преобразования UCS 8 (UTF-8) как определено в Приложении R ISO/IEC 10646 [10]. Символика величины для этого типа - то же самое как символика величины для **`universal`** тип **`charstring`**.

Определение типа для этого типа:

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 BMP строка символов "bmpstring"

Этот тип поддерживает Basic Multilingual Plane (BMP) набор символов ISO/IEC 10646 [10]. BMP представляет все символы плоскости 00 из группы, из кодового набора символов Universal Multiple-octet. Его характерные значения – ноль, один, или более символов из BMP. Величины этого типа должны быть полностью (например, каждый символ величины индивидуально) быть закодированным и расшифрованным согласно кодировочной форме представления UCS-2 (см. 14.1 из ISO/IEC 10646 [10]). Символика величины для этого типа – то же самое как символика величины для **universal** тип **charstring**.

ПРИМЕЧАНИЕ. – Тип "bmpstring" поддерживает подмножество типа TTCN-3 **universal charstring**.

Определение типа для этого типа:

```
type universal charstring bmpstring  ( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
                    with { variant "UCS-2" };
```

E.2.2.2 UTF-16 строка символов "utf16string"

Этот тип поддерживает все символы самолетов 00 к 16 из группы, 00 из кодового набор символов Universal Multiple-octet (см. ISO/IEC 10646 [10]). Его характерные значения – ноль, один, или другие символы из этого набора. Величины этого типа должны быть полностью (например, каждый символ величины индивидуально) быть закодированными и расшифрованными согласно Формату Преобразования UCS 16 (UTF-16) как определено в Приложении Q ISO/IEC 10646 [10]. Символика величины для этого типа - то же самое как символика величины для **universal** тип **charstring**.

ПРИМЕЧАНИЕ. – Тип "utf16string" поддерживает подмножество типа TTCN-3 **universal charstring**.

Определение типа для этого типа:

```
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 строка символов "iso8859string"

Этот тип поддерживает все символы во всех алфавитах, определенных в многостороннем стандарте ISO/IEC 8859 (см. библиографию). Его характерные значения - ноль, один, или больше символов из наборов символов ISO/IEC 8859. Величины этого типа должны быть полностью (то есть, каждый символ величины индивидуально) быть закодированными и расшифрованными согласно закодированному представлению как определено в ISO/IEC 8859 (8-битовое кодирование). Символика величины для этого типа - то же самое как символика величины для **universal** тип **charstring**.

ПРИМЕЧАНИЕ 1. – Тип "iso8859string" поддерживает подмножество типа TTCN-3 **universal charstring**.

ПРИМЕЧАНИЕ 2. – В каждом ISO/IEC 8859 алфавите более низкая часть таблицы набора символов (положения от 02/00 до 07/14) совместима с набором символов ITU-T Rec. T.50 [9] . Следовательно, все специфические знаки языка определены только для верхней части таблицы символа (положения 10/00 к 15/15). Так как тип "iso8859string" определен как подмножество типа universal charstring из TTCN-3, любое закодированное представление символа любого ISO/IEC, 8859 алфавитов могут быть нанесены на карту в эквивалентный символ (символ с тем же самым закодированным представлением, когда кодируется на 8 битах) из Basic Latin или Latin-1 таблицы символов Приложения ISO/IEC 10646 [10].

Определение типа для этого типа:

```
type universal charstring iso8859string  ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                        with { variant "8 bit" };
```

E.2.3 Используемые структурированные типы

E.2.3.0 Десятичный литерал с фиксированной точкой

Этот тип поддерживает использование десятичного литерала с фиксированной точкой как определено в версии 2.6 Синтаксиса и Семантики IDL (см. библиографию). Он определен частью целого числа, десятичной запятой и дробной частью. Целое число и дробные части оба состоят из последовательности десятичного числа (по основанию 10), цифры. Число цифр сохранено в "цифрах", и размер дробной части дан в "масштабе". Сами цифры сохранены в "value_". Символика величины для этого типа - то же самое как символика величины для типа record. Величины этого типа должны быть закодированы и расшифрованы как десятичная цифра с фиксированной точкой IDL.

ПРИМЕЧАНИИ. – Точное кодирование величин этого типа зависит от фактических используемых правил кодирования. Детали правил кодирования вне области этой Рекомендации.

Определение типа для этого типа:

```
type record IDLfixed {
               unsignedshort digits,
               short scale,
               charstring  value_
               }
            with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 Используемые неделимые типы строки

E.2.4.1 Единичный тип символа IRV

Тип, характерные значения которого - единичные символы версии ITU-T Rec. T.50 [9] подчинение к Международной Версии Ссылки (IRV) как определено в 8.2/T.50 [9] (см. также Примечание 2 из 6.1.1).

Определение типа для этого типа:

```
type charstring char length (1);
```

ПРИМЕЧАНИЕ 1. – название этого используемого типа - то же самое, поскольку в форме quadraple TTCN-3 для обозначения использует ключевое слово **universal charstring**. Вообще, не позволительно, использовать ключевые слова TTCN-3 как идентификаторы. Используемый тип "char" - отдельное исключение и разрешено только для обратной совместимости с предыдущими версиями стандарта TTCN-3.

ПРИМЕЧАНИЕ 2. – специальная строка "8 битов", определенных в 28.2.3, может использоваться с этим типом, чтобы описать данное кодирование его величины. Кроме того, могут быть изменены другие свойства основного типа при использовании механизмов признака.

E.2.4.2 Единичный тип универсального символа

Тип, характерные величины которого - единичные символы от ISO/IEC 10646 [10].

Определение типа для этого типа:

```
type universal charstring uchar length (1);
```

ПРИМЕЧАНИЕ. – С этим типом могут использоваться специальные строки, определенные в 28.2.3, кроме "8 битов", чтобы определить данное кодирование его величины. Кроме того, могут быть изменены другие свойства основного типа при использовании механизмов признака.

E.2.4.3 Единичный тип бита

Тип, характерные значения которого - единичные двоичные знаки.

Определение типа для этого типа:

```
type bitstring bit length (1);
```

E.2.4.4 Единичный шестнадцатиричный тип

Тип, характерные значения которого - единичные шестнадцатеричные цифры.

Определение типа для этого типа:

```
type hexstring hex length (1);
```

E.2.4.5 Единичный тип октета

Тип, характерные значения которого - пары шестнадцатеричных цифр.

Определение типа для этого типа:

```
type octetstring octet length (1);
```

Приложение F (для информации)

Операции на активных объектах TTCN-3

F.1 Общие положения

Это приложение описывает в краткой форме семантику операций на активных объектах в TTCN-3, которыми являются тестовые компоненты, таймеры и порты. Это динамическое поведение описано в форме машин состояния и:

- состояния, называемые и идентифицированные как узлы;
- начальное состояние, идентифицируемое входящей стрелкой;
- переходы между состояниями, соединяющими два состояния (не обязательно различные состояния) и идентифицированные как указатель;
- переходы, помеченные с условием предоставления возможности для того перехода (то есть, операция или запросы выражения) и получающимся условием (например тестовая ошибка случая), оба отделены "/":
  - операция и запросы выражения – операции TTCN-3 и выражения, применимые к объекту (выделенный жирным);
  - ошибка как получающееся условие – означает ошибку тестового примера (выделенный жирным);
  - пустой указатель как получающееся условие – означает, что за исключением возможного изменения состояния, никакие другие результаты не применяются (выделенный жирным);
  - соответствие/нет соответствия – обращается к соответствующему результату перехода (выделенный жирным);
  - конкретные величины являются Булевыми или float значениями (написанный в полужирном курсиве);
  - все другие получающиеся условия описываются в текстовом виде (написанный в стандартном шрифте);
- используются примечания , чтобы объяснить дальнейшие детали машины состояний.

Для дальнейших деталей, пожалуйста, обратитесь к эксплуатационной семантике TTCN-3 [3]. В случае любого противоречия между этим приложением и эксплуатационной семантикой TTCN-3 [3], последний имеет приоритет .

F.2 Тестовые компоненты

F.2.1 Ссылки тестовых компонентов

Чтобы сослаться на тестовые компоненты, используются операции **`self`** и **`mtc,`** переменные типов тестовых компонентов. Операции **`start`**, **`stop`**, **`done`** and **`running`** непосредственно не применены на тестовые компоненты, но на составляющие ссылки. Тестовая система должна решить, должна ли операция, которую требуют, произвести компонентный объект непосредственно или является соответствующим другое действие (например, происходит ошибка, когда ссылка остановленного PTC используется в компонентной операции начала). Операция **`create`**, используемая, чтобы создать PTCs, возвращает уникальную ссылку на созданный PTC, который обычно связан с тестовой компонентной переменной. Поведение, связанное с самими компонентными переменными, показано на рисунке F.1.

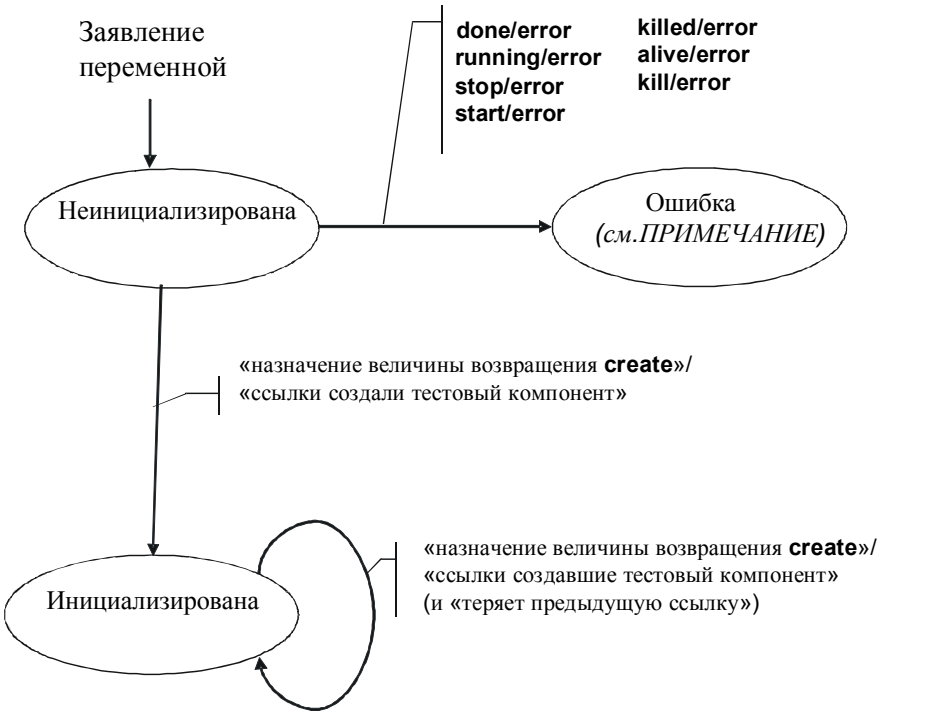
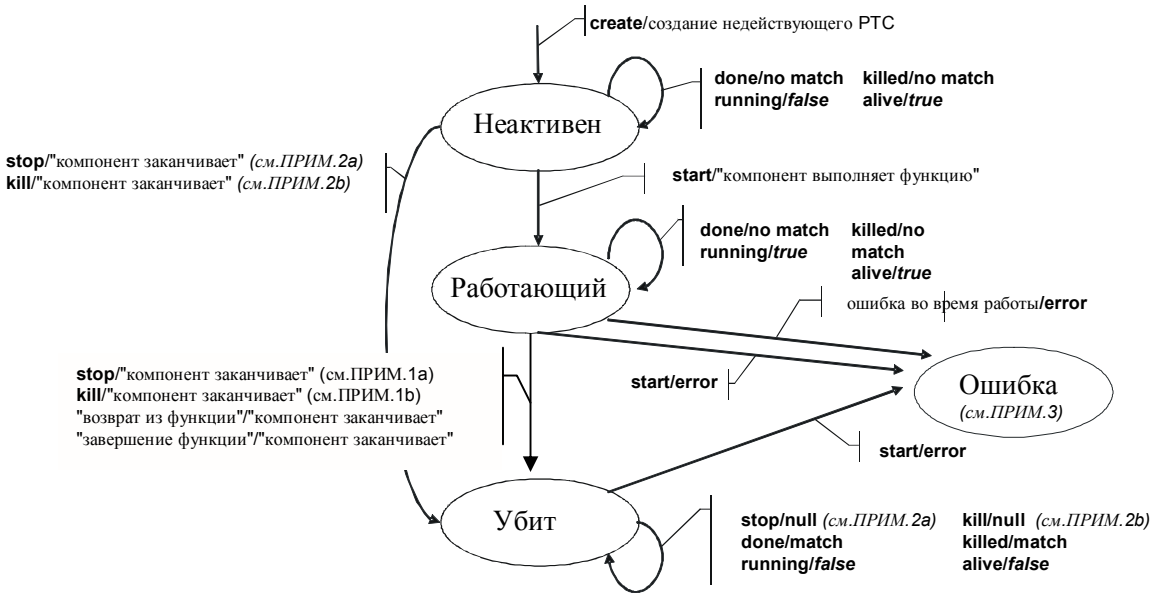


Рисунок F.1/Z.140 - Обработка ссылок тестовых компонентов

F.2.2 Динамическое поведение PTCs

PTCs может иметь недействующий или действующий тип. Недействующий тип PTCs может быть в Неактивном, Рабочем и Убитом состоянии. Их динамическое поведение показано на Рисунке F.2.



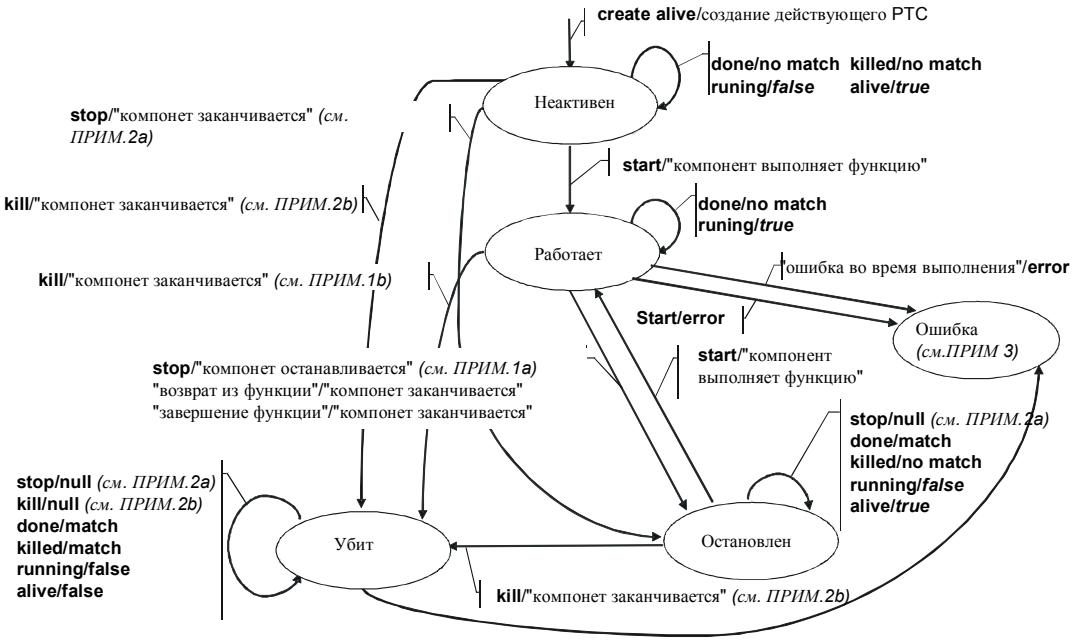
ПРИМЕЧАНИЕ 1 - a)Stop может быть либо самоостановкой, либо остановка благодаря другому тестовому компоненту;
b)Kill может быть либо самоубийством, либо убийством благодаря другому тестовому компоненту или тестовой системой (в случае ошибки).

ПРИМЕЧАНИЕ 2 - a)Stop может быть только остановкой благодаря другому тестовому компоненту;
b)Kill может быть только убийством благодаря другому тестовому компоненту или тестовой системой (в случае ошибки).

ПРИМЕЧАНИЕ 3 - Всякий раз, когда тестовый компонент вступает в свое состояние ошибки, локальному заключению приписывается заключение ошибки. Тестовый случай заканчивается, и общим результатом тестового случая будет ошибка.

Рисунок F.2/Z.140 - Динамическое поведение недействующего типа PTC

Действующий тип PTCs может быть в Неактивном, Рабочем и Убитом состоянии. Их динамическое поведение показано на Рисунке F.3.



ПРИМЕЧАНИЕ 1 - a)Stop может быть либо самоостановкой, либо остановка благодаря другому тестовому компоненту;
b)Kill может быть либо самоубийством, либо убийством благодаря другому тестовому компоненту или тестовой системой (в случае ошибки).

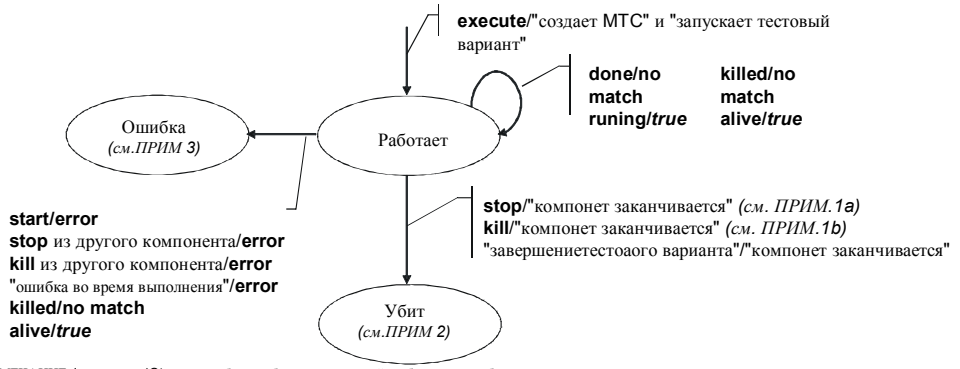
ПРИМЕЧАНИЕ 2 - a)Stop может быть только остановкой благодаря другому тестовому компоненту;
b)Kill может быть только убийством благодаря другому тестовому компоненту или тестовой системой (в случае ошибки).

ПРИМЕЧАНИЕ 3 - Всякий раз, когда тестовый компонент вступает в свое состояние ошибки, локальному заключению приписывается заключение ошибки. Тестовый случай заканчивается, и общим результатом тестового случая будет ошибка.

Рисунок F.3/Z.140 - Динамическое поведение действующего типа PTC

F.2.3 Динамическое поведение MTC

MTC может быть в Рабочем или Убитом состоянии. Динамическое поведение MTC показано на Рисунке F.4.



ПРИМЕЧАНИЕ 1 - a)Stop может быть либо самоостановкой, либо остановка благодаря другому тестовому компоненту;
b)Kill может быть либо самоубийством, либо убийством благодаря другому тестовому компоненту или тестовой системой (в случае ошибки).

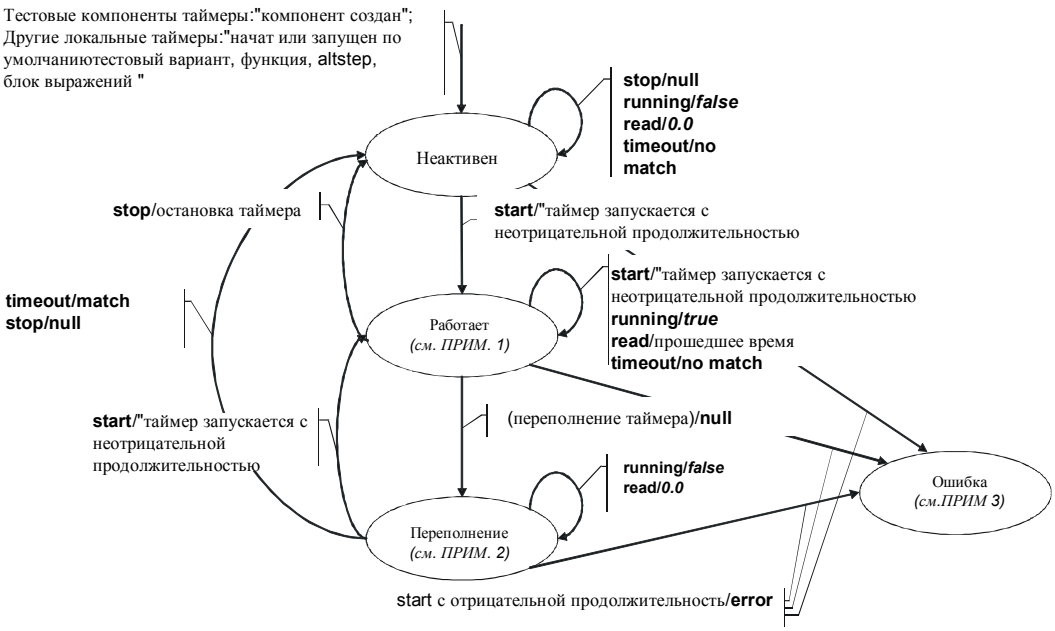
ПРИМЕЧАНИЕ 2 - Все оставшиеся PTC дложны быть уничтожены, как только закончится тестовый вариант

ПРИМЕЧАНИЕ 3 - Всякий раз, когда MTC входит в свое состояние ошибки, локальному заключению приписывается заключение ошибки, тестовый случай заканчивается, и общим результатом тестового случая будет ошибка.

Рисунок F.4/Z.140 - Динамическое поведение MTC

F.3 Таймеры

Таймеры могут быть в Неактивном, Рабочем или Переполненном состояние. Динамическое поведение таймера показано на Рисунке F.5.



ПРИМЕЧАНИЕ 1 -Для любой области наблюдения, все таймеры в той области, находящейся в рабочем состоянии, формируют список работающих таймеров

ПРИМЕЧАНИЕ 2 -Для любой области наблюдения, все таймеры в той области, находящейся в переполненном состоянии, формируют список таймаута

ПРИМЕЧАНИЕ 3 -Всякий раз, когда таймер попадает в состояние ошибки, тестовый компонент, которому он принадлежит тоже переходит в состоянии ошибки,

определяя локальный вердикт ошибки,тестовый вариант заканчивается и общим результатом тестового случая будет ошибка.

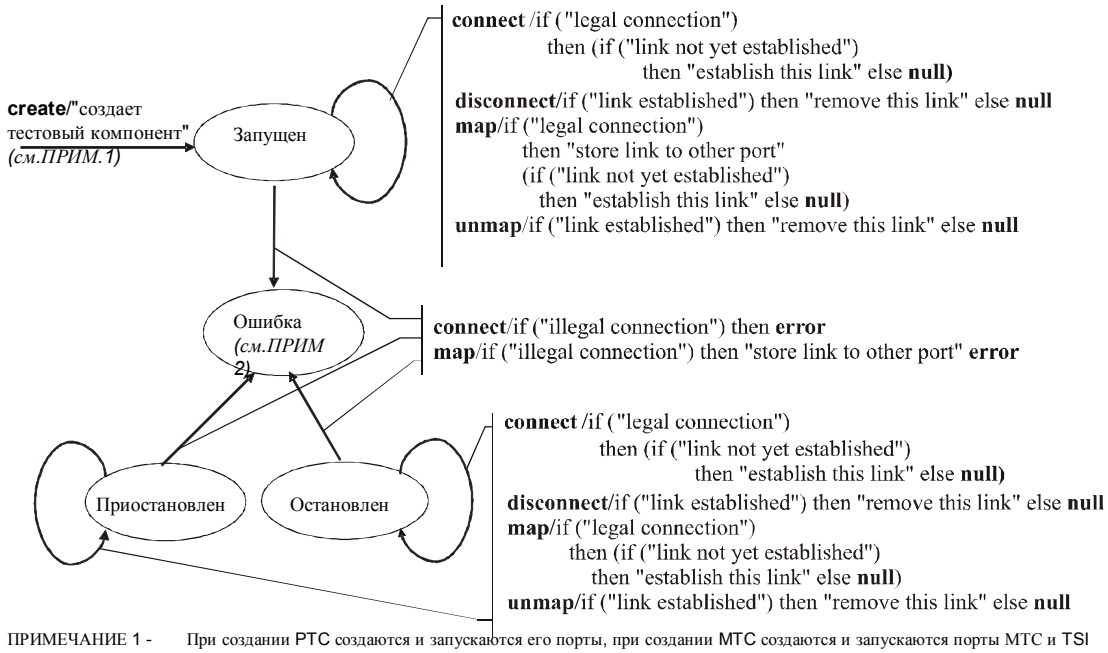
Рисунок F.5/Z.140 - Динамическое поведение таймеров

F.4 Порты

Порты могут быть в Запущенном или Остановленном состоянии. Поскольку их поведение довольно сложно, машина состояний была разделена на машину состояний, дающую динамическое поведение операций конфигурации (то есть, соедините, разъедините, нанесите на карту и нанесите на карту), операций управления порта (то есть, начнитесь, остановитесь, и ясный) и операций коммуникации (то есть, пошлите, получите, вызовите, getcall, поднимите, поймайте, ответьте, getreply, и чек). Поскольку спусковой механизм - стенография для высокого звука вместе с, получают, это не рассматривают здесь.

F.4.1 Операции Конфигурации

Операции конфигурации порта (то есть, connect, disconnect, map и unmap) безразличны к состоянию порта. Они показывают поведение, показанное на Рисунке F.6.



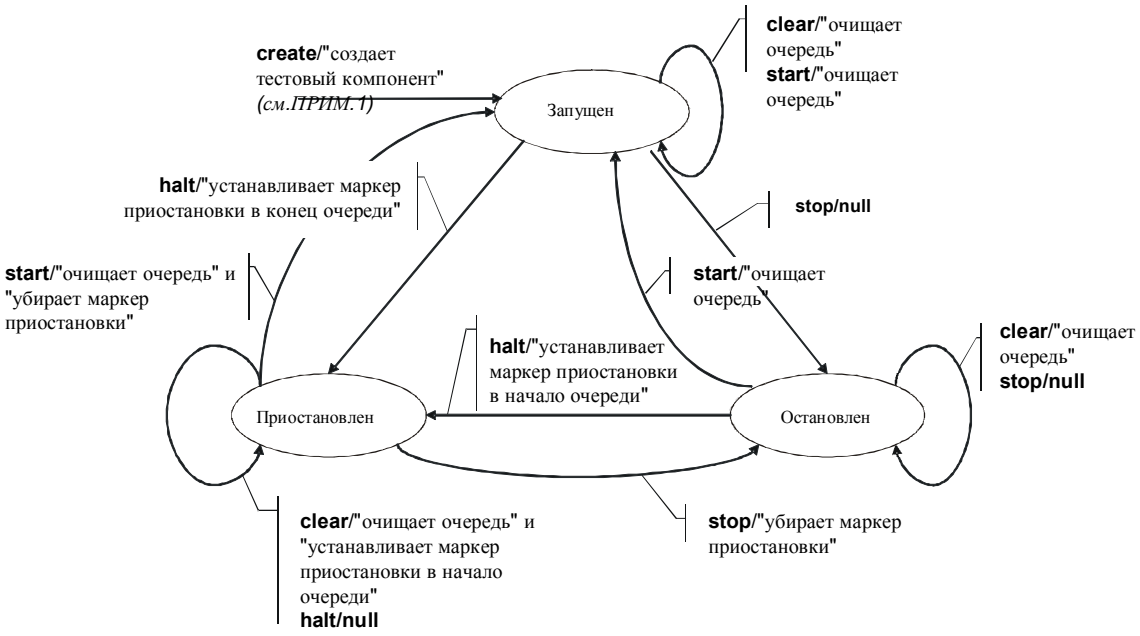
ПРИМЕЧАНИЕ 1 - При создании PTC создаются и запускаются его порты, при создании MTC создаются и запускаются порты MTC и TSI
ПРИМЕЧАНИЕ 2 - Всякий раз, когда порт попадает в состояние ошибки, тестовый компонент, которому он принадлежит тоже переходит в состоянии ошибки, определяя локальный вердикт ошибки,тестовый вариант заканчивается и общим результатом тестового случая будет ошибка.

Рисунок F.6/Z.140 - Динамическое поведение портов: операции конфигурации порта

Переходы не изменяют главное состояние порта; то есть, порт остается в Запущенном или Остановленном состоянии.

F.4.2 Операции управления Портом

Результаты операций управления порта показаны на Рисунке F.7.

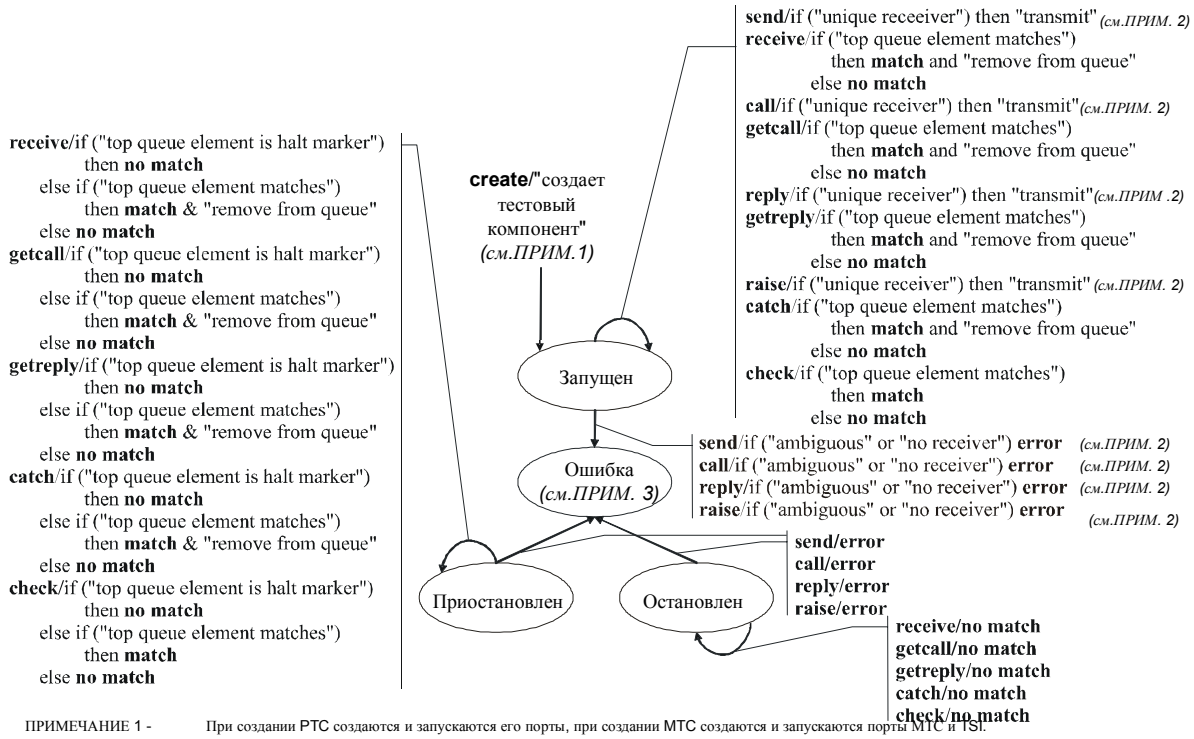


ПРИМЕЧАНИЕ - При создании PTC создаются и запускаются его порты, при создании MTC создаются и запускаются порты MTC и TSI
ПРИМЕЧАНИЕ 1 - При создании PTC создаются и запускаются его порты, при создании MTC создаются и запускаются порты MTC и TSI
ПРИМЕЧАНИЕ 2 - Всякий раз, когда порт попадает в состояние ошибки, тестовый компонент, которому он принадлежит тоже переходит в состоянии ошибки, определяя локальный вердикт ошибки,тестовый вариант заканчивается и общим результатом тестового случая будет ошибка.

Рисунок F.7/Z.140 – Динамическое поведение портов: операции управления порта

F.4.3 Операции Коммуникации

Результаты коммуникационных операций **`send, receive, call, getcall, raise, catch, reply, getreply, check`** показаны на Рисунке F.8.



ПРИМЕЧАНИЕ 1 - При создании PTC создаются и запускаются его порты, при создании MTC создаются и запускаются порты MTC и TSI.

ПРИМЕЧАНИЕ 2 - Уникальный приемник существует в том случае, если для этого порта существует только одно соединение, или же для адресации ссылок на выражения для тестового компонента, чей порт соединен с данным портом (завершенный тестовый компонент не является разрешенным приемником).

ПРИМЕЧАНИЕ 3 - Всякий раз, когда порт попадает в состояние ошибки, тестовый компонент, которому он принадлежит тоже переходит в состоянии ошибки, определяя локальный вердикт ошибки,тестовый вариант заканчивается и общим результатом тестового случая будет ошибка.

ПРИМЕЧАНИЕ 4 - Так как триггер является сокращенным обозначением для alt, объединенного с receive (прием), то здесь он не рассматривается.

Рисунок F.8/Z.140 - Динамическое поведение портов: операции коммуникации

Приложения G (для информации)

Исключенные языковые особенности

G.1 Групповое определение стиля параметров модуля

Предыдущая версия этой Рекомендации потребовала использования группоподобного синтаксиса, который, как показано в примере ниже, объявил параметры модуля. Синтаксис параметра модуля был объединен с постоянным и переменным синтаксисом декларации в этой версии, но группоподобный синтаксис не удален полностью, чтобы дать временя для поставщиков инструмента и пользователей, чтобы измениться от старого синтаксиса к новому. В следующем изданном выпуске этой Рекомендации запланировано удалить полностью группоподобный синтаксис деклараций параметра модуля .

ПРИМЕР (излишний синтаксис):

```
module MyModuleWithParameters
{
    modulepar { integer TS_Par0, TS_Par1 := 0;
                boolean TS_Par2 := true
              };
    modulepar { hexstring TS_Par3 };
}
```

G.2 Рекурсивный импорт

Предыдущая версия этой Рекомендации позволяет импортировать неявно назначенные определения, посредством импортирования других определений того же самого модуля, используя их в рекурсивном способе. В этой версии не рекомендуются пользоваться этим, и планируются, что в следующей изданной версии эта возможность будет удалена.

G.3 Использование `all` в определении типа порта

Предыдущая версия этой Рекомендации позволяла использовать ключевое слово **`all`** в определениях типа порта вместо явного списка типов и подписей, позволенных через данный порт. В этой версии не рекомендуется пользоваться этим, и планируется, что в следующей изданной версии эта возможность будет удалена.

Библиография

- ETSI ES 201 873-1 V1.1.2 (2001-06), *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language.*
- ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.*
- ISO/IEC 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1.*
- Object Management Group (OMG): *The Common Object Request Broker: Architecture and Specification, Chapter 3 – IDL Syntax and Semantics. Version 2.6, FORMAL/01-12-01, December 2001.*
- IEEE 754 (1985*), Binary Floating-Point Arithmetic.*

СЕРИИ РЕКОМЕНДАЦИЙ МСЭ-Т

Серия A	Организация работы МСЭ-Т
Серия D	Общие принципы тарификации
Серия E	Общая эксплуатация сети, телефонная служба, функционирование служб и человеческие факторы
Серия F	Нетелефонные службы электросвязи
Серия G	Системы и среда передачи, цифровые системы и сети
Серия H	Аудиовизуальные и мультимедийные системы
Серия I	Цифровая сеть с интеграцией служб
Серия J	Кабельные сети и передача сигналов телевизионных и звуковых программ и других мультимедийных сигналов
Серия K	Защита от помех
Серия L	Конструкция, прокладка и защита кабелей и других элементов линейно-кабельных сооружений
Серия M	Управление электросвязью, включая СУЭ и техническое обслуживание сетей
Серия N	Техническое обслуживание: международные каналы передачи звуковых и телевизионных программ
Серия O	Требования к измерительной аппаратуре
Серия P	Качество телефонной передачи, телефонные установки, сети местных линий
Серия Q	Коммутация и сигнализация
Серия R	Телеграфная передача
Серия S	Оконечное оборудование для телеграфных служб
Серия T	Оконечное оборудование для телематических служб
Серия U	Телеграфная коммутация
Серия V	Передача данных по телефонной сети
Серия X	Сети передачи данных, взаимосвязь открытых систем и безопасность
Серия Y	Глобальная информационная инфраструктура, аспекты протокола Интернет и сети последующих поколений
Серия Z	**Языки и общие аспекты программного обеспечения для систем электросвязи**

Отпечатано в Швейцарии
Женева, 2009 г.